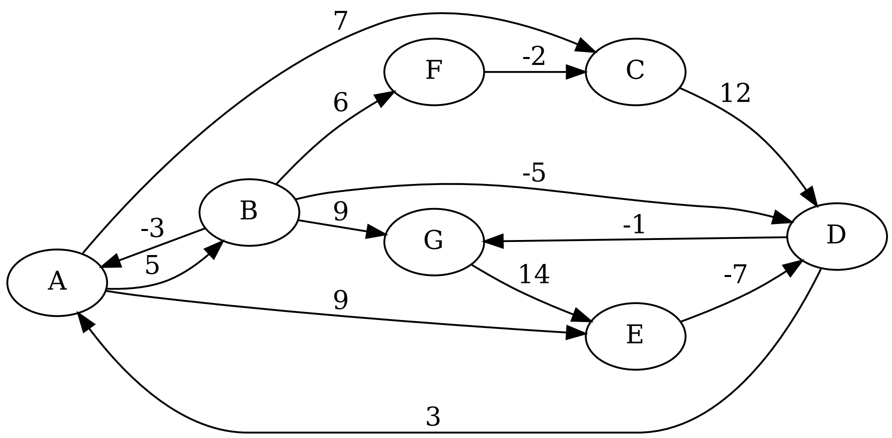
TD 6: APSP

1 Execute Johnson's algorithm

Execute Johnson's algorithm on the graph below.



Solution:

We add a new vertex S to the graph and arcs of weight 0 from s to all vertices. We now execute Bellman-Ford's algorithm. Initially, distances from s are set to ∞, but we can speed things up by initializing such distances to 0 (this is equivalent to first checking if the arcs from s are shortcuts).

We have the following list of arcs:

Arc	Weight
AB	5
AC	7
AE	9
BA	-3
BD	-5
BF	6
BG	9
CD	12
DA	3
DG	-1
ED	-7
FC	-2
GE	14

We update the distances from s as follows:

		BA	BD	DG	ED	FC	DA	DG
A	0	-3					-4	
B	0							
C	0					-2		
D	0		-5		-7			
E	0							
F	0							
G	0			-6				-8

We now calculate new weights for all the arcs setting $w'(ij) = w(ij) + \mathrm{dist}(s,i) - \mathrm{dist}(s,j)$.

Arc	Weight	New weight
AB	5	1
AC	7	5
AE	9	5
BA	-3	1
BD	-5	2
BF	6	6
BG	9	17
CD	12	17
DA	3	5
DG	-1	0
ED	-7	0
FC	-2	0
GE	14	6

Now what remains is to execute Dijkstra from each vertex, since all weights are non-negative. Details are left to the reader…

2 Shortest Cycles

We are given as input a directed graph $G = (V, A)$ with positive weights on the arcs in adjacency list form (that is, you can assume that for each vertex you have a list of outgoing arcs and their corresponding weights).

- Give an efficient algorithm that takes as input an arc $ij \in A$ and outputs the shortest cycle that contains ij (or correctly reports that none exists).
- Give an efficient algorithm that outputs the shortest cycle in G.

How does your answer change if the graph may have negative weights? (you are promised that no negative cycles exist)

Solution:

- The shortest cycle that contains ij must also contain a shortest path from j to i. We execute Dijkstra's algorithm starting from j and find a shortest $j \to i$ path. Together with the arc ij this forms a cycle. The complexity is $O(m \log n)$ (executing Dijkstra).
- One idea is to execute the algorithm of the previous question for every arc and take the minimum. This gives complexity $O(m^2 \log n)$. Can we do better?

  We calculate APSP on G (for instance by running Dijkstra's algorithm n times). Now, for each $ij \in A$ we look up $\mathrm{dist}(j,i)$ and compute the shortest cycle going through ij in time $O(1)$. We output the minimum. Hence, the total complexity is now $O(nm \log n)$.

We observe that the latter problem does not change significantly if we have negative weights: it suffices to execute Johnson's algorithm for APSP, rather than directly running Dijkstra n times. For the former problem, we would still need to calculate the shortest $j \to i$ path. The algorithm we have seen in class (Bellman-Ford) is significantly slower than Disjkstra's; however, as we discussed, linear-time ($O(m \log^{O(1)} n)$-time) algorithm also exist for this case.

Interesting note: the algorithm we gave for finding a shortest cycle takes cubic time $O(n^3)$ for dense graphs. It is known that this can be improved **if and only if** APSP can be solved in sub-cubic time, so this algorithm may well be optimal.

3 Many distances

We are given as input a directed graph $G = (V, A)$ with positive arc weights in adjacency list form. Furthermore, we are given a set of k special vertices $S \subseteq V$. We define the S-distance of a vertex $x \in V$ as $\mathrm{dist}_S(x) = \min_{s \in S} \mathrm{dist}(x, s)$. Give an efficient algorithm that computes the S-distances of all vertices.

