The Budget-Constrained Maximum Flow Problem

In this chapter we consider the following problem which is called the constrained maximum flow problem (CMFP) [AO95]: We are given a budget $B$ and we seek a flow of cost at most $B$ such that the amount of flow from a designated source to a designated sink is maximized.

An interesting application of CMFP is for the case when the capacities of the arcs in the network are not fixed but can be increased at certain costs per unit (see e.g. [Ful59, KNR+98, SK98]). One is allowed to spend a certain budget for raising capacities and wishes to maximize the throughput. This network improvement problem can be formulated as an instance of CMFP where the cost functions on the arcs are piecewise linear. Observing that the cost functions are also convex and using techniques described in [AMO93] the problem can be reformulated as an instance of CMFP with linear costs.

The algorithms presented here are based on Megiddo’s parametric search technique [Meg79, Meg83]. In Section 9.3 we show how to simulate the minimum mean cycle canceling algorithm. This algorithm illustrates the main ingredients in the design of our algorithms. We will also use it to indicate how techniques from [Meg83] can be used to decrease the running time. The speedup is achieved by simulating appropriate parallel algorithms sequentially.

9.1 Problem Definition and Preliminaries

As in the minimum cost flow problems in Chapter 4, we are given a directed graph $G = (V, R, \alpha, \omega)$ with capacities $c: R \rightarrow \mathbb{R}_+$ and costs $k: R \rightarrow \mathbb{R}_+$ on the arcs. A variant of the classical minimum cost flow problem consists of finding a feasible $s$-$t$-flow with minimum cost subject to the constraint that the value of the flow equals a prescribed amount. The problem CMFP which is the topic of this chapter is in some sense the symmetric problem obtained from the minimum cost flow problem by exchanging the two objectives, namely cost and value of the flow: Given a directed graph $G$ with nonnegative capacities and nonnegative costs for the arcs find a flow $f$ from the source node $s$ to the sink node $t$ with maximum value $\text{val}(f)$ subject to the constraint that the cost of the flow is at most the given budget $B$.

Definition 9.1 (Constrained Maximum Flow Problem (CMFP)) The con-
The constrained maximum flow problem (CMFP) consists of solving the linear program

\[ \varphi^* := \text{maximize } \varphi \]

subject to

\[ f(\delta^-(v)) - f(\delta^+(v)) = \begin{cases} \varphi & \text{if } v = t \\ 0 & \text{for all } v \in V \setminus \{s, t\} \\ -\varphi & \text{if } v = s \end{cases} \]

\[ \sum_{r \in R} k(r)f(r) \leq B \]

\[ 0 \leq f(r) \leq c(r) \quad \text{for all } r \in R. \]

In the sequel we denote by \( \varphi^* := \max \{\text{val}(f) : f \text{ is a flow of cost at most } B\} \) the optimum flow value achievable subject to the budget constraint on the total flow cost. The following result proved in the homework exercises turns out to be useful in our studies:

**Theorem 9.2** Let \( f^* \) be an optimal solution of the minimum cost flow problem, when the supply of the source node is constrained to be equal to \( \varphi^* \). Then, \( f^* \) is also an optimal solution of the constrained maximum flow problem with solution value \( \varphi^* \) if \( k(f) = B \).

Theorem 9.2 tells us that in order to solve CMFP it suffices to solve the corresponding instance of the minimum cost flow problem where the supply of the source node equals \( \varphi^* \). As \( \varphi^* \) is not known in advance this easy observation seems to be of no help. We will show that using Megiddo’s parametric search method [Meg79] we can simulate a minimum cost flow algorithm which computes a minimum cost flow of value \( \varphi^* \), where \( \varphi^* \) is not known in advance.

We use \( T_{\text{MCF}}(n, m) \) to denote the time needed to compute a minimum cost flow in a graph with \( n \) nodes and \( m \) arcs. Currently, the best strongly polynomial-time algorithm for solving the minimum cost flow problem with linear flow costs is the enhanced capacity scaling algorithm due to Orlin which runs in time \( O(m \log n \cdot S(n, m)) \) (see [AMO93]), where \( S(n, m) \) denotes the time needed to solve a shortest path problem in a graph with \( n \) nodes and \( m \) arcs and nonnegative arc lengths.

