Exercise 16.6. Let $G = (V, E)$ be an undirected graph with distinct nonnegative edge weights $w : E \to \mathbb{R}$. For a spanning tree T, we say that the *bottleneck weight of* T is the maximum weight edge in T, $\max_{e \in T} w(e)$.

Exercise 16.6.1. Prove that the MST is also a minimum bottleneck weight spanning tree of G.

Solution. Let α be the MST of G. Assume ad absurdum that there exists an minimum bottleneck weight spanning tree (MBWST) β of G such that $\alpha \neq \beta$. Then by definition of bottleneck weight, we have that

$$\max_{e \in \alpha}\{w(e)\} > \max_{e \in \beta}\{w(e)\}.$$

That is to say, there exists some edge $e = (v_1, v_2) \in \alpha$ such that $w(e)$ is greater than $w(e')$ for every $e' \in \beta$. Obviously $e \notin \beta$, but by definition of spanning tree β spans e. Since α does not contain any cycles by definition of tree, WLOG there exists some $f = (v_1, v_3) \in \beta$ such that $f \notin \alpha$. Then we have that

$$w(e) > w(f).$$

By the spanning property of α, we know there exists an edge $(v_3, v_k) \in \alpha$ for some arbitrary vertex v_k. Thus by removing e and adding f to α, we can preserve the spanning property of α while reducing its total weight, contradicting that α is the MST of G. Thus the minimum spanning tree of G must also be a minimum bottleneck spanning tree of G. □

Exercise 16.6.2. Design and analyze a $O(m+n)$-time algorithm for computing a minimum bottleneck weight spanning tree of G. (This is faster than any of our algorithms for MST.)[4]

[4]Here's step 1: compute the median edge weight in $O(m)$ time.

Solution. □

Exercise 17.3. Let $G = (V, E)$ be an undirected graph, and let $a, b, c \in V$ be three distinct vertices. We define an (a, b, c)-path as a path from a to c that goes through b. Consider the problem of deciding if there exists an (a, b, c)-path. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that the problem is NP-Hard.

Solution. This problem has a polynomial-time solution.

Abstractly, if we can find two simple paths in G—one from a to b and one from b to c—that are vertex-disjoint except for at vertex b, then we can concatenate these two paths to get an (a, b, c)-path.

To do this, we construct an auxiliary digraph G'.

First, to force vertex-disjointedness, we split each vertex $v \in V$ into two vertices, v_{in} and v_{out} and assign a directed edge from v_{in} to v_{out}. Then for each undirected edge $\{u, v\}$ in G, we add two directed edges (u_{out}, v_{in}) and (v_{out}, u_{in}). Finally, we assign b_{out} to be the source vertex and direct both a_{out} and c_{out} to a single destination vertex t.

Now, finding two edge-disjoint paths from b_{out} to t in G' will guarantee vertex-disjoint paths from b to a and b to c in G. The former path can be reversed and concatenated to obtain an (a, b, c)-path.

`abc-path`(G, a, b, c)`:`

`/*` given $a, b, c \in G$ in an undirected graph, computes whether an (a, b, c)-path exists in G. `*/`

1. Construct the auxiliary digraph G':
   A. For every vertex $v \in V$, create vertices v_{in} and v_{out} and add the edge (v_{in}, v_{out}).
   B. For every undirected edge $\{u, v\} \in E$, add the edges (u_{out}, v_{in}) and (v_{out}, u_{in}).
   C. Let $s \leftarrow b$.
   D. Create a sink vertex t, and add edges from a_{out} and c_{out} to t.
2. If `augmenting-paths`$(G', s, t) = 2$ then
   A. Return `true`.
3. Return `false`.

Runtime. We use `augmenting-paths`(G', s, t) to compute the maximal number of edge-disjoint (s, t)-paths. Note that we can slightly modify `augmenting-paths()` to return true immediately when we find 2 such paths so that the runtime is in $O(m)$. ■

Correctness. We now prove that an (a, b, c)-path exists in G if and only if `augmenting-paths`$(G', s, t) = 2$.

First, suppose there are two edge-disjoint paths from b_{out} to t in G'. Since the only incoming vertices to t are a_out and c_out, one path translates to a path that goes to vertex a (a_{in}, a_{out}) and other, a path that goes to vertex c (c_{in}, c_{out}) from b. Vertex-splitting guarantees that these two paths are vertex-disjoint in G; hence, there exist vertex-disjoint paths from b to a and b to c. The path from b to a can be reversed, since G is undirected, and prepended to the path from b to c to get an (a, b, c)-path.

