Combinatorial Optimization

lecture notes, winter term 2023/24
Prof. Dr. Yann Disser
February 8, 2024

Contents

1 Introduction

This lecture focuses on advanced combinatorial algorithms. We will revisit most topics of the lecture *Algorithmic Discrete Mathematics* and provide a deeper and more complete picture of the state of the art in each of them. In particular, we will introduce more sophisticated algorithmic techniques that will allow us to design algorithms that come close to achieving the best known running times for various problems. Some of these techniques are based on structural insights in the theory of mathematical optimization that were discussed in the lectures *Introduction to Optimization* and *Discrete Optimization*. Specifically, we will first consider shortest path problems (Chapter 2), then maximum flow problems (Chapters 3 and 4), and finally matching problems (Chapter 5). This advanced lecture relies on fundamental concepts and notations of computational complexity theory and graph theory. We refrain from reintroducing these notions here and refer to the lecture notes of the course *Algorithmic Discrete Mathematics* instead.

2 Shortest Paths

In this chapter, we consider the problem of finding shortest paths in a given directed graph $G = (V, E)$ with arc weights $c\colon E \to \mathbb{R}$. For convenience, we consider an arbitrary but fixed directed graph G with weights c throughout this section. As usual, we use the notations $c(u, v) := c((u, v))$ for $(u, v) \in E$, $c(u, u) := 0$ for $u \in V$, and $c(u, v) := \infty$ for $(u, v) \notin E$ and $u \neq v$. We recall the following fundamental definition of distances in G with respect to c.

Definition 2.1. We let $\mathcal{W}_{uv}$ denote the set of all walks from u to v in G and define[a]

$$d_c^{(k)}(u, v) := \min\{c(W) \mid W = (v_0, e_1, \dots, v_{k'}) \in \mathcal{W}_{uv} \text{ with } k' \leq k\},$$
$$d_{c,G}(u, v) := d_c(u, v) := \lim_{k\to\infty} d_c^{(k)}(u, v).$$

If P is a u-v-path in G with $c(P) = d_c(u, v)$, we say that P is a *shortest path* in G with respect to c.

[a] We use the convention that $d_c^{(k)}(u, v) = \infty$ if $\mathcal{W}_{uv} = \emptyset$.

Throughout this chapter, we will be interested in computing the values $d_c(u, v)$ for $u, v \in V$ with the understanding that we can obtain shortest paths from these values by backtracking using depth-first search.

2.1 Generic Search

In the lecture *Algorithmic Discrete Mathematics* we saw several *greedy* algorithms that compute spanning trees and/or shortest paths by growing a component from the starting vertex s. Abstractly, these greedy strategies can be formulated as follows, where the different algorithms differ in their choices of $g(u, v)$.

Algorithm: GenericSearch(G, c, s)

input: graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $(d_{sv})_{v\in V}$, parents $(p_v)_{v\in V}$

$$d_{sv} \leftarrow \begin{cases} 0, & v = s, \\ \infty, & \text{otherwise.} \end{cases}$$
$p_v \leftarrow v \; \forall v \in V$
$R \leftarrow V$ (Insert)
while $R \neq \emptyset$**:**
  $u \leftarrow \arg\min_{v\in R}\{d_{sv}\}$
  $R \leftarrow R \setminus \{u\}$ (ExtractMin)
  for $v \in \Gamma(u) \cap R$**:**
    if $g(u, v) < d_{sv}$**:**
      $d_{sv} \leftarrow g(u, v)$ (DecreaseValue)
      $p_v \leftarrow u$
return $(d_{sv})_{v\in V}$, $(p_v)_{v\in V}$

For example, if we set $g(u, v) = d_{su} + 1$ we recover the simple graph traversal algorithm breadth-first search (BFS) that can be used to compute shortest paths in an unweighted graph. If we set $g(u, v) = c(u, v)$, we obtain Prim's algorithm to compute a minimum spanning tree in an undirected graph. Finally, if we set $g(u, v) = d_{su} + c(u, v)$, then GenericSearch turns into Dijkstra's algorithm for computing shortest paths in directed graphs. We will see another parameterization of the greedy strategy in Section 2.3 below.

We recall the following result for Prim and Dijkstra. Note that BFS can be implemented to run in linear time.

Theorem 2.2. GenericSearch can be implemented with a running time of $\mathcal{O}(m + n \log n)$.

This result relies on an efficient data structure, called a *priority queue,* that supports the operations Insert to insert elements, ExtractMin to extract the element associated with the smallest value, and DecreaseValue to decrease the value associated with some element. We will now show in detail how such a data structure can be obtained.

2.1.1 Fibonacci Heaps

We describe a data structure called a *Fibonacci heap*. A *heap* is a rooted tree T with the additional property that the root of every subtree is associated with the smallest value amongst all values associated to nodes of the subtree. For convenience, in the following we identify nodes with the values associated to them, assuming each value to be unique. With this, a heap is a rooted tree where every subtree has its smallest value as the root.

A Fibonacci heap stores its values in a forest of rooted trees, each of which is a heap. Additionally, it maintains a list R of all roots. The crucial feature of a Fibonacci heap is that it ensures that the heaps it contains have relatively low degrees but contain a relatively large number of nodes. This structure makes extraction of the smallest element easy: Since the smallest element is guaranteed to be a root and the number of roots is relatively small, we can afford to search R to identify the smallest element.

We now describe how a Fibonacci heap maintains its special structure by defining its operations. The operation MakeRoot and MakeChild are auxiliary routines that are used by the main operations. In the following, we write T_x to denote the subtree rooted at node x and $|T_x|$ to denote the number of nodes in T_x.

Algorithm: MakeRoot(x)

$R \leftarrow R \cup \{x\}$
$p_x \leftarrow x$, **unmark** x

Algorithm: MakeChild(y, x)

$R \leftarrow R \setminus \{y\}$
$p_y \leftarrow x$

Algorithm: Insert(x)

MakeRoot(x)
$x_{\min} \leftarrow \min\{x, x_{\min}\}$

Algorithm: ExtractMin

$r \leftarrow x_{\min}$
for each *child* x *of* r**:**
  MakeRoot(x)
$R \leftarrow R \setminus \{r\}$
while R *contains vertices* $x < y$ *of the same degree*:
  MakeChild(y, x)
$x_{\min} \leftarrow \min\{x \in R\}$
return r

Algorithm: DecreaseValue(x)

$x_{\min} \leftarrow \min\{x, x_{\min}\}$
if $x < p_x$**:**
  mark x
  while x **is marked**:
    $y \leftarrow p_x$
    MakeRoot(x)
    $x \leftarrow y$
  if $x \neq p_x$**:**
    mark x

Observe that we allow nodes to lose two children before moving the corresponding subtree to R. Additionally, we defer reducing the length of R until the next ExtractMin operation. We will see that this "laziness" is crucial for the performance of the Fibonacci heap. While this laziness introduces some variability in the shape of the rooted trees, we will show that the shape is closely related to the famous *Fibonacci numbers*.

Definition 2.3. The *Fibonacci numbers* $(F_k)_{k\in\mathbb{N}\cup\{0\}}$ are defined by

$$F_k := \begin{cases} 0 & \text{for } k = 0, \\ 1 & \text{for } k = 1, \\ F_{k-1} + F_{k-2} & \text{for } k \geq 2. \end{cases}$$

The Fibonacci numbers have many beautiful properties. In particular, the following holds (the proof is left as an exercise).

Proposition 2.4. The Fibonacci numbers satisfy

(i) $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ for all $k \in \mathbb{N} \cup \{0\}$,

(ii) $F_{k+2} \geq \phi^k$ for all $k \in \mathbb{N} \cup \{0\}$, where $\phi := (1 + \sqrt{5})/2$ is the golden ratio.

With this, we can establish that the trees of a Fibonacci heap must be relatively large, compared to the degrees of their roots. Here and throughout we assume that Fibonacci heaps start empty and are built up using the above operations.

Lemma 2.5. For every node x with k children in a Fibonacci heap it holds that $|T_x| \geq F_{k+2}$.

Proof. Let s_k be the smallest possible size of a subtree whose root has k children. Observe that s_k is monotonically increasing in k: We can always reduce both the degree and number of vertices of the root by invoking DecreaseValue on a child, which implies $s_{k-1} \leq s_k$. We show that $s_k \geq F_{k+2}$ by induction on k. The induction basis for $k \leq 1$ holds, since $F_3 = 2$ and $F_2 = 1$.

Now let T be a smallest possible tree whose root x has $k \geq 2$ children. Let $c_1, \ldots, c_k$ denote these children in the order in which they were attached to x. Observe that every vertex that is not in the root list R can have lost at most one child during DecreaseValue operations. This means that c_i has $k' \geq i - 2$ children. By induction, and since s_i grows monotonically in i, for $i \geq 2$, we have $|T_{c_i}| \geq s_{k'} \geq s_{i-2} \geq F_i$. Overall, by Proposition 2.4 (i), we obtain

$$s_k = 1 + |T_{c_1}| + \sum_{i=2}^{k} |T_{c_i}| \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i \overset{\text{Prop. 2.4}}{=} F_{k+2}. \qquad \square$$

This immediately implies that Fibonacci trees maintain small degrees.

Corollary 2.6. Every node in a Fibonacci heap with n nodes has at most $\lfloor \log_\phi n \rfloor$ children.

Proof. If there was a node with $k > \log_\phi n$ children, by Proposition 2.4 (ii) and Lemma 2.5, the corresponding subtree would have to have size at least $F_{k+2} \geq \phi^k > n$. $\square$

We are now ready to show that Fibonacci heaps support arbitrary sequences of operations efficiently. Note that the bound below immediately yields the running time of Theorem 2.2.

Theorem 2.7. Fibonacci heaps can be implemented to take time $\mathcal{O}(m + n \log n)$ for inserting $\mathcal{O}(n)$ elements and performing $\mathcal{O}(m)$ ExtractMin and DecreaseValue operations.

Proof. We define a potential function

$$\Phi := \kappa \cdot (|R| + 2n_{\text{marked}}),$$

where n_{marked} is the number of marked nodes, and $\kappa \in \mathbb{N}$ is a suitably large constant. Let $\Phi_0 := 0$ and let Φ_i denote the value of Φ at the end of the i-th Insert, ExtractMin or DecreaseValue operation. Let further t_i denote the running time needed for the i-th operation and $a_i := t_i + \Phi_i - \Phi_{i-1}$ denote the *amortized running time* in step i. Because the potential is always non-negative, the total running time for N operations can be bounded via

$$\sum_{i=1}^{N} t_i = \sum_{i=1}^{N} (a_i - \Phi_i + \Phi_{i-1}) = \sum_{i=1}^{N} a_i - \Phi_N + \Phi_0 \leq \sum_{i=1}^{N} a_i.$$

It thus suffices to bound the amortized running time in each step.

First note that the Insert operation has a constant running time and only changes the potential by a constant amount, hence its amortized running time is $\mathcal{O}(1)$.

The operation ExtractMin consists of three phases. First, all children of the smallest node in the root list R are moved to the root list themselves. By Corollary 2.6, this phase takes time at most $\mathcal{O}(\log n)$ and the increase in potential is also at most $\mathcal{O}(\log n)$. The amortized running time of this phase thus is $\mathcal{O}(\log n)$. In a second phase, we merge trees until no two roots have the same degrees. We can accomplish this going over all roots and inserting them into a list of length $\lfloor \log_\phi n \rfloor + 1$ at a position corresponding to their degree. Whenever this position is already occupied, we merge the two trees and put the resulting tree into the next position of the list, etc. Since, in every step, we either find a free position or reduce the number of trees, the running time of the second phase is linear in $|R|$, i.e., it is bounded by $\kappa' \cdot |R|$ for some constant $\kappa' \in \mathbb{N}$. After the second phase, the root list is R' and has length $|R'| \leq \log_\phi n + 1$, thus the potential is decreased by $\kappa \cdot (|R| - |R'|)$. For $\kappa \geq \kappa'$, we obtain an amortized running time of $\kappa' \cdot |R| - \kappa \cdot (|R| - |R'|) \leq \kappa \cdot |R'| = \mathcal{O}(\log n)$ for the second phase. In the third phase, the new minimum is determined by going over all elements in R'. This takes time linear in $|R'|$ and does not change Φ, hence the amortized running time of this phase is $\mathcal{O}(\log n)$. Overall, ExtractMin has an amortized running time of $\mathcal{O}(\log n)$.

The running time of the DecreaseValue(x) operation is constant if the value of x does not get smaller than the value of its parent (and the potential does not change in this case). Otherwise, the running time is linear in the number $n_{\text{marked}}(x)$ of (consecutive) marked ancestors of x, i.e., it is bounded by $\kappa'' \cdot n_{\text{marked}}(x)$ for some constant κ''. These ancestors are unmarked and added to the root list R. Since at most one vertex is marked by DecreaseValue(x), the change in potential is at most $\kappa \cdot (n_{\text{marked}}(x) + 1) - \kappa \cdot 2n_{\text{marked}}(x) + \kappa \cdot 2$. For $\kappa \geq \kappa''$, we obtain an amortized running time of $\kappa'' \cdot n_{\text{marked}}(x) - \kappa \cdot n_{\text{marked}}(x) + 3\kappa < 3\kappa = \mathcal{O}(1)$.