Let us denote by \( \text{val}_{\text{max}} \) the maximum flow value in the graph \( G \) (ignoring the budget constraint). For \( 0 \leq \varphi \leq \text{val}_{\text{max}} \) we define the flow cost function

\[ C(\varphi) := \min \{k(f) : f \text{ is a feasible flow in } G \text{ with } \text{val}(f) = \varphi \}. \quad (9.1) \]

Using this definition we can rephrase the problem CMFP as the problem of finding

\[ \varphi^* = \max \{0 \leq \varphi \leq \text{val}_{\text{max}} : C(\varphi) \leq B \}. \]

**Theorem 9.3** The function \( C \) defined in (9.1) is nonnegative, piecewise-linear, and convex.

**Proof:** Homework. \( \square \)

Notice that given a candidate flow value \( \varphi \) we can decide by one minimum cost flow computation whether \( \varphi^* \leq \varphi \) or not: We simply compute a minimum cost flow of the candidate value \( \varphi \) and then check whether its cost exceeds the given budget \( B \). This allows us to solve the CMFP in polynomial time by a binary search over the
interval $[0, nC]$ as was done in the homework. However, although the resulting algorithm is polynomial, it is not strongly polynomial.

In our algorithm we will use as a building block a strongly polynomial time algorithm for the minimum cost flow problem. We will use the minimum mean-cycle algorithm which itself is based on Klein’s algorithm to illustrate the techniques. Then, we will replace this algorithm by the enhanced capacity scaling algorithm of Orlin which runs in time $O(m \log n \cdot S(n, m))$ (see [AMO93]), where $S(n, m)$ denotes the time needed to solve a shortest path problem in a graph with $n$ nodes and $m$ arcs and nonnegative arc lengths and is currently still the fastest strongly polynomial time algorithm for the minimum cost flow problem. We could use also the scale-and-shrink algorithm from Section 4.5, however, here the case is much more involved.

Klein’s algorithm is rather simple. We start with any feasible $b$-flow (this can be done by a maximum flow computation) and then, as long as there exists a cycle of negative length in the residual network, and as much flow as possible around this cycle.

**Theorem 9.4** If all data, i.e., $b$, capacities $c$ and costs $k$ are integral, Klein’s Algorithm terminates after at most $O(mCK)$ iterations with an optimum $b$-flow which is also integral. Here $C = \max \{ c(r) : r \in R \}$ and $K := \max \{ k(r) : r \in R \}$.

**Proof:** The cost of the initial feasible flow $f$ is at most $\sum_{r \in R} k(r) f(r) \leq \sum_{r \in R} KC = mKC$. In each iteration this cost decreases strictly by an integral amount. Since the total cost of a flow is lower bounded by 0 (recall that we have assume the costs to be nonnegative), the algorithm must terminate after $O(mCK)$ iterations. By the cycle optimality condition (Theorem 4.5), the flow found must be optimal.

Klein’s algorithm is not a polynomial algorithm. However, it can be turned into one by carefully choosing the negative cycle. For a cycle $C$ in $G_f$ denote by

$$\frac{k(C)}{|C|} = \frac{\sum_{r \in C} k(r)}{|C|}$$

the ratio cost of the cycle. The minimum mean cycle algorithm selects in any iteration a cycle of minimum ratio cost.

**Theorem 9.5** The minimum mean cycle algorithm uses $O(nm^2 \log n)$ iterations and runs in time $O(n^2m^3 \log n)$.

**Proof:** See e.g. [AMO93].

### 9.2 The Minimum Ratio Cycle Problem

In this section we illustrate the idea of parametric search for a somewhat simpler problem. Suppose that we are given a directed graph $G = (V, R)$ with $a: R \to \mathbb{R}$ and $b: R \to \mathbb{R}_{>0}$ on the arcs. We wish to solve the following problem:

$$\text{MRCP} \quad \min \frac{\sum_{r \in C} a(r)}{\sum_{r \in C} b(r)}$$

$C \in X := \{ K : K \text{ is a cycle in } G \}$.

This problem generalizes the minimum mean cycle problem for which we have $a = k$ and $b \equiv 1$. Recall that by the Bellman-Ford algorithm (see Algorithm 9.2) we can
find a negative length cycle or certify that none exists in $O(nm)$ time. Algorithm 9.3 shows how we can find the negative length cycle by the information provided by the Bellmann-Ford algorithm. If no negative length cycle exists, then by the algorithm of Floyd and (see Algorithm 9.1) we can find a shortest cycle in time $O(n^3)$.