Conversely, suppose G has an (a, b, c)-path. This path can be split in to two vertex-and-edge-disjoint paths; one from b to a and the other, from b to c. It follows that these two paths can be extended to two edge-disjointed paths from b_{out} to t in G'. Since the maximum path-packing in G' clearly has size 2, `augmenting-paths`(G', b_{out}, t) will return 2. ■

□

Exercise 18.3. Suppose you had an (s,t)-flow f. We know that there exists an (s,t)-path packing of the same size as f; here we are interested in algorithms that take f and compute such a path packing. Such a path packing is called a *flow decomposition* of f.

Design and analyze an algorithm that, in $O(m^2)$ time, computes a maximum path packing x of the same size as f, such that:

1. There are at most m distinct paths (with nonzero value) in x.
2. If f is integral, then x is also integral.

Solution. The below algorithm guarentees at most m paths since each iteration removes at least one edge, maintains the same $|f|$ and if f is integral, then $\min|f(e)|$ for an edge e will also be integral.

`Fdecomp(`G, f, s, t`):`

/ given (s,t)-flow f decompose it into a maximum path packing x of same size such that there are at most m distinct paths in x and if f is integral then x is also integral */*

1. $\mathcal{P}, w \leftarrow \emptyset$
2. $G' \leftarrow$ subgraph of G with edges e where $f(e) > 0$
3. While a (s,t)-path in G' exists:
   A. $p \leftarrow$ any (s,t)-path in G' */* using BFS or DFS */*
   B. $w_p \leftarrow \min\{f(e) : e \in p\}$
   C. Add p to $\mathcal{P}$ and w_p to w
   D. For each edge $e \in p$:
      1. $f(e) \leftarrow f(e) - w_p$
      2. If $f(e) = 0$, remove e from G'
4. Return $(\mathcal{P}, w)$

Runtime. There are m edges in the flow, we can find each path using BFS or DFS in $O(m)$ time hence the total runtime is $O(m^2)$ ■

Correctness. The flow decomposition algorithm correctly decomposes an (s,t)-flow f into a maximum path packing x of the same size by iteratively finding paths and extracting flow along them: Valid Path Packing: In each iteration, we extract a path p with weight w_p (the minimum flow along the path) and reduce the flow on each edge in p by exactly w_p. This ensures that for any edge e, the sum of weights of paths containing e never exceeds f(e).

Maximum Size: The algorithm continues until no (s,t)-path remains in G'. At this point, all flow from s to t has been decomposed into paths. The total weight of the path packing equals the original flow value $|f|$.

At most m distinct paths: Each iteration removes at least one edge from G' (the edge with minimum flow w_p). Since there are at most m edges, the algorithm terminates after at most m iterations, resulting in at most m distinct paths.

If f is integral, then all flow values f(e) are integers. The minimum of integers is an integer, so w_p will always be integral. Therefore, all path weights in the decomposition are integral.

The algorithm terminates when there is no (s,t)-path in G', meaning all flow has been decomposed into paths, and the resulting path packing satisfies all the required properties. ■

□

Exercise 18.4. This exercise develops a $O((m^2 + mn\log(n))\log(\lambda))$-time algorithm for maximum (s,t)-flow and builds on ideas from exercise 18.3.

Exercise 18.4.1. Prove the following: Given any (s,t)-flow problem with max flow value $\lambda > 0$, there exists an (s,t)-path where the minimum capacity edge is at least λ/m.

Solution. Let f be a max (s,t)-flow with value $\lambda > 0$, by *flow decomposition* solved in 18.3 we know f can be decomposed into at most m (s,t)-paths say $p_1, \ldots, p_k : k \leq m$ with flow values $f_1, \ldots, f_k$ where

$$\sum_{i=1}^{k} f_i = \lambda$$

For the sake of contradiction, assume that every path p_i has minimum capacity edge is less than λ/m. Then summation over all paths gives

$$\sum_{i=1}^{k} f_i < k(\lambda/m) \leq m(\lambda/m) = \lambda$$

Hence, a contradiction is reached as the total flow value is equal to λ. therefore, there must exist at least one path with minimum capacity of at least λ/m

□

Exercise 18.4.2. Describe an $O(m + n\log(n))$-time algorithm to find the path described above.[4]

[4] $O(m \log n)$ time is a little easier and this running time would still get partial credit. Even if the $O(m+nlog(n))$-running time eludes you, you can assume it as a black box for the next part.