Finally, observe that there can be at most $\mathcal{O}(n)$ ExtractMin operations. Thus, the total amortized cost for all Insert, ExtractMin and DecreaseValue operations is

$$\mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(m) = \mathcal{O}(m + n \log n).$$

As discussed above, this also bounds the total running time of these operations. Note, however, that we cannot guarantee the individual operations to be efficient. □

2.2 All-Pairs Shortest Paths

We move on to the problem of computing *all* shortest paths in G, or, more precisely, we consider the following problem.

All-Pairs Shortest Paths Problem
input: graph $G = (V, E)$; edge weights $c\colon E \to \mathbb{R}$
problem: find shortest u-v-paths in G for all $u, v \in V$

Of course, we can solve this problem by solving the single-source shortest path problem starting from every vertex $s \in V$. However, optimum substructure (see lecture *Algorithmic Discrete Mathematics*) suggests that we can do better by reusing optimal subpaths instead of recomputing all paths from scratch n times.

Recall that the algorithm of Bellman-Ford(-Moore) is based precisely on optimum substructure in that it extends shortest paths/walks into longer shortest paths/walks in each iteration. The final formulation of BellmanFord(Moore) is a refinement of the following dynamic program (the final algorithm does not explicitly maintain $d_{sv}^{(k)}$ for all $k \in \{1, \dots, n-1\}$ and only iterates over actual arcs in the inner loop).

Algorithm: DynamicSP(G, c, s)
input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

$$d_{sv}^{(0)} \leftarrow \begin{cases} 0, & \text{if } v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

for $k \leftarrow 1, 2, \dots, n-1$**:**
  for $v \in V$**:**
    $d_{sv}^{(k)} \leftarrow \min_{u \in V}\{d_{su}^{(k-1)} + c(u, v)\}$

return $(d_{sv}^{(n-1)})_{v \in V}$

Correctness of this algorithm is based on the following insights (for proofs see lecture *Algorithmic Discrete Mathematics*).

Proposition 2.8. Let $G = (V, E)$ be a directed graph, let $u \in V$, and let $c\colon E \to \mathbb{R}$ not induce negative cycles reachable from u. Then, for all $v \in V$ reachable from u, there exists a shortest u-v-path in G with respect to c and it holds that $d_c(u, v) = d_c^{(n-1)}(u, v)$.

Lemma 2.9. The following recurrence holds:

$$d_c^{(k)}(u, v) = \begin{cases} c(u, v) & \text{if } k = 1, \\ \min_{w \in V}\{d_c^{(k-1)}(u, w) + c(w, v)\} & \text{if } k \geq 2. \end{cases}$$

In particular, DynamicSP correctly computes $d_c^{(n-1)}(s, v)$ for all $v \in V$.

These insights immediately imply the following result.

Theorem 2.10. DynamicSP solves the single-source shortest path problem in time $\Theta(n^3)$ on directed graphs with arcs weights that do not induce negative cycles reachable from s.

Note that DynamicSP only computes shortest paths for graphs without negative cycles. Importantly, however, Lemma 2.9 does allow negative cycles, which means that the values $d_c^{(n-1)}(s,v)$ for all $v \in V$ computed by DynamicSP are still correct in this case. This yields an easy way of verifying whether c induces negative cycles (the proof is left as an exercise):

Proposition 2.11. The graph G contains a negative cycle reachable from s if and only if there are $u, v \in V$ with $d_c^{(n-1)}(s,v) > d_c^{(n-1)}(s,u) + c(u,v)$.

With this, we can solve the all-pairs shortest paths problem naively by repeating DynamicSP from all starting vertices, which yields the algorithm RepeatBFM. A more systematic, but equivalent, approach is algorithm ParallelBFM that computes the values $d_{uv}^{(k)}$ in order of increasing k (i.e., it changes the order of computation of the dynamic program). For convenience, we let $V = \{1, \dots, n\}$ in the following and introduce the cost matrix $C = (c_{uv})_{u,v \in V} \in \mathbb{R}^{n \times n}$ with $c_{uv} := c(u,v)$. Similarly, we organize the values $d_c^{(k)}(u,v)$ and $d_{uv}^{(k)}$ into matrices $D_c^{(k)}$ and $D^{(k)}$.

Algorithm: RepeatBFM(V, C)

input: vertices $V = \{1, \dots, n\}$,
weights $C \in (\mathbb{R} \cup \{\infty\})^{n \times n}$
output: distance matrix $D_c^{(n-1)}$

$D^{(1)} \leftarrow C$
for $u \leftarrow 1, \dots, n$**:**
  for $k \leftarrow 2, \dots, n-1$**:**
    for $v \leftarrow 1, \dots, n$**:**
      $d_{uv}^{(k)} \leftarrow \min_{v' \in V} \{d_{uv'}^{(k-1)} + c(v', v)\}$
return $D^{(n-1)}$

Algorithm: ParallelBFM(G, C)

input: vertices $V = \{1, \dots, n\}$,
weights $C \in (\mathbb{R} \cup \{\infty\})^{n \times n}$
output: distance matrix $D_c^{(n-1)}$

$D^{(1)} \leftarrow C$
for $k \leftarrow 2, \dots, n-1$**:**
  for $u \leftarrow 1, \dots, n$**:**
    for $v \leftarrow 1, \dots, n$**:**
      $d_{uv}^{(k)} \leftarrow \min_{v' \in V} \{d_{uv'}^{(k-1)} + c(v', v)\}$
return $D^{(n-1)}$

We can further refine this naive algorithm by expressing it using matrix multiplications in the *min tropical semiring* (or *min-plus-algebra*), where '+' plays the role of the product and 'min' plays the role of addition (see algorithm MatrixBFM). We denote this matrix multiplication using the $\bigotimes_{\min,+}$-operator.

Of course, the running time of MatrixBFM is still $\Theta(n^4)$. However, this reformulation allows for an elegant trick to improve the running time. Consider the computation of a conventional matrix product $C \times C \times \cdots \times C = C^n$. Assuming that n is a power of 2, we can significantly reduce our computational effort by computing the equivalent product $(C \times \cdots \times C) \times (C \times \cdots \times C) = \cdots = (\dots((C^2)^2)^2 \dots)^2 = C^n$. This product only requires $\log_2 n$ matrix multiplications instead of $n-1$. To apply this reformulation, we only used that the matrix product in associative. It turns out that this is still the case in the tropical semiring (the proof is left as an exercise).

Lemma 2.12. The matrix product $\bigotimes_{\min,+}$ in the tropical semiring is associative.

We can therefore reformulate MATRIXBFM using repeated matrix squaring in order to improve overall performance to $\Theta(n^3 \log n)$ (see algorithm MATRIXSQUARING). This is significantly better than naively computing single-source shortest paths from every starting location via REPEATBFM!

Algorithm: MATRIXBFM(G, C)

input: vertices $V = \{1, \dots, n\}$,
weights $C \in (\mathbb{R} \cup \{\infty\})^{n \times n}$
output: distance matrix D_c

$D^{(1)} \leftarrow C$
for $k \leftarrow 2, \dots, n-1$**:**
  $D^{(k)} \leftarrow D^{(k-1)} \bigotimes_{\min,+} C$
return $D^{(n-1)}$

Algorithm: MATRIXSQUARING(G, C)

input: vertices $V = \{1, \dots, n\}$,
weights $C \in (\mathbb{R} \cup \{\infty\})^{n \times n}$
output: distance matrix D_c

$Q^{(1)} \leftarrow C$
for $k \leftarrow 2, \dots, \lceil \log_2(n-1) \rceil$**:**
  $Q^{(k)} \leftarrow Q^{(k-1)} \bigotimes_{\min,+} Q^{(k-1)}$
return $Q^{(\lceil \log_2(n-1) \rceil)}$

In terms of the underlying dynamic program, we achieved the above improvement by shrinking the dynamic programming table and changing dependencies. We now improve the dynamic program further by identifying a more suitable form of optimum substructure, based on induced subgraphs of the form $G[\{1, 2, \dots, w\}]$ of G for increasing values of $w \in \{0, \dots, n\}$.

Definition 2.13. For $u, v, w \in V = \{1, \dots, n\}$ we define

$$d_{uv}^{(\leq 0)} := c(u, v),$$
$$d_{uv}^{(\leq w)} := \min\left\{ d_{uv}^{(\leq w-1)}, d_{uw}^{(\leq w-1)} + d_{wv}^{(\leq w-1)} \right\}.$$

We formalize the intuition behind this definition:

Lemma 2.14. Let $u, v \in V = \{1, \dots, n\}$ and $w \in V \cup \{0\}$. Every u-v-path in $G[\{1, \dots, w\} \cup \{u, v\}]$ has length at least $d_{uv}^{(\leq w)}$, and, if $d_{uv}^{(\leq w)} < \infty$, there exists a u-v-walk in $G[\{1, \dots, w\} \cup \{u, v\}]$ of length $d_{uv}^{(\leq w)}$.

Proof. We prove the statement via induction on w. The induction basis for $w = 0$ is trivial.

By Definition 2.13, $d_{uv}^{(\leq w)} < \infty$ implies that $d_{uv}^{(\leq w-1)} < \infty$ or $d_{uw}^{(\leq w-1)} + d_{wv}^{(\leq w-1)} < \infty$. In either case, the induction hypothesis gives a u-v-walk in $G[\{1, \dots, w\} \cup \{u, v\}]$ of length $d_{uv}^{(\leq w)}$.It remains to show that $d_{uv}^{(\leq w)} \leq c(P_{uv})$ for every u-v-path P_{uv} in $G[\{1, \dots, w\} \cup \{u, v\}]$.

If $w \notin P_{uv}$, by the induction hypothesis, we have $d_{uv}^{(\leq w)} \leq d_{uv}^{(\leq w-1)} \leq c(P_{uv})$. Otherwise, let $P_{uw} \oplus P_{wv} := P_{uv}$ be such that $P_{uw} \cap P_{wv} = \{w\}$. We have $P_{uw} \subseteq \{1, \dots, w-1\} \cup \{u, w\}$ and $P_{wv} \subseteq \{1, \dots, w-1\} \cup \{w, v\}$. By induction, this implies $d_{uv}^{(\leq w)} \leq d_{uw}^{(\leq w-1)} + d_{wv}^{(\leq w-1)} \leq c(P_{uw}) + c(P_{wv}) = c(P_{uv})$. □

This interpretation of Definition 2.13 suggests a different dynamic program called the *Floyd-Warshall* algorithm.

Algorithm: FLOYDWARSHALL(G, c)

input: vertices $V = \{1, \dots, n\}$, weights $C \in (\mathbb{R} \cup \{\infty\})^{n \times n}$
output: distance matrix D_c

$D^{(\leq 0)} \leftarrow C$
for $w \leftarrow 1, \dots, n$**:**
  for $u \leftarrow 1, \dots, n$**:**
    for $v \leftarrow 1, \dots, n$**:**
      $d_{uv}^{(\leq w)} \leftarrow \min\{d_{uv}^{(\leq w-1)}, d_{uw}^{(\leq w-1)} + d_{wv}^{(\leq w-1)}\}$
return $D^{(\leq n)}$

It is easy to see that this algorithm is correct and yields another (minor) improvement to the running time. Also observe that the distances computed by FLOYDWARSHALL allow to infer which vertex lies on a negative cycle by inspecting the diagonal entries of $D^{(\leq n)}$.

Theorem 2.15. If $c\colon V \to \mathbb{R}$ does not induce negative cycles in $G = (V, E)$, then FLOYDWARSHALL solves the all pairs shortest paths problem in time $\Theta(n^3)$.

Proof. The value $d_{uv}^{(\leq n)}$ are correctly compute by the algorithm by definition. By Lemma 2.14, for every $u, v \in V$ with $d_c(u, v) < \infty$, we have $d_{uv}^{(\leq n)} < \infty$. Also, if $d_{uv}^{(\leq n)} < \infty$, there is a u-v-walk W of length $c(W) = d_{uv}^{(\leq n)} \leq d_c(u, v)$ in G. We can decompose W into a u-v-path P and a set $\mathcal{C}$ of cycles (see lecture *Algorithmic Discrete Mathematics*). Since c does not induce negative cycles, we have $d_c(u, v) \geq d_{uv}^{(\leq n)} = c(W) = c(P) + \sum_{C \in \mathcal{C}} c(C) \geq c(P)$. Since P cannot be shorter than a shortest u-v-path in G, equality must hold, and, in particular, $d_{uv}^{(\leq n)} = d_c(u, v)$. This implies that $D^{(\leq n)} = D_c$ and FLOYDWARSHALL computes the correct solution. The algorithm obviously has cubic running time. □

In fact, the all-pairs shortest paths problem is a very important problem in algorithm theory and many more efficient algorithms have been developed. However, all of these algorithms have a running time of the form $\mathcal{O}(n^3)/n^{o(1)}$, which means that their running times are better than $\mathcal{O}(n^3)$ only by a subpolynomial factor (for example by $\log n / \log \log n$). Because of our inability to obtain better algorithms, many algorithmists believe that the following conjecture holds. This conjecture is sometimes used as a basis for conditional proofs of hardness for other problems.

