

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 guarantees 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 $p_1, \dots, p_k : k \leq m$ with flow values f_1, \dots, 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 $O(m + n \log(n))$ -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.⁵⁶ (The algorithm does not know the true value of λ *a priori*.)

⁵This is polynomial with respect to the bit complexity of the input.

⁶A 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.  $p \rightarrow \text{FMinCapPath}(G = (V, E), s, t, c, mid)$ 
    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. ■

□