Solution. Below algorithm given any (s,t)-flow problem with max flow $\lambda > 0$ finds (s,t)-path where minimum capacity edge is at least λ/m

`FMinCapPath(`$G = (V, E), s, t, c, \lambda$`):`

/ given any (s,t)-flow problem with max flow $\lambda > 0$ finds (s,t)-path where minimum capacity edge is at least $c_{min} = \lambda/m$ */*

1. $G' \leftarrow G \setminus \{e \in E : c(e) < c_{min}\}$
2. $p \leftarrow$ Dijkstra's algorithm using Fibonacci Heap to find (s,t)-path
3. Return p */* if path does not exist Dijkstra returns null or ∞ */*

Runtime. Removing edges with capacity less than $c_{min} = O(m)$ and running Dijkstra on G' using Fibonacci heap takes $O(m + n\log n)$. Hence, total runtime

$$O(m) + O(m + n\log n) = O(m + n\log n)$$

■

□

Exercise 18.4.3. Based on the two parts above, design and analyze an (s,t)-max flow algorithm that runs in $O(m(m+n\log(n))\log(\lambda))$ time for integer capacities, where λ denotes the value of the maximum flow.[5][6] (The algorithm does not know the true value of λ *a priori*.)

[5]This is polynomial with respect to the bit complexity of the input.
[6]A possibly helpful bit of math: for (small) $\epsilon > 0, \log 1+\epsilon(x) \leq O(\log(x)/\epsilon)$ is a good approximation (which you may want to verify for yourself).

Solution. `MaxFlow(`$G=(V,E), s, t, c$`):`

/ Computes max (s,t)-flow in G for integer capacities */*

1. $f(e) = 0 \forall e \in E, low = 1, high = \sum_{e\in\delta^+(s)} c(e)$
2. $c_f(u,v) \leftarrow c(u,v)$ for all $(u,v) \in E$ */* Initialize residual capacities */*
3. While $high - low > 1$
   A. $mid \leftarrow (low + high)/2$
   B. $p \rightarrow$ `FMinCapPath(`$G=(V,E), s, t, c, mid$`)`
   C. if p exists:
      1. $w_p \leftarrow \min\{c_f(u,v) : (u,v) \in p\}$ */* Bottleneck capacity */*
      2. Augment flow along p
      3. For each $(u,v) \in p$:
         a. $f(u,v) \leftarrow f(u,v) + w_p$
         b. $f(v,u) \leftarrow f(v,u) - w_p$
         c. $c_f(u,v) \leftarrow c_f(u,v) - w_p$
         d. $c_f(v,u) \leftarrow c_f(v,u) + w_p$
      4. $low \leftarrow mid$ */* Update lower bound */*
   D. else $high \leftarrow mid$
4. Return f

Runtime. The algorithm performs a binary search on the maximum flow value λ, which is initially bounded by the total outgoing capacity from s. Since capacities are integers, the binary search takes $O(\log(\lambda))$ iterations.

In each iteration, finding a path with minimum capacity at least λ/m using FMinCapPath in $O(m + n\log n)$ time, augmenting flow along the path and updating residual capacities takes $O(m)$ time.

Each augmenting path found by FMinCapPath has minimum capacity at least λ/m , since the maximum flow value is λ, there can be at most $O(m)$ such augmentations before reaching the maximum flow.

Hence, the total time complexity as required is:

$$O(\log(\lambda)) \cdot O(m + n\log n) \cdot O(m) = O(m(m + n\log n)\log(\lambda))$$

■

Correctness. The algorithm uses binary search to find the maximum flow value λ. In each iteration, it tries to find an augmenting path where all edges have capacity at least mid/m.

If such a path exists, we know that the maximum flow value is at least mid, so we update the lower bound. We then augment flow along this path, which increases the total flow while maintaining flow conservation.

If no such path exists, we can infer from part 18.4.1 that the maximum flow value must be less than *mid*, so we update the upper bound.

The binary search continues until the gap between upper and lower bounds is at most 1, at which point we have found the maximum flow value being an integer. ■

□