Conjecture 2.16 (APSP Conjecture)**.** *No algorithm can solve the All-Pairs Shortest Paths problem in time $\mathcal{O}(n^{3-\varepsilon})$ for any fixed $\varepsilon > 0$.*

2.2.1 All-Pairs Dijkstra

If all arc weights are non-negative, we can obtain a running time of $\mathcal{O}(mn + n^2 \log n)$ simply by repeating DIJKSTRA from all start vertices $s \in V$. This is never worse than the running time of FLOYDWARSHALL and better on sparse graphs, i.e., on graphs with $m \in o(n^2)$. We will now see a way of adapting this algorithm for the case where arc weights can be negative. The idea is to modify arc weights in such a way that they become non-negative, while preserving shortest paths. We can achieve this if we can find a potential function in the following sense.

Definition 2.17. The *reduced costs* c_π with respect to $\pi\colon V \to \mathbb{R}$ are defined by

$$c_\pi(u,v) := c(u,v) + \pi(u) - \pi(v),$$

for all $u, v \in V$. If $c_\pi(u,v) \geq 0$ for all $u, v \in V$, we say that π is a *vertex potential*.

Crucially, vertex potentials have the property that they do not alter the set of shortest paths.

Proposition 2.18. Let $\pi\colon V \to \mathbb{R}$ and P be a u-v-path in G with $u, v \in V$. Then, P is a shortest path with respect to c if and only if P is a shortest path with respect to c_π.

Proof. For every u-v-walk W we have

$$c_\pi(W) = \sum_{(a,b)\in W} c_\pi(a,b) = \sum_{(a,b)\in W} (c(a,b) + \pi(a) - \pi(b)) = \pi(u) - \pi(v) + \sum_{(a,b)\in W} c(a,b) = \delta_{uv} + c(W),$$

where $\delta_{uv} := \pi(u) - \pi(v)$ is a constant that only depends on u and v. This means that the costs of all u-v-walks are shifted by the same additive constant. Thus, the set of shortest paths is preserved. Note that, in particular, c induces negative cycles if and only if c_π induces negative cycles. □

Proposition 2.18 implies that, if we have a vertex potential $\pi\colon V \to \mathbb{R}$, we can solve the all-pairs shortest paths problem for arbitrary weights by applying DIJKSTRA using the reduced costs c_π.

Algorithm: ALLPAIRSDIJKSTRA(G, c)
input: vertices $V = \{1, \dots, n\}$, weights $c\colon E \to \mathbb{R}$
output: distances $(d_c(u,v))_{u,v\in V}$

$\pi \leftarrow$ POTENTIAL(G, c)
for $u \in V$ **:**
  $(d_{uv})_{v\in V} \leftarrow$ DIJKSTRA(G, c_π, u)
return $(d_{uv} - \pi(u) + \pi(v))_{u,v\in V}$

We still need a way of finding vertex potentials. This can be accomplished as follows.

Algorithm: POTENTIAL(G, c)
input: vertices $V = \{1, \dots, n\}$, weights $c\colon E \to \mathbb{R}$
output: vertex potential

$s \leftarrow n+1$; $n' \leftarrow n+1$
$G' \leftarrow (V \cup \{s\}, E \cup (\{s\} \times V))$

$$c'(u,v) \leftarrow \begin{cases} 0, & \text{if } u = s \text{ and } v \in V \cup \{s\}, \\ \infty, & \text{if } u \in V \text{ and } v = s, \\ c(u,v), & \text{if } u, v \in V. \end{cases}$$

$(d_{sv}^{(n'-1)})_{v\in V\cup\{s\}} \leftarrow$ BELLMANFORD(MOORE)(G', c', s)
return $(d_{sv}^{(n'-1)})_{v\in V}$

We show that POTENTIAL finds a vertex potential if there are no negative cycles. Conversely, if a vertex potential π exists, then c_π cannot induce negative cycles and Proposition 2.18 implies that c cannot induce negative cycles.

Proposition 2.19. If c does not induce negative cycles, POTENTIAL computes a vertex potential in time $\mathcal{O}(mn)$.

Proof. Since c does not induce negative cycles in G, neither does c' in G'. Proposition 2.11, together with the fact that in G' all vertices are reachable from s, yields that $d^{(n'-1)}_{c',G'}(s,v) \leq d^{(n'-1)}_{c',G'}(s,u) + c'(u,v)$ for all $u, v \in V$. By correctnes of BELLMANFORD(MOORE), it follows that $d^{(n'-1)}_{su} + c(u,v) - d^{(n'-1)}_{sv}(s,v) \geq 0$ for all $u, v \in V$. This means that $\pi(v) := d^{(n'-1)}_{sv}$ defines a vertex potential. The running time of POTENTIAL is dominated by the running time of BELLMANFORD(MOORE) (see lecture *Algorithmic Discrete Mathematics*). □

With this, we obtain the following result.

Theorem 2.20. If c does not induce negative cycles in G, then ALLPAIRSDIJKSTRA solves the all pairs shortest paths problem in time $\mathcal{O}(n(m + n \log n))$.

Proof. Correctness follows from Propositions 2.18 and 2.19. The running time follows from Proposition 2.19 together with the fact that DIJKSTRA can be implemented with a running time of $\mathcal{O}(m+n \log n)$ (Theorem 2.2). □

Remark 2.21. Note that this result does not contradict the APSP Conjecture, since it only provides an improved running time for sparse graphs. In dense graphs, ALLPAIRSDIJKSTRA and FLOYDWARSHALL provide the same theoretical performance guarantee. For practical purposes, FLOYDWARSHALL is often superior in dense graphs, since it is much leaner in terms of the hidden constant in the $\mathcal{O}$-notation and is much easier to implement.

2.2.2 A Duality Approach to Vertex Potentials

We now expose a tight relationship between negative cycles and vertex potentials in terms of linear programming theory. We start with an integer linear programming (ILP) formulation for finding (a collection of) negative closed walks:

(NC)

$$\begin{array}{lll} \min & \sum_{(u,v)\in E} c_{uv} x_{uv} & \\ \text{s.t.} & \sum_{u:(v,u)\in E} x_{vu} = \sum_{u:(u,v)\in E} x_{uv}, & \forall v \in V \\ & x_{uv} \in \mathbb{N} \cup \{0\} & \forall (u,v) \in E \end{array}$$

Clearly, this ILP is feasible, since $x_{uv} = 0$ for all $u, v \in V$ trivially is a feasible solution. If there are no negative cycles, then (NC) is also bounded by 0, otherwise it is unbounded. Furthermore, the polytope defined by the constraints of (NC) is integral, which means that we do not need to require x_{uv} to be integral to have an integral optimum.

Proposition 2.22. The LP-relaxation of (NC) is integral.

Proof. First observe that the constraint matrix of (NC) is the incidence matrix of the underlying graph G. We know that the incidence matrix of a graph is totally unimodular, which implies that the corresponding polytope is integral (see lecture *Discrete Optimization*). Thus, the LP-relaxation of (NC) is integral, as claimed. □

We can therefore consider the LP-relaxation (NCP) instead of (NC).

(NCP)

$$\begin{aligned} \min \quad & \textstyle\sum_{(u,v)\in E} c_{uv} x_{uv} & & \\ \text{s.t.} \quad & \textstyle\sum_{u:(u,v)\in E} x_{uv} - \sum_{u:(v,u)\in E} x_{vu} = 0, & \forall v \in V \\ & x_{uv} \geq 0 & \forall (u,v) \in E \end{aligned}$$

(NCD)

$$\begin{aligned} \max \quad & 0 & & \\ \text{s.t.} \quad & y_v - y_u \leq c_{uv}, & \forall (u,v) \in E \\ & y_v \in \mathbb{R} & \forall v \in V \end{aligned}$$

Observation 2.23. The linear program (NCP) is always feasible, and it is bounded if and only if c does not induce negative cycles.

We will now consider the dual linear program (NCD) and use weak duality to relate the existence of vertex potentials to the existence of negative cycles. Note that the following proposition also follows from Propositions 2.18 and 2.19, as discussed above.

Proposition 2.24. A directed graph $G = (V, E)$ with arc weights $c\colon E \to \mathbb{R}$ admits a vertex potential if and only if c does not induce negative cycles.

Proof. Consider the dual LP (NCD) for (NCP). Trivially, (NCD) is always bounded. By identifying $\pi(v) \equiv y(v)$, we see that (NCD) is feasible (and bounded) if and only if we can find a vertex potential.

On the other hand, by strong duality (see lecture *Introduction to Optimization*), the dual (NCD) is feasible and bounded if and only if the primal (NCP) is feasible and bounded. From Observation 2.23 we know that this is the case precisely if c does not induce negative cycles. □

2.3 Goal-Directed Search

In many applications, in particular routing, we are not interested in all shortest paths in a graph but rather a shortest s-t-path between specific vertices s and t. Intuitively, from a theoretical worst-case perspective, this problem is equivalent to the single-source shortest path problem, since the shortest s-t-path may contain all other shortest paths from s. However, in practice, many heuristic techniques have been developed that are tailored to finding a shortest path to the destination as quickly as possible. While these algorithms do not perform better than general shortest path algorithms in the worst case, they are often superior on real graphs like road networks. We present one of the most important heuristics and establish its correctness.

As a motivation, consider a densely connected graph on points of the plane with arc weights being Euclidean distances (see Figure 2.1). In such a graph, Dijkstra explores a ball of radius $d_c(s,t)$ before finding a shortest s-t-path. We can significantly reduce the search space by invoking Dijkstra simultaneously from s (in $G = (V, E)$) and from t (in the reverse graph $(V, \bar{E})$). The corresponding search space consists of two balls of radius $\frac{1}{2} \cdot d_c(s,t)$, which, intuitively, should contain about half the number of vertices as before. We can further reduce the search space if we have some reasonable estimate $h_t(v)$ for the distance from v to t for all $v \in V$. Such an estimate might, for example, be the length of the straight-line connection between v and t in the plane. With such an estimate, we can direct Dijkstra towards t by favoring vertices with smaller estimates when choosing where to proceed. One way of doing this is the algorithm A* below.

Algorithm: A$^\star(G, c, s, t)$

input: graph $G = (V, E)$,
weights $c\colon E \to \mathbb{R}_{\geq 0}$,
vertices $s, t \in V$

output: distance $d_c(s, t)$

$$d_{sv} \leftarrow \begin{cases} 0, & v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

$R \leftarrow V$

while $t \in R$**:**
- $u \leftarrow \arg\min_{v \in R}\{d_{sv} + h_t(v)\}$
- $R \leftarrow R \setminus \{u\}$
- **for** $v \in \Gamma(u) \cap R$**:**
  - $d_{sv} \leftarrow \min\{d_{sv}, d_{su} + c(u, v)\}$

return d_{st}

Algorithm: A*-Reformulation(G, c, s, t)

input: graph $G = (V, E)$,
weights $c\colon E \to \mathbb{R}_{\geq 0}$,
vertices $s, t \in V$

output: distance $d_c(s, t)$

$$d'_{sv} \leftarrow \begin{cases} h_t(s), & v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

$R \leftarrow V$

while $t \in R$**:**
- $u \leftarrow \arg\min_{v \in R}\{d'_{sv}\}$
- $R \leftarrow R \setminus \{u\}$
- **for** $v \in \Gamma(u) \cap R$**:**
  - $d'_{sv} \leftarrow \min\{d'_{sv}, d'_{su} - h_t(u) + c(u, v) + h_t(v)\}$

return $d'_{st} - h_t(t)$

This algorithm can be expressed as a parametrization of GenericSearch with $g(u, v) = d'_{su} - h_t(u) + c(u, v) + h_t(v)$, see A*-Reformulation. We have to be careful not to be too generous with our estimation $h_t(v)$, otherwise we may lose correctness of the algorithm. The following theorem gives a relatively weak condition for h_t under which A* is guaranteed to compute a shortest s-t-path. Whether or not our choice of h_t actually improves the practical performance entirely depends on the structure of G and c.

Theorem 2.25. If h_t is monotone, i.e., if $h_t(u) \leq c(u, v) + h_t(v)$ for all $(u, v) \in E$, then A* is correct.

Proof. Consider algorithm A*-Reformulation. This algorithm is equivalent to A* with $d'_{sv} = d_{sv} + h_t(v)$. On the other hand, the algorithm is nothing else than Dijkstra(G, c_π, s) with vertex potential $\pi(v) = -h_t(v)$ for all $v \in V$. Observe that π defines a potential, since $c_\pi(u, v) = c(u, v) + \pi(u) - \pi(v) = c(u, v) - h_t(u) + h_t(v) \geq 0$ by monotonicity of h_t. Correctness thus follows from correctness of Dijkstra and Proposition 2.18. □

Note that, by Proposition 2.24, a monotone distance estimate can only exist in the absence of negative cycles.