To motivate the problem, consider the following scenario: G represents the road network between different cities, arc weights represent distances, and S represents a set of cities that contain an important facility (e.g. a large hospital). For other cities, we want to calculate what is the minimum distance we need to traverse to reach *some* important facility.

Solution:

1. Obvious idea: Solve APSP on G, for each $x \in V \setminus S$ compute $\mathrm{dist}_S(x) = \min_{s \in S} \mathrm{dist}(x, s)$ in time $O(n)$, so in total this takes $O(nm \log n + n^2)$ (for running Dijkstra n times to solve APSP).

2. Slightly better idea: define the SINGLE TARGET SHORTEST PATH problem as the problem of computing the shortest path distance from any vertex to a specific destination t. This problem can be solved using Dijkstra's algorithm on the transpose of G (keeping the same weights). Solve this problem k times, once for each vertex of S. Then, for each $x \in V \setminus S$, compute $\mathrm{dist}_S(x) = \min_{s \in S} \mathrm{dist}(x, s)$ in time $O(k)$. This more careful analysis gives complexity $O(km \log n)$, since we run Dijkstra k times and $k < n$.

3. Even better idea: the problem can in fact be solved by solving STSP **once**. Add to the graph a new vertex s_0 and arcs of weight 0 from all vertices of S to s_0. We now run Dijkstra to solve STSP with target s_0 and the distance calculated for each x will be the minimum distance from x to S.

4 (Min,+)-Product

4.1 The basics

Suppose we are given two matrices A, B with dimensions $n_1 \times n_2$ and $n_2 \times n_3$ respectively. Then, the $(\min, +)$-product of A, B, denoted $A \otimes B$ is defined as the $n_1 \times n_3$ matrix C satisfying the following:

$$C[i, j] = \min_{k \in \{1, \dots, n_2\}} \{A[i, k] + B[k, j]\}, \ \ \forall i \in \{1, \dots, n_1\}, j \in \{1, \dots, n_3\}$$

Recall the following facts, discussed in class:

- Given two $n \times n$ matrices A, B, $A \otimes B$ can be computed in $O(n^3)$ time (assuming arithmetic operations take time $O(1)$).
- Assume we have an algorithm that can calculate $A \otimes A$, where A is an $n \times n$ matrix, in time $O(T)$. Then, we can solve APSP on weighted directed graphs without negative cycles in time $O(T \log n)$.

Solution:

- A simple algorithm consisting of three nested loops is given in the course slides.

- As discussed in class, we can define $A_\ell[i,j]$ to be the length of the shortest path from i to j using at most ℓ arcs. $A_1[i,j]$ is given by the edge weights (with missing weights replaced by ∞). We then observe that $A_\ell = A\ell/2 \otimes A_{\ell/2}$, because the shortest $i \to j$ path using at most ℓ arcs must have a mid-point k, breaking the path down into two paths, each using at most $\ell/2$ arcs (we assume for simplicity that ℓ is even, though this does not affect the analysis much).

4.2 (min,+) and APSP equivalence

As discussed in class, the question of whether APSP can be solved in time $O(n^{3-\varepsilon})$ is one of the most important open problems in theoretical computer science. The above indicate that one approach to obtain such an algorithm would be to design a faster algorithm for $(\min,+)$-product. This approach, however, presents some serious challenges (discussed again below). One could, then, hope, that perhaps we can obtain a faster APSP algorithm through some other means.

- Prove that it is in fact impossible to obtain a sub-cubic APSP algorithm without improving upon the best algorithm for $(\min,+)$-product. More precisely, show that if there is an algorithm solving APSP in time $O(n^{3-\varepsilon})$, then such an algorithm also exists for computing the $(\min,+)$-product of two $n \times n$ matrices. (Hint: given two matrices, construct a graph such that the result is hidden in the APSP matrix of the graph).

Solution:

Given two $n \times n$ matrices A, B we construct a graph G as follows: G has $3n$ vertices labeled $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, $c_1, \ldots, c_n$. The end goal is that the APSP matrix of G will satisfiy that $\text{dist}(a_i, b_j) = A \oplus B[i,j]$, for all i, j, which will allow us to claim that if we can compute APSP on G, we can look at a sub-area of the result and obtain the $(\min,+)$-product of A and B.

We now place arcs from all vertices a_i to all vertices c_k, and from all vertices c_k to all vertices b_j (all missing arcs can be considered as arcs of weight ∞). We set $w(a_i c_k) = A[i,k]$ and $w(c_k b_j) = B[k,j]$.

We now observe that in the new graph, $\text{dist}(a_i, b_j) = \min_k\{w(a_i c_k) + w(c_k b_j)\} = A \otimes B[i,j]$ as desired.