**Algorithm 9.1** Algorithm of Floyd and Warshall

**FLOYD-WARSHALL**($G, c$)

- **Input:** A directed graph $G = (V, R)$, arc weights $c: R \rightarrow \mathbb{R}$
- **Output:** For all $u, v \in V$ the distance $D_n[u, v] = \text{dist}_c(u, v)$

1. for all $v_i, v_j \in V$ do
2. $D_0[v_i, v_j] := +\infty$
3. end for
4. for all $(v_i, v_j) \in R$ do
5. $D_0[v_i, v_j] := c(v_i, v_j)$
6. end for
7. for $k = 0, \ldots, n - 1$ do
8. for $i = 1, \ldots, n$ do
9. for $j = 1, \ldots, n$ do
10. $D_{k+1}[v_i, v_j] := \min \{ D_k[v_i, v_j], D_k[v_i, v_{k+1}] + D_k[v_{k+1}, v_j] \}$
11. end for
12. end for
13. end for
14. return $D_n[]$

**Algorithm 9.2** Algorithm of Bellman and Ford

**BELLMAN-FORD**($G, c, s$)

- **Input:** Directed graph $G = (V, R)$, arc weights $c: R \rightarrow \mathbb{R}$, a vertex $s \in V$
- **Output:** For all $v \in V$ the distance $d[v] = \text{dist}_c(s, v)$ and a shortest path tree $G_\pi$

1. **INIT**($G, s$)
2. for $k := 1, \ldots, n - 1$ do
3. { *Start of phase $k$* }
4. for all $(u, v) \in R$ do
5. **Test**$(u, v)$
6. end for
7. { *End of phase $k$* }
8. end for
9. return $d[]$ and $G_\pi$

**Test**$(u, v)$

1. if $d[v] > d[u] + c(u, v)$ then
2. $d[v] := d[u] + c(u, v)$
3. $\pi[v] := u$
4. end if

Let us denote by $\lambda^*$ the optimum value of the MRCP. Suppose that $t \in \mathbb{R}$ is an estimate for $\lambda^*$. We imagine that we run the combination of the Bellman-Ford algorithm and the Floyd-Warshall-algorithm with arc weights $c = a - tb$. There are three cases:

**Case 1:** There is no negative length cycle and the shortest cycle $C$ has length $c(C) = 0$. 
9.2 The Minimum Ratio Cycle Problem

Algorithm 9.3 Algorithm for testing for a negative length cycle.

\textsc{Test-Negative-Cycle}(G, c, d)

\textbf{Input:} Directed graph \( G = (V, R) \), weights \( c: R \rightarrow \mathbb{R} \), distance labels \( d \) from the Bellman-Ford-Algorithm

1. \textbf{for all} \((u, v) \in R\) \textbf{do}
2. \hspace{1em} \textbf{if} \( d[v] > d[u] + c(u, v) \) \textbf{then}
3. \hspace{2em} \textbf{return} “yes”
4. \hspace{1em} \textbf{end if}
5. \hspace{1em} \textbf{end for}
6. \hspace{1em} \textbf{return} “no”

In this case we have

\[ 0 = c(C) = a(C) - tb(C) \iff \frac{a(C)}{b(C)} = t \]

and for all other cycles \( C' \) it holds that

\[ 0 \leq c(C') = a(C') - tb(C') \iff \frac{a(C')}{b(C')} \geq t. \]

Hence, \( C \) is a cycle with minimum ratio cost and \( t = \lambda^* \).

\textbf{Case 2:} The shortest cycle has length strictly larger than \( 0 \) In this case \( a(C) - tb(C) > 0 \) for all cycles and by the calculation above we can conclude that \( a(C)/b(C) > t \) for all cycles. Consequently \( t < \lambda^* \).

\textbf{Case 3:} There is a cycle \( C \) of negative length.

From \( c(C) = a(C) - tb(C) < 0 \) we get as above \( a(C)/b(C) < t \), hence it must be the case that \( \lambda^* < t \).

Hence, we can first run the Bellman-Ford algorithm to test in time \( O(mn) \) whether there is a negative length cycle with respect to the weights \( a - tb \). If this is the case, we are in Case 3 and we know that \( \lambda^* < t \). If there is no negative length cycle, we use the Floyd-Warshall algorithmus to find a shortest one in time \( O(n^3) \). Inspecting this cycle gives us the information whether \( \lambda^* > t \) or \( \lambda^* = t \).

We could now run a binary search for \( \lambda^* \), where for each decision in the search we have to run the combined shortest path algorithm. The issue is that we do not really know when to stop the search. In any case, there exists a much more elegant way: parametric search.

Suppose that we run our combined algorithm for \( \lambda^* \). What will happen? As seen before, we must end up in Case 1, and hence, we do not even have to run the Bellman-Ford part, since there is no negative length cycle. The idea is now the following, we simulate the Floyd-Warshall algorithmus for the case that \( t = \lambda^* \) without knowing \( \lambda^* \) a priori. If our simulated cycle algorithm makes all decisions exactly as in the case \( t = \lambda^* \), then we must end up with the optimum ratio cycle at the end!