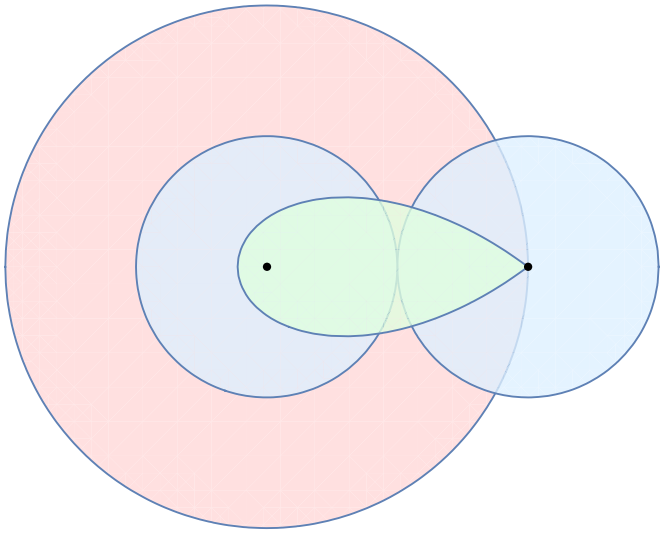

Figure 2.1: An intuitive illustration of the search spaces of DIJKSTRA (red), bidirectional DIJKSTRA (blue), and A* (green) in the plane with Euclidean distances.

3 Maximum Flows

In this chapter we revisit the maximum flow problem defined as follows. We refer to the lecture *Algorithmic Discrete Mathematics* for all basic definitions including residual graphs, augmenting paths and flow decompositions, as well as fundamental theorems like the max-flow-min-cut theorem, Menger's theorem, etc. For the remainder of the chapter we will consider a fixed but arbitrary network $(G = (V, E), \mu, s, t)$.

Maximum Flow Problem
input: network $(G = (V, E), \mu, s, t)$
problem: find s-t-flow in G of maximum value

3.1 Dinic Algorithm

Recall the Edmonds-Karp algorithm that implements the Ford-Fulkerson method for computing a maximum s-t-flow by increasing flow along augmenting paths until we obtain a maximum flow:

Algorithm: EdmondsKarp(G, μ, s, t)
input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$f \leftarrow 0$
while $\exists s$-t-*path in* G_f **:**
  $P \leftarrow$ shortest s-t-path in G_f (unweighted)
  $\delta \leftarrow \min_{e \in P} \mu_f(e)$
  Augment(P, δ)
return f

Recall that the length of the shortest path P is monotonically increasing during an execution of EdmondsKarp. This means that, conceptually, we can subdivide an execution into phases of constant path lengths. We know that there can be at most $n - 1$ such phases, since this is the maximum number of arcs any s-t-path can have. We showed that EdmondsKarp executes each phase in time $\mathcal{O}(m^2)$ on average, which yields an overall running time of $\mathcal{O}(m^2 n)$. We now present the algorithm of Dinic, which is based on an idea how to perform each phase more efficiently.

Intuitively, we aim to execute a phase in a single step, i.e., to simultaneously saturate all shortest s-t-paths in G_f. To do this, we consider the level graph of G_f, which is defined as follows.

Definition 3.1. The *level graph* $G_{\mathrm{L}} = (V, E_{\mathrm{L}})$ of a graph $G = (V, E)$ with respect to a source vertex $s \in V$ is a subgraph of G defined by $E_{\mathrm{L}} := \{(u, v) \in E \mid d(s, v) = d(s, u) + 1\}$.

We can restrict our attention to the level graph without losing shortest paths, while gaining additional structure.

Observation 3.2. The level graph G_{L} is acyclic and contains all shortest s-v-paths of G for $v \in V$.

The fact that the level graph $G_{f,\mathrm{L}}$ of G_f contains all shortest augmenting paths implies that EdmondsKarp increases the flow only along arcs of $G_{f,\mathrm{L}}$ in each iteration. The idea of the Dinic algorithm is to increase flow along more than one path in $G_{f,\mathrm{L}}$ in order to simultaneously saturate all s-t-paths in $G_{f,\mathrm{L}}$. The algorithm achieves this by computing a blocking flow in $G_{f,\mathrm{L}}$, which is defined as follows.

Definition 3.3. An s-t-flow f in a network (G, μ, s, t) is called a *blocking flow* if there is no augmenting path in $G \cap G_f$.

With this, we can state Dinic's algorithm, based on a subroutine for computing blocking flows.

Algorithm: Dinic(G, μ, s, t)
input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$f \leftarrow 0$
while $\exists$ *s-t-path in* G_f **:**
  $G_{f,\mathrm{L}} \leftarrow (V, \{(v, w) \in G_f \mid d_{G_f}(s, w) = d_{G_f}(s, v) + 1\})$
  $f' \leftarrow$ BlockingFlow$(G_{f,\mathrm{L}}, \mu_f, s, t)$
  $f \leftarrow f + f'$
return f

We will now show that the length of a shortest path in G_f is strictly increasing in each iteration of Dinic. In addition, because $G_{f,\mathrm{L}}$ is acyclic, we will be able to efficiently compute a blocking flow.

Proposition 3.4. Dinic computes a maximum s-t-flow in at most $n - 1$ iterations. In each iteration $d_{G_f}(s, t)$ strictly increases.

Proof. By definition of the algorithm, once Dinic terminates, there is no augmenting path in G_f, which implies that f is a maximum s-t-flow (see lecture *Algorithmic Discrete Mathematics*). It remains to show that this happens after $n - 1$ iterations at the latest. To prove this, we show that the distance $d_{G_f}(s, t)$ between s and t in G_f increases in each iteration.

Let f, $f'' = f + f'$ be the flows before and after some iteration of Dinic. We show that $d_{G_f}(s, t) < d_{G_{f''}}(s, t)$, i.e., that shortest s-t-paths in $G_{f''}$ (if any exist) are longer than shortest s-t-paths in G_f. Let $G_{f,\mathrm{L}}$ be the level graph of G_f, and let $G_{f,\bar{\mathrm{L}}} := G_f \cup \bar{G}_{f,\mathrm{L}}$ be the graph that consists of the arcs of G_f and the reverse arcs of $G_{f,\mathrm{L}}$. It holds that $G_{f''} \subseteq G_{f,\bar{\mathrm{L}}}$, since all new arcs in the residual graph after the iteration must be reverse arcs of $G_{f,\mathrm{L}}$. Moreover, $G_{f,\bar{L}}$ contains no arcs (u, v) with $d_{G_f}(s, v) > d_{G_f}(s, u) + 1$: Obviously such arcs cannot be part of G_f and for $(u, v) \in \bar{G}_{f,L}$ it even holds that $d_{G_f}(s, v) = d_{G_f}(s, u) - 1$. In particular, the arcs $(u, v) \in G_{f,\bar{L}}$ with $d_{G_f}(s, v) = d_{G_f}(s, u) + 1$ must be from G_f are thus part of $G_{f,\mathrm{L}}$. Since f' is a blocking flow for $G_{f,\mathrm{L}}$, there is no s-t-path in $G_{f''} \cap G_{f,\mathrm{L}}$. Hence, every shortest s-t-path P in $G_{f''} \subseteq G_{f,\bar{\mathrm{L}}}$ must use at least one arc $(u, v) \in G_{f''} \setminus G_{f,\mathrm{L}}$, which must therefore satisfy $d_{G_f}(s, v) \leq d_{G_f}(s, u)$. We obtain

$$d_{G_f}(s, t) = d_{G_f}(s, t) - d_{G_f}(s, s) = \sum_{(u,v) \in P} (d_{G_f}(s, v) - d_{G_f}(s, u)) \leq |P| - 1 = d_{G_{f''}}(s, t) - 1,$$

i.e., $d_{G_f}(s, t) < d_{G_{f''}}(s, t)$, as claimed. □

3.1.1 Blocking Flows

We still need a way of efficiently computing a blocking flow, i.e., we need to solve the following problem.

Blocking Flow Problem
input: acyclic network $(G = (V, E), \mu, s, t)$
problem: find blocking s-t-flow

Crucially, we restrict ourselves to acyclic inputs, since the level graph for which we need a blocking flow in DINIC is acyclic (Observation 3.2). This gives us the existence of a *topological ordering* of the graph.

Definition 3.5. A topological ordering of a graph $G = (V, E)$ is an ordering $\{v_1, \dots, v_n\} = V$ of the vertices of G with $i < j$ for all $(v_i, v_j) \in E$.

A topological ordering can be computed efficiently (proof left as an exercise).

Proposition 3.6. A directed graph is acyclic if and only if it has a topological ordering. In that case a topological ordering can be computed in time $\mathcal{O}(n + m)$.

A natural approach to finding a blocking flow is to "push" as much flow forward, in topological order, as possible. If we never decrease flow values, the last vertex, before t, that is still overflowing will never be able to push more flow. So we may as well send flow back until it is not overflowing anymore and afterwards keep all flow entering and leaving this vertex unchanged. We can continue alternating push and balancing phases, until no vertex (other than t) has excess left. We obtain the following algorithm.

Algorithm: BLOCKINGFLOW(G, μ, s, t)
input: acyclic network $(G = (V, E), \mu, s, t)$
output: blocking flow

$(s = v_1, \dots, v_n = t) \leftarrow$ TOPOLOGICALSORT(G) (we set $\delta^-(s) = \delta^+(t) = \emptyset$)
$L_{v \in V} \leftarrow \emptyset$, $f \leftarrow 0$, $\mathrm{ex}_f(s) \equiv \infty$
for $n - 1$ *times*:
  /* push step */
  for $v \leftarrow v_1, \dots, v_n$ *with* $\mathrm{ex}_f(v) > 0$:
    for $(v, w) \in \delta^+(v)$ *with* w **not marked**:
      $e \leftarrow (v, w)$
      $\gamma \leftarrow$ PUSH(v, e, ∞) (send as much flow as possible)
      $L_w \leftarrow L_w \oplus (e, \gamma)$ (remember push)
  /* balancing step */
  $i \leftarrow \max\{i < n \mid \mathrm{ex}_f(v_i) > 0\}$
  while $\mathrm{ex}_f(v_i) > 0$ *and* $v_i \neq s$:
    $(L_{v_i} \oplus (e, \gamma)) \leftarrow L_{v_i}$ (extract last edge)
    PUSH$(v_i, \bar{e}, \gamma)$ (send as much flow as possible back)
  mark v_i
return f

Algorithm: PUSH$(v, e, \gamma_{\max}$ [default $= \infty$]$)$

$\gamma \leftarrow \min\{\mathrm{ex}_f(v), \mu_f(e), \gamma_{\max}\}$
AUGMENT(e, γ)
return γ

Note that the function $f\colon E \to \mathbb{R}_{\geq 0}$ that the above algorithm is maintaining is not a valid flow until the very end, since it does not satisfy flow conservation. Observe however, that no vertex other than s may underflow, i.e., have negative excess. Functions with relaxed flow conservation in this sense will become important later, which warrants the following definition.

Definition 3.7. A function $f\colon E \to \mathbb{R}_{\geq 0}$ is an *s-t-preflow* in the network $(G = (V, E), \mu, s, t)$ if it satisfies:

(i) (capacity constraints) $f(e) \leq \mu(e)$ for all arcs $e \in E$,

(ii) (overflow) $\mathrm{ex}_f(v) \geq 0$ for all vertices $v \in V \setminus \{s\}$.

We establish the key invariant of the BLOCKINGFLOW algorithm.

Lemma 3.8. After BLOCKINGFLOW performs a balancing step with vertex v_i, it maintains $\mathrm{ex}_f(v_i) = 0$ unless $v_i = s$, and the fact that there is no v_i-t-path in $G_f \cap G$.

Proof. When starting a balancing step with v_i, there cannot be an overflowing vertex in $\{v_{i+1}, \ldots, v_{n-1}\}$, i.e., behind v_i in the topological order. By definition of a topological ordering, flow is only sent to predecessors of v_i in the topological order during the balancing step. This means that the balancing step maintains $\mathrm{ex}_f(v_j) = 0$ for all $j \in \{i+1, \ldots, n-1\}$. We say that all these vertices are in a balanced state. Since the incoming flow at each vertex is undone in reverse order of when it was created, none of the vertices v_j, $j \in \{i+1, \ldots, n\}$ will send flow back to v_i in future balancing steps (provided they do not receive additional flow from v_i). In addition, since v_i is marked after the balancing step, it will never receive new flow in future push steps. If $v_i \neq s$, then the balancing step continues until v_i is in a balanced state, therefore $\mathrm{ex}_f(v_i) = 0$ is maintained from this time onward.