4.3 (min,+) squares

So far we have tried (and failed) to improve upon the fastest APSP algorithm by using $(\min,+)$-matrix multiplication. One could, however, object that there is an angle we have neglected: in order to speed up the best APSP algorithm it is not necessary to speed up $(\min,+)$-multiplication in general; rather, it is sufficient to have a faster algorithm for computing **squares**. In other words, we can focus on the special case of the problem where $A = B$.

- Show that this idea does not help. That is, there is an algorithm which can compute $A \times B$, for two $n \times n$ matrices, in time $O(T)$, if and only if there is such an algorithm for the case $A = B$.

Solution:

We show that matrix squaring is not easier than general matrix multiplication for both the normal matrix product and the $(\min,+)$-product. Suppose we are given $n \times n$ matrics A, B and wish to compute their product. We define

$$C = \begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}$$

which gives

$$C^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$$

Similarly for the $(\min,+)$-product we can define

$$C = \begin{bmatrix} \infty & A \\ B & \infty \end{bmatrix}$$

which gives

$$C \otimes C = \begin{bmatrix} A \otimes B & \infty \\ \infty & B \otimes A \end{bmatrix}$$

In both cases C is a $2n \times 2n$ matrix, where two quadrants have been replaced by the neutral element of the corresponding operation (0 for addition, or ∞ for $\min$). If squaring matrices were easy, then we could use this trick to compute arbitrary products.

4.4 (min,+) to normal products

Recall that, as we discussed in class, the usual matrix multiplication operations (where $\min$ is replaced by $+$ and $+$ is replaced by $\times$) does have sub-cubic algorithms (notably, Strassen's algorithm). Unfortunately, it does not seem possible to use such algorithms to compute the $(\min, +)$-product faster than time $O(n^3)$, because such algorithms rely on the ability to perform subtractions. In the $(\min, +)$ case we would therefore need a function that is the inverse of $\min$. Let us explore an idea that may allow us to do this.

We are given two $n \times n$ matrices A, B, and let W be the largest absolute value of any entry in A, B. Consider the following algorithm to compute $A \otimes B$:

1. Compute that matrix A' with $A'[i,j] = (n+1)^{W-A[i,j]}$. Similarly, compute B'.
2. Compute $C' = A' \cdot B'$ (using the normal matrix product).
3. For each $i, j \in \{1, \dots, n\}$ compute $C[i,j]$ as follows: let x be the maximum integer such that $(n+1)^x \leq C'[i,j]$; then set $C[i,j] := 2W - x$.

- Prove that the algorithm above correctly calculates $A \otimes B$.
- Explain why the algorithm above still does not lead to a faster APSP algorithm (even though step 2 can be performed using Strassen's algorithm).

Solution:

- The algorithm can be shown correct as follows. Fix i, j and we want to claim that the value $C[i,j]$ we compute is equal to $q := \min_k \{A[i,k] + B[k,j]\}$. For C' we have: $C'[i,j] = \sum_k A[i,k]B[k,j] = \sum_k (n+1)^{2W-A[i,k]-B[k,j]}$. We observe that $n(n+1)^{2W-q} \geq C'[i,j] \geq (n+1)^{2W-q}$. Indeed, the sum we calculated contains at least one term with value $(n+1)^{2W-q}$ and all terms are at most $(n+1)^{2W-q}$, as the exponent is maximized when $A[i,k] + B[k,j]$ is minimized. We conclude that

$$(n+1)^{2W-q+1} > C'[i,j] \geq (n+1)^{2W-q}$$

so the largest integer x such that $(n+1)^x \leq C'[i,j]$ satisfies

$$2W - q + 1 > x \geq 2W - q \Rightarrow x = 2W - q \Rightarrow q = 2W - x$$

- Step 2 of the algorithm can indeed be implemented with n^ω arithmetic operations, where $\omega < 2.4$ is the complexity of matrix multiplication. The problem, however, is that arithmetic operations cannot be assumed to take constant time. Observe that the integers involed are of the order n^{2W}, so require $O(W \log n)$ bits to store. Naively multiplying such integers takes time $O(W^2 \log^2 n)$, but even faster multiplication algorithms take time $\Omega(W \log n)$ (we have to at least read the integers!). The question is then, what is the value of W we can expect. Unfortunately, even for unweighted graphs, we could have

$W = \Theta(n)$, as W will depend on the shortest path distance between any two vertices. Therefore, the arithmetic operations will add a factor polynomial in n to the complexity of our algorithm, more than offseting the gain of using Strassen's (or similar) algorithms for matrix multiplication. Nevertheless, the idea given here may be useful in cases where W is small, and indeed, APSP in unweighted graphs can indeed be computed in n^ω time (see Seidel's algorithm), albeit with a more sophisticated argument.