We use \( \lambda^* \) as a symbolic constant in the Floyd-Warshall algorithmus with weights \( c = a - tb \). We imagine these weights as linear functions \( t \mapsto a(r) - tb(r) \), which we want to evaluate at \( t = \lambda^* \).

Again: We “simulate” the Floyd-Warshall algorithm for the weights \( a - \lambda^*b \), where \( \lambda^* \) is a symbolic constant. Since the algorithm only adds the linear functions and compares sums of them, it follows by induction that all values \( D_i[u, v] \) are always of the
form $D_k[u, v] = \alpha(u, v) + t\beta(u, v)$ with known constants $\alpha(u, v)$ and $\beta(u, v)$. We run the algorithm until the first comparison $D_{k+1}[u, v] := \min \{ D_k[u, v], D_k[u, z] + D_k[z, v] \}$, which means that we ask whether

$$D_k[u, v] > D_k[u, z] + D_k[z, v].$$

We have

$$D_k[u, v] = \alpha(u, v) + t\beta(u, v)$$
$$D_k[u, z] = \alpha(u, z) + t\beta(u, z)$$
$$D_k[z, v] = \alpha(z, v) + t\beta(z, v)$$

with known constants $\alpha$ and $\beta$. The corresponding comparison is thus

$$\alpha(u, v) + t\beta(u, v) > \alpha(u, z) + t\beta(u, z) + \alpha(z, v) + t\beta(z, v) \quad (9.3a)$$

$$\iff \alpha(u, v) - \alpha(u, z) > \alpha(z, v) + t(\beta(u, z) - \beta(u, v) + \beta(z, v)) \quad (9.3b)$$

$$\iff A > tB \quad (9.3c)$$

and we wish to evaluate the comparison for $t = \lambda^*$. If $B = 0$, then the result of the comparison is independent from $\lambda^*$ and we can give the answer directly. After that we can continue with our simulation in the same way as if $\lambda^*$ had been known. If $B > 0$, then the question in (9.3) reduces to whether $\lambda^* < A/B$, and if $B < 0$, we ask whether $\lambda^* > A/B$. The right hand side of the comparison contains only known constants, hence, we wish to compare $\lambda^*$ with a fixed value $w := A/B$. But this is exactly the situation which we handled in Cases 1 to 3 above. We stop the simulation and compute a shortest cycle for the weights $a - wb$. The answer lets us resolve the comparison in (9.3) and we can continue with our simulation. Again, the same happens in the simulation as if we had known $\lambda^*$.

The running time can be estimated as follows: For each of the $O(n^3)$ steps of the Floyd-Warshall algorithm we must resolve at most one comparison by running the shortest cycle algorithm. Hence, resolving a comparison needs $O(n^3)$ time and we obtain an overall running time of $O(n^6)$.

**Observation 9.6** A minimum ratio cycle can be computed in time $O(n^6)$.

The above running time can be reduced dramatically by using techniques similar to those that we will use for our flow problem.

### 9.3 A Strongly Polynomial Algorithm

In this section we present a strongly polynomial algorithm for solving CMFP. Our algorithm is based on Megiddo’s parametric search method [Meg79, Meg83]. The main idea is to “simulate” an appropriate minimum cost flow algorithm (we will use the minimum mean cycle algorithm) which computes a minimum cost flow of value $\varphi^*$, where $\varphi^*$ is not known in advance. During the simulation we keep $\varphi^*$ as a (semi-)symbolic value. The peculiarity of $\varphi^*$ is that we may compare it to other values, but we can perform arithmetic operations with it only symbolically. Therefore we will have to use a symbolic representation for the current flow on an arc $r$. This representation has the form

$$f(r) = g(r) + h(r) \cdot \varphi^*, \quad (9.4)$$
where \( g(r) \) and \( h(r) \) are known values. We will call a flow \( f \) where each flow value \( f(r) \) is of the form (9.4) a linear parametric flow, since the flow values have the formal appearance of linear functions of the unknown \( \varphi^* \).

In the simulation we run the minimum cost flow algorithm with the only difference that we add and subtract values of the form (9.4) instead of performing standard arithmetic operations. Each time the simulated algorithm makes a comparison, we will resolve this comparison by computing minimum cost flows (for known values) and then take the branch the algorithm would have taken if it had known \( \varphi^* \).

Recall that given a candidate flow value \( \varphi \) we can decide by one minimum cost flow computation whether \( \varphi^* \leq \varphi \) or not: We simply compute a minimum cost flow of the candidate value \( \varphi \) and then check whether its cost exceeds the given budget \( B \).