For the second part of the claim we use induction on the number of performed balancing steps. The claim trivially holds at the beginning. At the time when v_i is selected for the balancing step, it overflows. This means that v_i can only have outgoing arcs to marked vertices in $G_f \cap G$, otherwise it could have pushed more flow. All these vertices were previously selected for balancing steps. This means that, by induction, there are no paths from v_i's neighbors to t in $G_f \cap G$ now and in the future. As discussed above, the flow to successors of v_i in the topological order will never be undone and v_i will never receive new flow, which means that the set of neighbors of v_i in $G_f \cap G$ cannot grow. Thus, there will not be a v_i-t-path in $G_f \cap G$ from this point in time onward. □

With this, we are ready to show correctness and running time of BLOCKINGFLOW.

Theorem 3.9. BLOCKINGFLOW computes a blocking flow in time $\mathcal{O}(n^2)$.

Proof. By Lemma 3.8, BLOCKINGFLOW computes a blocking flow: The lemma implies that no vertex is selected in more than one balancing step, and after all vertices have been selected once, at the latest, no vertex, except

for s, has any overflow. Thus, in iteration $n-1$ at the latest, s is eventually selected in a balancing step, and we have that there is no s-t-path in $G_f \cap G$ in the end, which means that f is a blocking flow.

By Proposition 3.6, the initial topological sort can be computed in time $\mathcal{O}(n+m)$. Since we can omit iterations in push steps for arcs (v,w) where $\mathrm{ex}_f(v) = 0$ or w is marked, it suffices to analyze the number of PUSH operations. First observe that the number of PUSH operations in balancing steps is bounded by the number of PUSH operations in push steps. Therefore, it suffices to bound the number of PUSH operations in push steps. Since every vertex only pushes flow forward until some arc is not saturated by a push, and v_n does not perform any pushes, the number of unsaturating pushes is bounded by $n-1$ in each iteration. For every saturating push we observe that the capacity along the arc only becomes available again in the future once a balancing step is performed with the other endpoint of the arc. Therefore this arc will not be used again in a push step. Hence, the total number of saturating pushes over all iterations is at most m. Overall, the total number of PUSH operations can be bounded by $\mathcal{O}(n^2+m) = \mathcal{O}(n^2)$. Our claim for the running time follows, since each PUSH operation can be performed in constant time. □

Finally, using BLOCKINGFLOW as a subroutine for DINIC yields the following running time. Note the improvement over EDMONDSKARP for dense graphs, where we can expect many shortest augmenting paths in each phase of the algorithm.

Theorem 3.10. DINIC can be implemented with a running time of $\mathcal{O}(n^3)$.

Proof. We can compute the level graph in time $\mathcal{O}(m)$ using BFS. The statement therefore follows from Proposition 3.4 and Theorem 3.9. □

3.2 Push-Relabel Algorithm

We now turn to a quite different approach to computing maximum flows. Intuitively, this approach is motivated by physical water networks, where flow is naturally maximized simply because water flows downhill whenever possible. With this in mind, we can try to obtain a maximum flow simply by emulating a system where the source vertex is at a higher elevation than the sink, and by simulating the water flow. The following definition serves as a mathematical basis for implementing this idea.

Definition 3.11. A function $h\colon V \to \mathbb{N}\cup\{0\}$ is a *height function* with respect to an s-t-preflow f if the following hold:

(i) $h(s) = n$,

(ii) $h(t) = 0$,

(iii) $h(u) \leq h(v)+1$ for all $(u,v) \in G_f$.

Note that we forbid arcs to go downhill by more than one unit of height, which implies the following.

Observation 3.12. Let h be a height function with respect to an s-t-preflow f. For every u-v-path P in G_f it holds that $h(u) \leq h(v) + |P|$.

In particular, the height of vertices with a path to s is limited by $2n-1$. We show next that this includes all overflowing vertices.

Lemma 3.13. If f is an s-t-preflow, then s is reachable in G_f from every vertex in $\{v \in V \mid \mathrm{ex}_f(v) > 0\}$.

Proof. Let $v' \in \{v \in V \mid \mathrm{ex}_f(v) > 0\}$ be an overflowing vertex, and let R denote the set of all vertices reachable from v' in G_f. Then, $\mu_f(e) = 0$ for all $e \in \delta^+_{G \cup \bar{G}}(R)$ and, in particular, $f(e) = 0$ for all $e \in \delta^-_G(R)$. By definition of excess, we thus have

$$\begin{aligned}
\sum_{v \in R} \mathrm{ex}_f(v) &= \sum_{v \in R} \left(\sum_{(u,v) \in E} f(u,v) - \sum_{(v,w) \in E} f(v,w) \right) \\
&= \sum_{v \in R} \left(\sum_{(u,v) \in E \setminus R^2} f(u,v) - \sum_{(v,w) \in E \setminus R^2} f(v,w) \right) \\
&= \sum_{e \in \delta^-_G(R)} f(e) - \sum_{e \in \delta^+(R)} f(e) \\
&= - \sum_{e \in \delta^+(R)} f(e) \leq 0.
\end{aligned}$$

But since $\mathrm{ex}_f(v') > 0$, there must be a vertex $u \in R$ with $\mathrm{ex}_f(u) < 0$. By definition of a preflow, this can only be s. Hence s is reachable from v' in G_f, as claimed. □

We can initially set up a preflow and height function by giving s a large height and by completely saturating all arcs in $\delta^+(s)$ (here and throughout we assume $\delta^-(s) = \delta^+(t) = \emptyset$, otherwise, we remove these arcs from the graph without changing the maximum flow value). At this point, the vertices in $\Gamma(s)$ have positive excess. Now the idea is to gradually increase the height of overflowing vertices and to simulate water flowing downhill whenever possible. The definition of a height function requires that we raise vertices one unit of height at a time, immediately simulate all water flow, and stop increasing the height once a vertex is not overflowing anymore. Ideally, we repeat this process until no vertex other than t has positive excess. We now show that if we reach this point, we are guaranteed to have computed a maximum flow!

Lemma 3.14. Let f be an s-t-flow in the network (G, μ, s, t). If any height function with respect to f exists, then f is a maximum flow.

Proof. By Observation 3.12, every s-t-path in G_f must contain at least $h(s) - h(t)$ arcs. But, by definition, we have $h(s) - h(t) = n$ and an s-t-path cannot consist of more than $n - 1$ arcs. Hence, no augmenting path can exist in G_f, which implies that f is a maximum s-t-flow (see lecture *Algorithmic Discrete Mathematics*). □

We are now ready to formulate our approach as an algorithm. This algorithm is called the Goldberg-Tarjan algorithm, or the push-relabel algorithm.

Algorithm: GOLDBERGTARJAN(G, μ, s, t)

input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$$f(e) \leftarrow \begin{cases} \mu(e), & \text{for } e \in \delta^+(s), \\ 0, & \text{for } e \in E \setminus \delta^+(s). \end{cases}$$

$$h(v) \leftarrow \begin{cases} n, & \text{for } v = s, \\ 0, & \text{for } v \in V \setminus \{s\}. \end{cases}$$

while $\exists u \in V \setminus \{t\}$ *with* $\mathrm{ex}_f(u) > 0$**:**
  if $\exists e = (u, v) \in \delta^+_{G_f}(u)$ *with* $h(u) = h(v) + 1$**:**
    PUSH(u, e, ∞) (push)
  else
    $h(u) \leftarrow \min_{v \in \Gamma_{G_f}(u)}\{h(v) + 1\}$ (relabel)

return f

Algorithm: RELABELTOFRONT(G, μ, s, t)

input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$$f(e) \leftarrow \begin{cases} \mu(e), & \text{for } e \in \delta^+(s), \\ 0, & \text{for } e \in E \setminus \delta^+(s). \end{cases}$$

$$h(v) \leftarrow \begin{cases} n, & \text{for } v = s, \\ 0, & \text{for } v \in V \setminus \{s\}. \end{cases}$$

while $\exists u \in \arg\max_{v \in V \setminus \{t\}}\{h(v) \mid \mathrm{ex}_f(v) > 0\}$**:**
  if $\exists e = (u, v) \in \delta^+_{G_f}(u)$ *with* $h(u) = h(v) + 1$**:**
    PUSH(u, e, ∞) (push)
  else
    $h(u) \leftarrow \min_{v \in \Gamma_{G_f}(u)}\{h(v) + 1\}$ (relabel)

return f

We show that we never need to raise vertices above a height of $2n$ in order to eliminate all excess.

Lemma 3.15. GOLDBERGTARJAN maintains a height function h with respect to the preflow f with $h(v) < 2n$ for all $v \in V$.

Proof. At the beginning of the algorithm we have $h(s) = n$ and $h(v) = 0$ for all $v \in V \setminus \{s\}$. Moreover, all arcs in $\delta^+(s)$ are saturated, hence G_f does not have any "downhill" arcs and h is a height function.

During a relabel step, h remains a height function by definition. Furthermore, we only apply relabeling steps to vertices u with $\mathrm{ex}_f(u) > 0$. At this point, by Observation 3.12, there exists a u-s-path P in G_f. By Lemma 3.13 it further follows that $h(u) \leq h(s) + |P| \leq h(s) + n - 1 = 2n - 1$.

During a PUSH operation along arc $e = (u, v)$ the height function h is not changed, but the reverse arc (v, u) may be created in G_f. That is no problem, since $h(v) = h(u) - 1 < h(u) + 1$ ensures that h remains a height function with respect to G_f. □

We have bounded the maximum height of all vertices and it remains to show that GOLDBERGTARJAN keeps increasing heights, which then implies that the algorithm eventually terminates with f being an s-t-flow.

Lemma 3.16. In every relabel step GOLDBERGTARJAN strictly increases the height of some vertex, and heights never decrease.

Proof. The height of a vertex v can only change during a relabel step, which is only applied once no push step is possible anymore. By Lemma 3.15, h is a height function, hence all arcs $(u, v) \in \delta^+_{G_f}(u)$ satisfy $h(v) \geq h(u)$ before a relabel step for u. By Lemma 3.13, $\mathrm{ex}_f(u) > 0$ implies that a u-s-path exists in G_f, and, in particular, $\delta^+_{G_f}(u) \neq \emptyset$. Thus, the height of u increases during the relabel step. □

From Lemmas 3.14 and 3.15 it follows that GOLDBERGTARJAN is correct. We proceed to bound the number of operations needed by the algorithm to obtain a bound on its running time.

Proposition 3.17. GoldbergTarjan computes a maximum s-t-flow using $\mathcal{O}(n^2m)$ Push and $\mathcal{O}(n^2)$ relabel operations.

Proof. By Lemma 3.15, the algorithm maintains a height function. When GoldbergTarjan terminates, no vertex except t may overflow, thus f in an s-t-flow. By Lemma 3.14, f thus is a maximum flow.

We analyze the number of relabel steps. By Lemma 3.16, we increase the height by at least one during each relabel step. By Lemma 3.15, the number of relabel steps is thus bounded by $\mathcal{O}(n^2)$.

Now consider the number of push steps in which the corresponding arc (u, v) is saturated. Since the heights of vertices can never decrease (Lemma 3.16), the height of v must increase by at least two before a push step can be performed along the reverse arc, after which a push step with (u, v) can occur again. Since the height is bounded by $2n - 1$ (Lemma 3.15), the number of saturating push steps along the same arc is thus bounded by $\mathcal{O}(n)$. The total number of saturating push steps thus is in $\mathcal{O}(mn)$.

To bound the number of non-saturating push steps, we consider the potential $\Phi = \sum_{v:\mathrm{ex}_f(v)>0} h(v)$. During a relabel step, the potential can increase by at most $2n - 1$ (Lemma 3.15). During each push step at most one new overflowing vertex is created. During a saturating push step this may increase the potential by at most $2n - 2$ (Lemma 3.15). During a non-saturating push step along arc (u, v), however, u loses its excess completely. Since $h(u) = h(v)+1$, the potential therefore decreases by one (at least – since no new overflowing vertex may have been created). We already know that the number of relabel and saturating push steps is $\mathcal{O}(n^2 + mn) = \mathcal{O}(mn)$ (since we can restrict ourselves to the part of the graph connected to s and t, such that $n \in \mathcal{O}(m)$). During these steps the potential increases by $\mathcal{O}(n)$, hence the potential can only decrease at most $\mathcal{O}(n^2m)$ times overall (every decrease is by at least one, we initially have $\Phi = 0$, and $\Phi \geq 0$ throughout). Hence, the total number of non-saturating push steps is at most $\mathcal{O}(n^2m)$. □

Proposition 3.17 suggests that GoldbergTarjan does not perform better than Dinic. However, we can further improve its running time by choosing the vertex more carefully that we consider in each iteration of the algorithm. The version of GoldbergTarjan that always selects an overflowing vertex of largest height is often called relabel-to-front algorithm. We show that this algorithm requires significantly fewer operations.

Proposition 3.18. RelabelToFront only requires $\mathcal{O}(n^2\sqrt{m})$ push steps.

Proof. Recall from the proof of Proposition 3.17 that the number of saturating push steps is bounded by $\mathcal{O}(mn)$. Since we can restrict ourselves to the part of the graph connected to s and t, this falls within the claimed running time. Hence it suffices to bound the number of non-saturating push steps.