An important consequence is the following: Even though we do not know \( \varphi^* \), we can still compare two values of the form (9.4) correctly.

**Lemma 9.7** The comparison \( f(r) < f'(r') \), where the values \( f(r) \) and \( f(r') \) are of the form (9.4) can be resolved by at most one minimum cost flow computation for the “critical value” \( v_{r,r'} \), defined by

\[
v_{r,r'} := \frac{g(r) - g(r')}{h(r') - h(r)}.
\]

(If \( h(r) = h(r') \), then the answer does not depend on \( \varphi^* \) and we can simply compare \( g(r) \) and \( g(r') \) without a minimum cost flow computation.) Moreover, the answer to the comparison \( f(r) < f'(r') \) is uniform for each of the cases \( \varphi^* \in (-\infty, v_{r,r}] \) and \( \varphi^* \in [v_{r,r'}, +\infty) \).

**Proof:** We have

\[
f(r) < f'(r') \Leftrightarrow g(r) + h(r) \cdot \varphi^* < g(r') + h(r') \cdot \varphi^* \\
\Leftrightarrow (h(r') - h(r))\varphi^* > g(r) - g(r').
\]

If \( h(r') = h(r) \), then the answer depends only on the difference \( g(r) - g(r') \) which we can evaluate even without performing a minimum cost flow computation. Assume that \( h(a') > h(a) \) (the other case is similar). Then we have that \( f(a) < f(a') \) if and only if \( \varphi^* > \frac{g(a) - g(a')}{h(a') - h(a)} = v_{a,a'} \). Now, computing a minimum cost flow for the flow value \( v_{a,a'} \) and testing whether the cost of the flow is at most the budget \( B \) yields the appropriate answer.

As mentioned before, we will simulate the minimum mean cycle-canceling algorithm. This algorithm works by first computing an arbitrary flow of the required value \( \varphi^* \) and then iteratively sending the maximum possible flow along a cycle in the residual network which has the smallest ratio-cost. The algorithm stops when no negative cycle in the residual network exists. We will show now that we can use the minimum mean cycle-canceling algorithm to solve CMFP in time \( O(nm^2 \log^2 n \log \log n \cdot T_{MCF}(n, m)) \).

As stated above, the minimum mean cycle-canceling algorithm will be simulated while keeping \( \varphi^* \) as a symbolic value. Notice that if all flow values are of the form stated in Equation (9.4) if we augment the maximum possible flow along a cycle, then the resulting flow values on the arcs will all still be of the form (9.4). Since all flow values are zero initially, it follows that during the whole simulation of the minimum mean cycle-canceling algorithm all flow values remain of the form (9.4).

During the simulation we also maintain an interval \([a, b]\) which is known to contain the optimum flow value \( \varphi^* \). The interval is initially set to \([0, mC]\). Notice that
the optimum flow value $\varphi^*$ is clearly contained in this interval. Then, during the simulation, we repeatedly have to check whether $\varphi^* \leq \varphi$ for some value $\varphi$. If $\varphi$ is outside the interval $[a, b]$ we directly derive the answer. Otherwise we compute one minimum cost flow to determine the answer to $\varphi^* \leq \varphi$ and update the interval to $[\max(a, \varphi), \min(b, \varphi)]$. Observe that this shrinking process has the following important property: If $[a, b]$ is the iteratively maintained interval, then all comparisons resolved so far are uniform over the interval $[a, b]$; that is, if we substitute any value from $[a, b]$ for $\varphi^*$, this will not change the outcome of any comparison made so far.

9.3.1 Finding an initial flow

To find an initial feasible flow of (unknown) value $\varphi^*$ we simulate an appropriate strongly polynomial maximum flow algorithm such as the shortest augmenting path algorithm (see e.g. [AMO93, Section 7.4]). This algorithm, also known as the Edmonds-Karp algorithm, always finds an augmenting path in the residual network $G_f$ with the least number of arcs and then augments the maximum possible flow value along this path.

**Theorem 9.8** The shortest augmenting path algorithm performs $O(mn)$ iterations and can be implemented to run in time $O(n^2m)$.

**Proof:** See e.g. [AMO93] □

The Edmonds-Karp algorithm can be used to find an initial feasible flow of value $\varphi^*$ in the following way: The initial flow is identically zero. All flow values on the arcs will remain to be known values (instead of being of the more general form (9.4)) except after the last augmentation. Given a current flow of (known) value $\varphi$, we find a shortest augmenting path $P$ in the residual network. Let $r_P$ be the residual capacity of this path. We now check whether $\varphi^* \geq \varphi + r_P$ by one minimum cost flow computation. If $\varphi^* \geq \varphi + r_P$ we augment the maximum possible flow value $r_P$ along $P$ and continue. If, however, $\varphi^* < \varphi + r_P$ we have to take appropriate steps. Notice that this situation can only occur in the last augmentation, since this augmentation will achieve the desired initial flow of value $\varphi^*$.

If $\varphi^* < \varphi + r_P$ we augment $\varphi^* - \varphi$ units of flow along $P$ (this will result in a flow of value $\varphi^*$). The augmentation is done symbolically: If the $+r$ is a forward arc on the path, and the flow on $r$ is $f(r)$ before the augmentation, it will be $f(r) - \varphi + \varphi^*$ after the augmentation (the case of a backward arc is similar). Notice that the flow value $f(r)$ on the arc $r$ before the augmentation was a known value and that $\varphi$ is also a determined value. Thus, all flow values in the flow at termination will be of the form (9.4). The time for computing the initial flow is thus $O(n^2m + mn \cdot T_{MCF}(n, m))$.

**Lemma 9.9** An initial feasible linear parametric flow of value $\varphi^*$ can be computed in time $O(n^2m + mn \cdot T_{MCF}(n, m))$. □

9.3.2 Simulation of the minimum cost flow algorithm

We will now show how to simulate the main iteration of the minimum mean cycle-canceling algorithm. Suppose that we are given a current feasible linear parametric flow $f$ of value $\varphi^*$.

The minimum mean cycle-canceling algorithm would now determine a minimum mean cost cycle in the residual network $G_f$. Notice that in order to simulate this
step it suffices to determine the topology of the residual network. After this step we can identify a minimum mean cost cycle in the residual network by any standard algorithm, since this cycle depends only on the flow costs \( k(\sigma r) \) of the arcs \( \sigma r \) in the residual network. A minimum mean cost cycle can be found in time \( \mathcal{O}(nm) \) (see e.g. [AMO93]).

The topology of the residual network could be computed by checking for each arc \( r \in R \) separately whether it is contained in \( G_f \) as a forward and/or backward arc. To check this we could resolve each of the comparisons \( f(r) < c(r) \) and \( f(r) > 0 \) by one minimum cost flow computation. This would result in \( \mathcal{O}(m) \) minimum cost flow computations. However, we can accomplish this task more efficiently by using only \( \mathcal{O}(\log m) \) minimum cost flow computations. This is a consequence of the following more general lemma:

**Lemma 9.10** We can answer \( \mathcal{O}(m^k) \) simultaneous (or independent) comparisons \( f(r) < f(r') \), where each flow value is of the form (9.4), in time \( \mathcal{O}(m^k + k \log m \cdot T_{MCF}(n,m)) \).

**Proof:** By Lemma 9.7 each comparison can be resolved by one minimum cost flow computation. Instead of answering each comparison separately, we first only compute the critical value defined in Equation (9.5) for each comparison.

We then use a binary search to locate \( \varphi^* \) between those \( \mathcal{O}(m^k) \) critical values, i.e., we find a partition \( P_1 \cup P_2 \) of the set of critical values such that \( \varphi^* \) satisfies \( p_1 \leq \varphi^* \) for all \( p_1 \in P_1 \) and \( \varphi^* < p_2 \) for all \( p_2 \in P_2 \). This is done in the following way: Start with \( L := -\infty \) and \( H := +\infty \). Then compute the median \( M \) of the critical values in \( \mathcal{O}(m^k) \) time. We decide whether \( \varphi^* \leq M \) by one minimum cost flow computation and adjust either \( L \) or \( H \) to \( M \) according to the result. All critical values that are no longer contained in the range between \( L \) and \( H \) are removed. Using a linear time median finding algorithm (see e.g. [CLR90]) all these operations can be carried out in time \( \mathcal{O}(m^k) \). We now continue for the remaining half of the critical values. Since the size of the set of critical values is decreased by a factor of two in each iteration, it follows that the total time effort for the binary search is \( \mathcal{O}(\log m^k) = \mathcal{O}(k \log m m) \) minimum cost flow computations plus an overhead of \( \mathcal{O}(m^k) \) elementary operations.

Notice that a similar result as in Lemma 9.10 could be achieved by first sorting the \( \mathcal{O}(m^k) \) critical values and then performing a standard binary search among the critical values. This would result in a time of \( \mathcal{O}(km^k \log m + k \log m \cdot T_{MCF}(n,m)) \).

**Corollary 9.11** For a linear parametric flow \( f \) the residual network \( G_f \) can be determined in time \( \mathcal{O}(m + \log m \cdot T_{MCF}(n,m)) \).