Let $h_{\max} := \max_{v\in V:\, \mathrm{ex}_f(v)>0} h(v)$ be the largest height among all overflowing vertices. We divide the execution of the algorithm into maximal phases in such a way that $h_{\max}$ remains constant during each phase. Since h can only change during a relabel step and can only increase (Lemma 3.16), and since $h < 2n$ (Lemma 3.15), the total increase of $\sum_{v\in V} h(v)$ is at most $2n^2$. Also $h_{\max}$ can only increase together with $\sum_{v\in V} h(v)$ during a relabel step and thus at most $2n^2$ times. Since $h_{\max} \geq 0$ and initially $h_{\max} = 0$, we obtain that $h_{\max}$ can only decrease at most $2n^2$ times (each time by at least one). Hence, the number of phases is at most $4n^2 = \mathcal{O}(n^2)$.

First consider "cheap" phases with at most $\sqrt{m}$ non-saturating push steps. The total number of non-saturating push steps in all these phases together is $\mathcal{O}(n^2\sqrt{m})$. It remains to consider "expensive" phases.

We define the potential

$$\Phi = \sum_{v\in\{v'\in V \,|\, \mathrm{ex}_f(v')>0\}} \Phi(v) := \sum_{v\in\{v'\in V \,|\, \mathrm{ex}_f(v')>0\}} |\{w \in V \,|\, h(w) \leq h(v)\}| \,.$$

Since $\Phi(v) \leq n$ holds for all $v \in V$, we initially have $\Phi \leq n^2$ and every relabel or saturating push step can increase Φ by at most $n-1$. A non-saturating push step reduces Φ. Since the number of relabel and saturating push steps is at most $\mathcal{O}(mn)$ (Proposition 3.17) and each such step can increase Φ by at most n, the amount by which Φ can be reduced over all non-saturating push steps is at most $\mathcal{O}(n^2 m)$.

Now consider an expensive phase and only the vertices of height $h_{\max}$ during that phase. Note that heights remain unchanged during a phase: Only vertices with excess and of maximum height are candidates for relabel steps. Thus, once a relabel step occurs (or when no vertex other than t has excess) the phase ends. In particular, the set $V_{\max} := \{v \in V \mid h(v) = h_{\max}\}$ remains unchanged during the phase. With every non-saturating push step a vertex in $V_{\max}$ loses its excess. Since there are more than $\sqrt{m}$ such steps in an expensive phase, we have $|V_{\max}| > \sqrt{m}$. It follows that $\Phi(v) > \sqrt{m}$ for all $v \in V_{\max}$. Every time when a vertex loses its excess, Φ thus decreases by more than $\sqrt{m}$ (note that a new vertex u with excess may be created, but for this vertex we have $\Phi(v) - \Phi(u) > \sqrt{m}$, since $h(u) < h(v)$). Since Φ can only be reduced by $\mathcal{O}(n^2 m)$ overall, the total number of non-saturating push steps in expensive phases is bounded by $\mathcal{O}(n^2\sqrt{m})$. □

In order to implement RelabelToFront efficiently, we need to ensure that each iteration of the algorithm needs constant time. We can achieve this by explicitly maintaining $h_{\max} := \max\{h(v) \mid v \in V \wedge \mathrm{ex}_f(v) > 0\}$ as well as the lists $L(h) = \{v \in V \setminus \{t\} \mid h(v) = h \wedge \mathrm{ex}_f(v) > 0\}$ for each height $h \in \{0, \dots, 2n-1\}$. Additionally, we store lists $E_v = \{(v,w) \in \delta^+_{G_f}(v) \mid h(v) = h(w) + 1\}$ for every vertex $v \in V$ that contain the arcs in $\delta^+_{G_f}(v)$ going downhill. Note that we only ever need to access or modify the first or last elements of these lists, which can be accomplished in constant time. The following gives an efficient implementation using these lists.

Algorithm: RelabelToFront (implementation)

input: network $(G = (V,E), \mu, s, t)$
output: maximum s-t-flow

$f(e) \leftarrow \begin{cases} \mu(e), & \text{for } e \in \delta^+(s), \\ 0, & \text{for } e \in E \setminus \delta^+(s). \end{cases}$

$h(v) \leftarrow \begin{cases} n, & \text{for } v = s, \\ 0, & \text{for } v \in V \setminus \{s\}. \end{cases}$

$L(h) \leftarrow \emptyset$ for all $h \in \{1, \dots, 2n-1\}$
$h_{\max} \leftarrow 0$, $L(0) \leftarrow \Gamma(s) \setminus \{t\}$
$E_v \leftarrow \emptyset$ for all $v \in V$
while $\exists v \in L(h_{\max})$**:** (take first element)
  if $\exists e \in E_v$**:** (take first element)
    Push(v, e)
  else
    Relabel(v)
return f

Algorithm: Push$(v, e \equiv (v,w))$ (impl.)

if $\mathrm{ex}_f(w) = 0$ **and** $w \notin \{s,t\}$**:**
  $L(h(w)) \leftarrow L(h(w)) \oplus (w)$
Augment$(e, \min\{\mathrm{ex}_f(v), \mu_f(e)\})$
if $\mathrm{ex}_f(v) = 0$**:**
  $L(h_{\max}) \leftarrow L(h_{\max}) \setminus \{v\}$ (remove first)
  while $h_{\max} > 0$ **and** $L(h_{\max}) = \emptyset$**:**
    $h_{\max} \leftarrow h_{\max} - 1$
if $\mu_f(e) = 0$**:**
  $E_v \leftarrow E_v \setminus \{e\}$ (remove first)

Algorithm: Relabel(v) (implementation)

$L(h_{\max}) \leftarrow L(h_{\max}) \setminus \{v\}$ (remove first)
$h(v) \leftarrow \min\{h(w) + 1 \mid w \in \Gamma_{G_f}(v)\}$
$E_v \leftarrow \{(v,w) \in \delta^+_{G_f}(v) \mid h(v) = h(w) + 1\}$
$h_{\max} \leftarrow h(v)$
$L(h_{\max}) \leftarrow \{v\}$

We analyze the running times of this implementation.

Observation 3.19. The running time for a Relabel operation for vertex v is $\Theta(\deg_G(v))$.

Lemma 3.20. The running time for k Push operations is $\mathcal{O}(k + n^2)$.

Proof. Except for updating $h_{\max}$, the running time of a PUSH operation is constant. Since h is never reduced (Lemma 3.16) and $h < 2n$ (Lemma 3.15), the total increase of $\sum_{v \in V} h(v)$ is bounded by $\mathcal{O}(n^2)$. Thus, the total decrease of $h_{\max}$ is also bounded by $\mathcal{O}(n^2)$, which proves the claim. □

With this, we can bound the running time of RELABELTOFRONT. Observe that the running time is never worse than that ofDINIC and better on sparse graphs.

Theorem 3.21. RELABELTOFRONT can be implemented with a running time of $\mathcal{O}(n^2\sqrt{m})$.

Proof. As shown above (Proposition 3.18), the number of PUSH operations is $\mathcal{O}(n^2\sqrt{m})$. By Lemma 3.20, the running time for all these operations together is thus $\mathcal{O}(n^2\sqrt{m} + n^2) = \mathcal{O}(n^2\sqrt{m})$. We have $\mathcal{O}(n)$ RELABEL operations for each vertex (Proposition 3.17), with a total running time of $\mathcal{O}(mn) = \mathcal{O}(n^2\sqrt{m})$ (Observation 3.19). □

4 Minimum-Cost Flows

In the previous chapter, we focused on the maximum flow problem as a framework for algorithmic problems where a throughput in a network needs to be maximized. In practice, we are often not only interested in maximizing throughput, but also on simultaneously minimizing the cost of our solution with respect to some cost function. For example, in a logistics network, we are not only interested in maximizing the amount of goods that we can deliver, but we also want to minimize (among other factors) the total distance we need to travel. To account for this, we now consider the problem of finding a maximum flow that has minimum cost among all maximum flows. Since we want to restrict ourselves to maximum flows, it will be useful to introduce a notion of networks with prescribed demands and supplies associated with all vertices. In this setting, we implicitly fix the total desired flow value, instead of having source and sink vertices with unbounded supply and demand and maximizing the flow.

Definition 4.1. A *circulation network* is a triple (G, μ, b) where $G = (V, E)$ is a directed graph, $\mu\colon E \to \mathbb{R}_{\geq 0}$ are capacities, and $b\colon V \to \mathbb{R}$ are balances satisfying $\sum_{v \in V} b(v) = 0$. As for networks, to simplify notation, we additionally require that $\{(u, v), (v, u)\} \not\subseteq E$ for all $u, v \in V$.

We need to adapt our definition of a flow accordingly. Note that we no longer need flow conservation.

Definition 4.2. A function $f\colon E \to \mathbb{R}_{\geq 0}$ is a *b-flow* in a circulation network (G, μ, b) if $f(e) \leq \mu(e)$ for all $e \in E$ and $\mathrm{ex}_f(v) = -b(v)$ for all $v \in V$.

Note how, intuitively, positive balances $b(v)$ act as supplies while negative balances act as demands.

The following observation summarizes how network flows and b-flows in circulation networks relate. In particular, observe that a circulation network fixes the flow value of the s-t-flow in the corresponding network. This allows us to easily restrict ourselves to maximum flows (e.g., by setting $b(s) = |f^\star| = -b(t)$ to transition from a network with maximum flow value $|f^\star|$ to a circulation network with prescribed flow value $|f^\star|$).

Observation 4.3. Let (G, μ, b) be a circulation network with $G = (V, E)$ and (G', μ', s, t) be a network with $G' = (V \cup \{s, t\}, E \cup (\{s\} \times V) \cup (V \times \{t\}))$. Let further $\mu'(s, v) = \max\{0, b(v)\}$ and $\mu'(v, t) = \max\{0, -b(v)\}$ for all $v \in V$, and $\mu'|_E = \mu$. Then, there is a b-flow f in (G, μ, b) if and only if there is an s-t-flow f' in (G', μ', s, t) with $|f'| = \frac{1}{2} \sum_{v \in V} |b(v)|$, and we may choose f and f' such that $f'|_E = f$.

This observation allows us to formulate the minimum-cost flow problem without explicitly demanding maximum flows:

Minimum-Cost Flow Problem
input: circulation network $(G = (V, E), \mu, b)$; arc costs $c\colon E \to \mathbb{R}$
problem: find b-flow f of minimum cost $c(f) := \sum_{e \in E} f(e) \cdot c(e)$

For convenience, we consider an arbitrary but fixed circulation network $(G = (V, E), \mu, b)$ with fixed costs $c\colon E \to \mathbb{R}$ throughout this chapter. We will often operate on residual graphs of G and we set $c(\bar{e}) := -c(e)$ for the cost of reverse arcs $\bar{e} \in \bar{E}$.

We begin by reformulating the max-flow min-cut theorem (see lecture *Algorithmic Discrete Mathematics*) for b-flows.

Proposition 4.4. A b-flow exists if and only if, for all $X \subseteq V$, it holds that

$$\sum_{e \in \delta^+(X)} \mu(e) \geq \sum_{v \in X} b(v).$$

Proof. Let G' be the graph of Observation 4.3. By the max-flow min-cut theorem, there is an s-t-flow of value (at least) $\frac{1}{2}\sum_{v\in V} |b(v)|$ in (G', μ', s, t), and therefore a b-flow in (G, μ, b), if and only if

$$\sum_{e \in \delta^+_{G'}(\{s\} \cup X)} \mu'(e) \geq \frac{1}{2} \sum_{v \in V} |b(v)| \tag{4.1}$$

for all $X \subseteq V$. In addition, since $\sum_{v\in V} b(v) = 0$ by definition,

$$\begin{aligned}
\sum_{e \in \delta^+_{G'}(\{s\} \cup X)} \mu'(e) &= \sum_{e \in \delta^+_G(X)} \mu(e) + \sum_{v \in V \setminus X} \max\{0, b(v)\} + \sum_{v \in X} \max\{0, -b(v)\} \\
&= \sum_{e \in \delta^+_G(X)} \mu(e) + \sum_{v \in V} \max\{0, b(v)\} - \sum_{v \in X} \max\{0, b(v)\} + \sum_{v \in X} \max\{0, -b(v)\} \\
&= \sum_{e \in \delta^+_G(X)} \mu(e) + \frac{1}{2} \sum_{v \in V} |b(v)| + \sum_{v \in X} (\max\{0, -b(v)\} - \max\{0, +b(v)\}) \\
&= \sum_{e \in \delta^+_G(X)} \mu(e) + \frac{1}{2} \sum_{v \in V} |b(v)| - \sum_{v \in X} b(v).
\end{aligned}$$

Together with (4.1) this concludes the proof. □

As discussed above, finding a valid b-flow is equivalent to finding a maximum network flow. This can be accomplished as follows:

Algorithm: bFlow(G, μ, b)

---

input: circulation network $(G = (V, E), \mu, b)$
output: b-flow if one exists

$G' \leftarrow (V \cup \{s, t\}, E \cup (\{s\} \times V) \cup (V \times \{t\}))$

$$\mu'(u, v) \leftarrow \begin{cases} \max\{0, b(v)\}, & \text{for } u = s, v \in V \\ \max\{0, -b(u)\}, & \text{for } u \in V, v = t, \\ \mu(u, v), & \text{for } u \in V, v \in V. \end{cases}$$

$f \leftarrow$ RelabelToFront(G', μ', s, t)
if $2|f| = \sum_{v\in V} |b(v)|$**:**
  return $f|_E$
else
  fail

Proposition 4.5. If a b-flow exists, bFlow finds one in time $\mathcal{O}(n^2\sqrt{m})$.