We have shown that we can find a minimum mean cycle in the residual network \( G_f \) of a linear parametric flow \( f \). We now have to handle the canceling of the negative cycle just found. In order to do this, we have to find the bottleneck value of the cycle, i.e., the minimum residual capacity on the cycle. To determine the bottleneck we could run a standard minimum finding algorithm on the at most \( m \) residual capacities. Each comparison of the algorithm could be resolved by one minimum cost flow computation, which would give us a total of \( \mathcal{O}(m) \) minimum cost flow computations. However, using Megiddo’s technique [Meg83] we can speed up the algorithm substantially.

**Lemma 9.12** We can sort \( \mathcal{O}(m) \) values of the form (9.4) in time \( \mathcal{O}(\log^2 m \cdot (m + T_{MCF}(n,m))) \).
The Budget-Constrained Maximum Flow Problem

**Proof:** The idea from [Meg83] is to use an adaptation of a sequentialized parallel sorting algorithm such as Cole’s scheme [Col88]. Cole’s algorithm uses \( m \) processors to sort an array of \( m \) elements in parallel time \( O(\log m) \). The algorithm is simulated serially, employing one “processor” at a time, according to some fixed permutation, letting each perform one step in each cycle.

When two values \( f(a) \) and \( f(a') \) have to be compared, we compute the critical value \( v_{a,a'} \) but we do not resolve the comparison at this point. The crucial observation is that these critical values can be computed independently, meaning that each of the “processors” does not need any knowledge about the critical points computed by the other ones.

After the first of the \( O(\log m) \) stages, we are given at most \( m \) critical values and we resolve all comparisons of the first round in time \( O(m + \log m \cdot T_{MCF}(n,m)) \) by the method given in Lemma 9.10.

The above process is repeated \( O(\log m) \) times, once for each parallel step of the parallel sorting machine. Since in each of the \( O(\log m) \) rounds we answer all comparisons of the parallel sorting scheme, upon termination we have found the desired sorting of the values.

Lemma 9.12 gives us a method of finding the bottleneck on the cycle in time \( O(m + \log^2 m \cdot T_{MCF}(n,m)) \). Using a sequential simulation of a parallel minimum finding algorithm instead of a sorting algorithm we can do even better. The minimum of \( m \) values can be found in \( O(\log \log m) \) parallel steps on \( O(m) \) processors [Val75]. Simulating this algorithm the way we did this for the sorting algorithm in Lemma 9.12 enables us to find the bottleneck on the minimum mean cycle in time \( O(\log \log m (m + \log m \cdot T_{MCF}(n,m))) \).

**Lemma 9.13** We can find the minimum of \( O(m) \) values of the form (2.1) in time \( O(\log \log m (m + \log m \cdot T_{MCF}(n,m))) \).

As noted above, augmenting the flow along the cycle will not change the general form of the flow values on the arcs. The result will still be a linear parametric flow. Thus, we can continue our simulation of the minimum mean cycle-canceling algorithm until it stops because no negative cycle remains.

### 9.3.3 Correctness of the simulation

At termination of the simulated minimum mean cycle-canceling algorithm we have a linear parametric flow \( f \) together with an interval \( [a, b] \) that contains \( \varphi^* \) and which has been updated during the run of the algorithm.

Observe that all decisions taken by the minimum mean cycle-canceling algorithm during the simulation are exactly the same as those it would have taken if it had known \( \varphi^* \). Moreover, if we look at our final interval \([a, b]\) at the end of the simulation it follows from the uniformity of comparisons made over \([a, b]\) that for all flow values in \([a, b]\) all comparisons made by the minimum mean cycle-canceling algorithm would be resolved in the same way as in our simulation.

Thus, for each \( \varphi \in [a, b] \) we obtain a minimum cost flow if we substitute the known value \( \varphi \) for the unknown \( \varphi^* \) in the linear parametric flow \( f \). Hence for all \( \varphi \in [a, b] \), the cost of a minimum cost flow of value \( \varphi \) is given by the linear function \( y(\varphi) := \sum_{r \in R} c(r) f(r) = \sum_{r \in R} c(r)(g(f) + \varphi \cdot h(r)) \). This enables us to find the optimum flow value \( \varphi^* \) by solving the linear equation \( y(\varphi^*) = B \). Notice that these observations imply in particular that the flow cost function \( C \) defined in (9.1) is linear over the interval \([a, b]\).
Using the simulation described above we obtain a solution of \( C_{mfp} \) by performing \( O(nm^2 \log^2 n \log \log n) \) minimum cost flow computations, whose running time clearly dominates the overall running time of the computation, which shows the following result:

**Theorem 9.14** The problem \( C_{mfp} \) can be solved in time \( O(nm^2 \log^2 n \log \log n \cdot T_{MCF}(n, m)) \).

### 9.4 A Faster Strongly Polynomial Algorithm

To obtain a better running time than that stated in Theorem 9.14 we simulate a faster strongly polynomial minimum cost flow algorithm than the minimum mean cycle-canceling algorithm. We choose the scale-and-shrink algorithm from Section 4.5. Since the techniques are essentially the same as in Section 9.3 we only sketch the main steps in the simulation. The overall result is then the following:

**Theorem 9.15** The problem \( C_{mfp} \) can be solved in time \( O(m \log n \cdot (\log \log n \log n + \log m) \cdot T_{MCF}(n, m)) \), where \( T_{MCF}(n, m) \in O(m^2 \log n + mn \log^2 n) \) denotes the time to compute a minimum cost flow in a directed network with \( n \) nodes and \( m \) arcs.

The enhanced capacity scaling algorithm performs \( O(m \log n) \) so called \( \Delta \)-scaling phases. The scaling parameter \( \Delta \) is initialized with the maximum imbalance of a node. In each \( \Delta \)-scaling phase the algorithm augments flow along shortest paths in the residual graph \( G_f \). Except for the augmentations the algorithm performs the following basic operations in each phase:

1. It finds a node with maximum imbalance.
2. It finds those nodes with sufficiently large flow excesses and deficiencies, respectively.
3. It determines for each arc whether this arc is an “abundant” arc. An arc \( r \) is called “abundant” if \( f(r) \geq 8n\Delta \), where \( \Delta \) is the scaling parameter.
4. It computes the components of the subgraph consisting of the vertex set \( V \) and the abundant arcs.

At the end of a \( \Delta \)-scaling phase the scaling parameter \( \Delta \) is usually updated to \( \Delta/2 \). The possibility to scale down \( \Delta \) by more than 2 makes the number of scaling phases independent of \( U \). The total number of augmentations in all \( \Delta \)-scaling phases is in \( O(m \log n) \). We refer to [AMO93, Section 10.7] for a complete description of the algorithm.

We simulate the enhanced capacity scaling algorithm much in the same way that we simulated the minimum mean cycle-canceling algorithm in Section 9.3. We keep the optimal flow value \( \phi^* \) as a symbolic constant and maintain an interval \([a, b]\) which is known to contain the optimum flow value \( \phi^* \). The interval \([a, b]\) is shrunk each time comparisons are resolved with the help of a minimum cost flow computation. In performing our simulation we exploit “parallelism” (as we did in Section 9.3) in order to reduce the number of minimum cost flow computations needed to resolve the comparisons made during the simulation.

By Lemma 9.13 we can simulate Step (i), i.e., finding a node with maximum imbalance, by \( O(\log \log n \log n) \) minimum cost flow computations. Step (ii) can be achieved by \( O(\log n) \) minimum cost flow computations, as the excess at at
a node also has the form (2.1). Determining abundant arcs (Step (iii)) takes time \( O(\log m \cdot T_{MCF}(n, m)) \). After this step, the topology of the “abundant subgraph” is known and Step (iv) needs no further minimum cost flow computations. Thus, except for the augmentations, our simulation of the enhanced capacity scaling algorithm needs time \( O(m \log n \cdot (\log \log n \log n + \log m) \cdot T_{MCF}(n, m)) \).

We now consider the augmentations. As mentioned before, the enhanced capacity scaling algorithm solves \( \mathcal{O}(m \log n) \) shortest path problems during its run. In each of these problems a shortest path tree in the residual network (with reduced costs on the arcs which imply nonnegative arc lengths) is computed. Observe that once we know the topology of the residual network (which as shown in Section 9.3 can be determined with \( O(\log m) \) minimum cost flow computations) we can use a standard shortest path algorithm. This is basically the same situation as in Section 9.3 where we had to determine a minimum mean cycle in the residual network. Thus, the complete simulation of the enhanced capacity scaling algorithm can be done in time \( O(m \log n \cdot (\log \log n \log n + \log m) \cdot T_{MCF}(n, m)) \), which is the time bound claimed in Theorem 9.15.

The correctness of the resulting algorithm follows by the same arguments as in Section 9.3.3: All decisions taken by the enhanced capacity scaling algorithm during the simulation are exactly the same as those it would have taken if it had known the optimal flow value \( \phi^* \). Thus after termination we can again find the optimum flow value \( \phi^* \) from our final linear parametric flow by solving a linear equation as in Section 9.3.3.