Proof. This follows from Observation 4.3 and the running time of RelabelToFront (Theorem 3.21). □

Intuitively, every pair of b-flows differ by a set of cycles. The following notion will be important to formalize this intuition.

Definition 4.6. A *circulation* is a function that is a b-flow in the circulation network (G, μ, b) with $b = 0$.

We move on by defining the difference between b-flows.

Definition 4.7. For a b-flow f and a b'-flow f' we define $f' \ominus f := g$ with $g(e) = \max\{0, f'(e) - f(e)\}$ and $g(\bar{e}) = \max\{0, f(e) - f'(e)\}$ for all $e \in E$.

The following shows in particular that two b-flows differ by a circulation.

Lemma 4.8. For a b-flow f and a b'-flow f' it holds that $g = f' \ominus f$ is a $(b' - b)$-flow in $(G_f, \mu_f, b' - b)$ with $c(g) = c(f') - c(f)$.

Proof. By definition, $g \geq 0$. For every vertex $v \in V$ of G_f it holds that

$$\begin{aligned}
-\mathrm{ex}_g(v) &= \sum_{e \in \delta^+_{G \cup \bar{G}}(v)} g(e) - \sum_{e \in \delta^-_{G \cup \bar{G}}(v)} g(e) \\
&= \sum_{e \in \delta^+_G(v)} g(e) - \sum_{e \in \delta^-_{\bar{G}}(v)} g(e) - \sum_{e \in \delta^-_G(v)} g(e) + \sum_{e \in \delta^+_{\bar{G}}(v)} g(e) \\
&= \sum_{e \in \delta^+_G(v)} (g(e) - g(\bar{e})) - \sum_{e \in \delta^-_G(v)} (g(e) - g(\bar{e})) \\
&= \sum_{e \in \delta^+_G(v)} \big(f'(e) - f(e)\big) - \sum_{e \in \delta^-_G(v)} \big(f'(e) - f(e)\big) \\
&= \Big(\sum_{e \in \delta^+_G(v)} f'(e) - \sum_{e \in \delta^-_G(v)} f'(e) \Big) - \Big(\sum_{e \in \delta^+_G(v)} f(e) - \sum_{e \in \delta^-_G(v)} f(e) \Big) \\
&= b'(v) - b(v).
\end{aligned}$$

For $e \in E$, we have $g(e) \leq f'(e) - f(e) \leq \mu(e) - f(e) = \mu_f(e)$ and $g(\bar{e}) \leq f(e) - f'(e) \leq f(e) = \mu_f(\bar{e})$. Hence, g is a $(b' - b)$-flow in $(G_f, \mu_f, b' - b)$.
Lastly, it holds that

$$\begin{aligned}
c(g) &= \sum_{e \in E \cup \bar{E}} c(e) \cdot g(e) \\
&= \sum_{e \in E} (c(e) \cdot g(e) + c(\bar{e}) \cdot g(\bar{e})) \\
&= \sum_{e \in E} c(e) \cdot (g(e) - g(\bar{e})) \\
&= \sum_{e \in E} c(e) \cdot (f'(e) - f(e)) \\
&= c(f') - c(f).
\end{aligned}$$

□

Finally, we can translate the fact that flows can be decomposed into paths and cycles (see lecture *Algorithmic Discrete Mathematics*) to circulations.

Corollary 4.9. For every circulation g there exists a family of cycles $\mathcal{K}$ with $|\mathcal{K}| \leq |E|$ and weights $w\colon \mathcal{K} \to \mathbb{R}_{>0}$ with $g(e) = \sum_{K \in \mathcal{K}: e \in K} w(K)$ for all $e \in E$.

Proof. This follows immediately by existence of a path decomposition for s-t-flows, since g is an s-t-flow (of value 0) for arbitrary $s, t \in V$. □

We are now ready to state the first fundamental characterization of minimum-cost b-flows. Note how negative(-cost) cycles play a similar role as augmenting paths for network flows.

Theorem 4.10. A b-flow f has minimum cost if and only if G_f does not contain a negative-cost cycle.

Proof. If there is a negative-cost cycle, we can increase the flow along this cycle and obtain a b-flow of lower cost, hence f cannot have minimum cost.

On the other hand, let f' be a b-flow with smaller cost than f. We consider $g = f' \ominus f$. By Lemma 4.8, g is a circulation in $G \cup \bar{G}$ with $c(g) = c(f') - c(f) < 0$. By Corollary 4.9, we can decompose g into cycles $\mathcal{K}$ with weights $w\colon \mathcal{K} \to \mathbb{R}_{>0}$, such that $g(e) = \sum_{K \in \mathcal{K}: e \in K} w(K)$. From $g(e) = 0$ for all $e \in (E \cup \bar{E}) \setminus G_f$ (Lemma 4.8) and $w > 0$ it follows that every cycle in $\mathcal{K}$ is a subgraph of G_f. We obtain

$$\begin{aligned} c(g) &= \sum_{e \in E \cup \bar{E}} c(e) \cdot g(e) \\ &= \sum_{e \in E \cup \bar{E}} c(e) \cdot \sum_{K \in \mathcal{K}: e \in K} w(K) \\ &= \sum_{K \in \mathcal{K}} w(K) \cdot \sum_{e \in K} c(e) \\ &= \sum_{K \in \mathcal{K}} w(K) \cdot c(K). \end{aligned}$$

From $c(g) < 0$ and $w > 0$ it follows that $\mathcal{K}$ must contain a negative-cost cycle. □

4.1 Minimum-Mean-Cycle Cancelling

Theorem 4.10 immediately suggests an algorithm that repeatedly augments flow along negative cycles until all such cycles are eliminated, very similarly to the FordFulkerson method. As with the FordFulkerson method, it turns out that it is important how we select negative cycles in each step. Much like EdmondsKarp selects a shortest augmenting path in each iteration, it turns out that selecting short (i.e., very negative) cycles first is beneficial. However, we additionally need to account for the number of arcs in the cycle. We do this by selecting the cycle first whose arcs have minimum cost on average. More precisely, we use the following notion to compare cycles.

Definition 4.11. Let f be a b-flow and let $\mathcal{K}_f$ be the set of all cycles in G_f. A cycle K in G_f is a *minimum-mean cycle* with respect to c if $c(K)/|K| = \mathrm{mc}(f) := \min_{K' \in \mathcal{K}_f} c(K')/|K'|$.

Assuming we can find minimum-mean cycles, we can compute a minimum-cost b-flow as follows.

Algorithm: MinMeanCycleCancelling(G, μ, b, c)

input: circulation network (G, μ, b), costs $c\colon E \to \mathbb{R}$
output: minimum cost b-flow

```
f ← bFlow(G, μ, b)
K ← MinMeanCycle(G_f, c)
while K ≠ ∅ and c(K) < 0:
    γ ← min_{e∈K} μ_f(e)
    Augment(K, γ)
    K ← MinMeanCycle(G_f, c)
return f
```

To prove that MinMeanCycleCancelling always terminates and to estimate its running time, we proceed analogously to the analysis of EdmondsKarp. We first establish that the algorithm makes progress in terms of the value of $\mathrm{mc}(f)$.

Lemma 4.12. Let f_i denote the flow and K_i the chosen cycle at the start of each iteration i of MinMeanCycleCancelling. For every two such cycles K_j, K_ℓ with $j < \ell$, we have

$$|\mathrm{mc}(f_j)| \geq \frac{n}{n - 2\min\{1, m_{j\ell}\}} \cdot |\mathrm{mc}(f_\ell)|,$$

where $m_{j\ell} := |\{e \in E \mid e, \bar{e} \in K_j \cup K_\ell\}|$.

Proof. We may assume that there is no cycle K_i, $j < i < \ell$ that contains an arc e with $\bar{e} \in K_\ell$. Otherwise, by induction on $\ell - j$, we immediately have $|\mathrm{mc}(f_j)| \geq |\mathrm{mc}(f_i)| \geq \frac{n}{n-2}|\mathrm{mc}(f_\ell)| \geq \frac{n}{n-2\min\{1,m_{j\ell}\}}|\mathrm{mc}(f_\ell)|$.

Let $\bar{E}_{j\ell} := \{e \in E \cup \bar{E} \mid e, \bar{e} \in K_j \cup K_\ell\}$. We consider the graphs $H_j = (V, E_j) := (V, K_j \setminus \bar{E}_{j\ell})$, $H_\ell = (V, E_\ell) := (V, K_\ell \setminus \bar{E}_{j\ell})$, and $H_{j\ell} = (V, E_{j\ell}) := (V, (K_j \cup K_\ell) \setminus \bar{E}_{j\ell})$. Since the cycles $K_{j+1}, \dots, K_{\ell-1}$ do not contain any reverse arcs to arcs in K_ℓ, we have $K_\ell \setminus \bar{E}_{j\ell} \subseteq G_{f_j}$ and hence $H_j, H_\ell, H_{j\ell} \subseteq G_{f_j}$. Moreover, we have $|\delta^+_{H_j}(v)| + |\delta^+_{H_\ell}(v)| = |\delta^-_{H_j}(v)| + |\delta^-_{H_\ell}(v)|$ for all $v \in V$, which means that $g\colon E_{j\ell} \to \{1, 2\}$ with $g(e) = |\{H \in \{H_j, H_\ell\} \mid e \in H\}|$ defines a circulation in $H_{j\ell}$. By Corollary 4.9 we can decompose g into cycles, each of which are subgraphs of $H_{j\ell} \subseteq G_{f_j}$ and thus have average cost at least $\mathrm{mc}(f_j)$, by definition. Since g encodes all arcs of H_j and H_ℓ, we obtain

$$c(E_j) + c(E_\ell) = c(g) \geq \mathrm{mc}(f_j) \cdot (|E_j| + |E_\ell|)\,. \tag{4.2}$$

Since the costs of opposing arcs cancel out, we further have

$$c(E_j) + c(E_\ell) = c(K_j) + c(K_\ell) = \mathrm{mc}(f_j)|K_j| + \mathrm{mc}(f_\ell)|K_\ell|. \tag{4.3}$$

Finally, it holds that

$$\begin{aligned} |E_j| + |E_\ell| &= |K_j| + |K_\ell| - 2m_{j\ell} \\ &\leq |K_j| + |K_\ell| - \frac{|K_\ell|}{n} 2m_{j\ell} \\ &= |K_j| + \frac{n - 2m_{j\ell}}{n}|K_\ell|. \end{aligned} \tag{4.4}$$

Together with $\mathrm{mc}(f_j) < 0$, this implies

$$\begin{aligned}\mathrm{mc}(f_j)\left(|K_j| + \frac{n-2m_{j\ell}}{n}|K_\ell|\right) &\overset{(4.4)}{\leq} \mathrm{mc}(f_j)\left(|E_j| + |E_\ell|\right)\\ &\overset{(4.2)}{\leq} c(E_j) + c(E_\ell)\\ &\overset{(4.3)}{=} \mathrm{mc}(f_j)|K_j| + \mathrm{mc}(f_\ell)|K_\ell|,\end{aligned}$$

and thus $\mathrm{mc}(f_j) \leq \frac{n}{n-2m_{j\ell}} \cdot \mathrm{mc}(f_\ell)$. This completes the proof, since $\mathrm{mc}(f_j) < 0$ and $\mathrm{mc}(f_\ell) < 0$. □

We can conclude that the value of $\mathrm{mc}(f)$ must approach 0 (from below) significantly in each batch of $n \cdot m$ iterations.

Lemma 4.13. MinMeanCycleCancelling reduces $|\mathrm{mc}(f)|$ by a factor of at least 2 in $n \cdot m$ consecutive iterations.

Proof. Let $K_i, \dots K_{i+m}$ be the augmenting cycles chosen in $m+1$ consecutive iterations, and let $f_i, \dots, f_{i+m}$ be the b-flows at the start of the corresponding iterations. Since every augmentation removes an arc of the residual graph, there are two cycles K_j, K_ℓ, $i \leq j < \ell \leq i+m$ who contain opposing arcs. According to Lemma 4.12, we have

$$\mathrm{mc}(f_i) \leq \mathrm{mc}(f_j) \leq \frac{n}{n-2}\mathrm{mc}(f_\ell) \leq \frac{n}{n-2}\mathrm{mc}(f_{i+m}).$$

Thus, after $n \cdot m$ iterations, the reduction of $|\mathrm{mc}(f)|$ is at least $\left(\frac{n}{n-2}\right)^n = \left(\left(\frac{x}{x-1}\right)^x\right)^2 > \mathrm{e}^2 \approx (2.72)^2 > 2$, where $x := n/2$. □

From this, it is easy to infer a bound on the overall number of iterations. Note, however, that the following bound is not strongly polynomial in n and m, since it depends on the number of bits (and not only the number of numbers) needed to encode the input.

Corollary 4.14. If $c(e) \in \mathbb{Z}$ for all $e \in E$, MinMeanCycleCancelling computes a min-cost b-flow in $\mathcal{O}(mn(\log n + \log |c_{\min}|))$ iterations, where $c_{\min} := \min_{e \in E} c(e)$.

Proof. At the beginning, we have $|\mathrm{mc}(f)| \leq |c_{\min}|$. By Lemma 4.13, the algorithm needs $\mathcal{O}(nm \log |c_{\min}|)$ iterations until $\mathrm{mc}(f) > -1$, and, hence, $\mathcal{O}(nm(\log n + \log |c_{\min}|))$ iterations until $\mathrm{mc}(f) > -1/n$. Since all costs in the residual graph remain integral, and augmenting cycles have at most n arcs, we have $\mathrm{mc}(f) \notin (-1/n, 0)$. Consequently, we have $\mathrm{mc}((f) \geq 0)$ after $\mathcal{O}(nm(\log n + \log |c_{\min}|))$ iterations and the algorithm terminates. By Theorem 4.10, the resulting b-flow is optimal. □

We now give a more careful estimation that yields a polynomial bound.

Theorem 4.15. MinMeanCycleCancelling computes a min-cost b-flow in $\mathcal{O}(m^2 n \log n)$ iterations.

Proof. Correctness follows from Theorem 4.10. We show that after each additional $mn(\lceil \log_2 n \rceil + 1)$ iterations an additional arc will remain unchanged in future iterations. Let f, f' be the flows before and after such a sequence of iterations, and let K_f be the augmenting cycle chosen for f. The proof proceeds as follows:

(i) We show that the costs of arcs in $G_{f'}$ are bounded from below.

(ii) We show that there is a very cheap arc $e_0 \in G_f \setminus G_{f'}$, so that $\bar{e}_0$ is very expensive in $G_{f'}$.

(iii) We claim that the residual capacity of e_0 remains 0 in future iterations of the algorithm. By Lemma 4.12, to show this, it suffices to show that, for all b-flows f'' with $e_0 \in G_{f''}$, it holds that $\mathrm{mc}(f'') < \mathrm{mc}(f')$.

(iv) We consider the decomposition of $g := f'' \ominus f'$ into a set of cycles $\mathcal{K}$ and show that there must be a cycle $K_0 \in \mathcal{K}$ with $\bar{e}_0 \in K_0 \subseteq G_{f'}$.

(v) We show that, because of (i) and (ii), K_0 is very expensive.

(vi) It follows that $\bar{K}_0 = \{\bar{e} \mid e \in K_0\} \subseteq G_{f''}$ is sufficiently cheap in $G_{f''}$ to ensure $\mathrm{mc}(f'') < \mathrm{mc}(f')$.

We now address these steps in detail.

(i) By Lemma 4.13, we have

$$\mathrm{mc}(f) \leq 2^{\lceil \log_2 n \rceil + 1} \mathrm{mc}(f') \leq 2n \,\mathrm{mc}(f'), \tag{4.5}$$

since $\mathrm{mc}(f), \mathrm{mc}(f') < 0$.

We define $c'(e) = c(e) + |\mathrm{mc}(f')| = c(e) - \mathrm{mc}(f')$ for all $e \in G_{f'}$. With this, c' does not induce any negative cycles in $G_{f'}$. By Proposition 2.24 there exists a potential $\pi \colon V \to \mathbb{R}$ with $0 \leq c'_\pi(e) = c_\pi(e) - \mathrm{mc}(f')$ — recall that $c_\pi(u,v) := \pi(u) - \pi(v) + c(u,v)$. Hence,

$$c_\pi(e) \geq \mathrm{mc}(f'), \tag{4.6}$$

for all $e \in G_{f'}$.

(ii) We obtain

$$\sum_{e \in K_f} c_\pi(e) = \sum_{e \in K_f} c(e) = \mathrm{mc}(f) \cdot |K_f| \overset{(4.5)}{\leq} 2n \,\mathrm{mc}(f') |K_f|.$$

This means we can choose $e_0 \in K_f \subseteq G_f$ with

$$c_\pi(e_0) \leq 2n \,\mathrm{mc}(f') < \mathrm{mc}(f'). \tag{4.7}$$

By (4.6) we have $e_0 \notin G_{f'}$.

(iii) We claim that the flow along e_0 remains unchanged in future iterations. Let f'' be a b-flow with $e_0 \in G_{f''}$. We show that $\mathrm{mc}(f'') < \mathrm{mc}(f')$. Since $\mathrm{mc}(f)$ is growing monotonically during the course of the algorithm (Lemma 4.12), this proves the claim.

(iv) Let $g := f'' \ominus f'$. By Lemma 4.8 and Corollary 4.9, we may consider the decomposition of g into a set of cycles $\mathcal{K}$ in $G_{f'}$.

We show that $g(e) = \max\{0, \mu_{f''}(\bar{e}) - \mu_{f'}(\bar{e})\}$ for all $e \in E \cup \bar{E}$: For $e \in E$ we have $g(e) = \max\{0, f''(e) - f'(e)\} = \max\{0, \mu_{f''}(\bar{e}) - \mu_{f'}(\bar{e})\}$. For $e \in \bar{E}$ we have $g(e) = \max\{0, f'(\bar{e}) - f''(\bar{e})\} = \max\{0, (\mu(\bar{e}) - \mu_{f'}(\bar{e})) - (\mu(\bar{e}) - \mu_{f''}(\bar{e}))\} = \max\{0, \mu_{f''}(\bar{e}) - \mu_{f'}(\bar{e})\}$. In particular, it follows from $e_0 \in G_{f''} \setminus G_{f'}$ that $\mu_{f'}(e_0) = 0 < \mu_{f''}(e_0)$, hence $g(\bar{e}_0) = \mu_{f''}(e_0) > 0$. We can therefore find $K_0 \in \mathcal{K}$ with $\bar{e}_0 \in K_0$. By Lemma 4.8, we have $g(e) = 0$ for $e \in (E \cup \bar{E}) \setminus G_{f'}$, which implies $K_0 \subseteq G_{f'}$.

(v) We obtain (using $\mathrm{mc}(f') < 0$, otherwise, the statement is trivial)

$$\begin{aligned}
c(K_0) &= \sum_{e \in K_0} c_\pi(e) \\
&= c_\pi(\bar{e}_0) + \sum_{e \in K_0 \setminus \{\bar{e}_0\}} c_\pi(e) \\
&\overset{(4.6)}{\geq} -c_\pi(e_0) + \sum_{e \in K_0 \setminus \{\bar{e}_0\}} \mathrm{mc}(f') \\
&\overset{(4.7)}{\geq} -2n \cdot \mathrm{mc}(f') + (n-1) \cdot \mathrm{mc}(f') \\
&> -n \cdot \mathrm{mc}(f').
\end{aligned} \tag{4.8}$$

(vi) Let $\bar{K}_0$ be the cycle consisting of the opposing arcs to the arcs in K_0. For all $e \in K_0$, it holds that $0 < g(e) = \max\{0, \mu_{f''}(\bar{e}) - \mu_{f'}(\bar{e})\}$, hence $\mu_{f''}(\bar{e}) > 0$. It follows that $\bar{K}_0 \subseteq G_{f''}$. But $c(\bar{K}_0) = -c(K_0) < n \cdot \mathrm{mc}(f')$ by (4.8). This completes the proof (by (iii)), since

$$\mathrm{mc}(f'') \leq \frac{c(\bar{K}_0)}{|\bar{K}_0|} < \frac{n\,\mathrm{mc}(f')}{|\bar{K}_0|} \overset{\mathrm{mc}(f')<0}{\leq} \mathrm{mc}(f'). \qquad \square$$

4.1.1 Finding a minimum-mean cycle

For a bound on the overall running time of MinMeanCycleCancelling we first need to describe how to find a minimum-mean cycle in each iteration. In other words, we consider the following problem.

Minimum-Mean Cycle Problem	
input:	graph $G = (V, E)$; arc costs $c\colon E \to \mathbb{R}$
problem:	find a minimum-mean cycle K in G with respect to c

We will need the following notation.

Definition 4.16. We let $d_c^{(=k)}(u, v)$ denote the minimum cost among all walks from u to v with *exactly* k arcs.

We first characterize the cases in which the minimum-mean cycle has cost exactly 0 and show how to find such a cycle.

Lemma 4.17. Let c not induce negative cycles in G and let all vertices be reachable from $s \in V$. Then G has a cycle of cost 0 if and only if

$$\min_{v \in V} \max_{0 \leq k < n} (d_c^{(=n)}(s, v) - d_c^{(=k)}(s, v)) = 0.$$

Let v be the vertex for which the minimum is attained. Then, every cycle on a shortest walk with n arcs from s to v has cost 0.

Proof. It holds that $d_c^{(=n)}(s, v) \geq d_c(s, v) = \min_{0 \leq k < n} d_c^{(k)}(s, v)$, since c does not induce negative cycles in G. Hence, we have $\max_{0 \leq k < n}(d_c^{(=n)}(s, v) - d_c^{(=k)}(s, v)) \geq 0$ for all vertices $v \in V$. Now let K be a cycle of cost 0 and $u \in K$ be a vertex on K. We consider a shortest s-u-path and extend it by adding n copies of K. The resulting walk is a shortest s-u-walk, thus, by optimum substructure (see lecture *Algorithmic Discrete Mathematics*), the prefix W with exactly n arcs must be a shortest walk from s to some vertex $v \in K$. By Proposition 2.8, this implies $d_c^{(=n)}(s, v) = c(W) = d_c(s, v) = \min_{0 \leq k < n} d_c^{(=k)}(s, v)$ and thus $\max_{0 \leq k < n}(d_c^{(=n)}(s, v) - d_c^{(=k)}(s, v)) = 0$.

Conversely, if there is a vertex $v \in V$ with $\max_{0 \leq k < n}(d_c^{(=n)}(s, v) - d_c^{(=k)}(s, v)) = 0$, then we have $d_c^{(=n)}(s, v) \leq d_c^{(=k)}(s, v)$ for all $k \in \{0, \ldots, n-1\}$, and hence $d_c^{(=n)}(s, v) = d_c(s, v)$. It follows that every s-v-walk with n arcs and length $d_c^{(=n)}(s, v)$ is a shortest s-v-walk. Thus any cycle along this walk must have cost 0 (note that every walk with n arcs must contain a cycle). $\square$

We can now apply Lemma 4.17 to the general case by shifting costs so that the minimum cycle has cost 0.

Lemma 4.18. Let all vertices of G be reachable from $s \in V$. Then the minimum mean cost mc among all cycles in G satisfies

$$\mathrm{mc} = \min_{v \in V} \max_{0 \le k < n} \frac{d_c^{(=n)}(s,v) - d_c^{(=k)}(s,v)}{n-k}.$$

Let v be the vertex for which the minimum is attained. Then, every cycle on a shortest walk with n arcs from s to v has minimum mean cost.

Proof. Consider the cost function given by $c'(e) := c(e) - \mathrm{mc}$ for all $e \in E$. Compared to c, with respect to c', the mean cost of every walk is shifted by exactly mc. It follows that c' does not induce negative cycles and cyles of cost 0 (with respect to c') correspond to cycles of minimum mean value with respect to c. In particular, c' induces a cycle of cost 0. By Lemma 4.17, we have

$$\begin{aligned} 0 &= \min_{v \in V} \max_{0 \le k < n} \frac{d_{c'}^{(=n)}(s,v) - d_{c'}^{(=k)}(s,v)}{n-k} \\ &= \min_{v \in V} \max_{0 \le k < n} \frac{d_c^{(=n)}(s,v) - n \cdot \mathrm{mc} - (d_c^{(=k)}(s,v) - k \cdot \mathrm{mc})}{n-k} \\ &= \min_{v \in V} \max_{0 \le k < n} \frac{d_c^{(=n)}(s,v) - d_c^{(=k)}(s,v)}{n-k} - \mathrm{mc}. \end{aligned}$$

The second part of the claim follows by Lemma 4.17, since cycles of cost 0 translate to minimum mean cycles. □

With this, we efficiently find a minimum-mean cycle.

Algorithm: MinMeanCycle(G, c)

input: graph $G = (V, E)$, costs $c\colon E \to \mathbb{R}$
output: cycle of minimum mean cost

$G' \leftarrow (V \cup \{s\}, E \cup (\{s\} \times V))$

$$c'(u,v) \leftarrow \begin{cases} 0, & \text{for } u = s, v \in V, \\ c(u,v), & \text{for } u \in V, v \in V. \end{cases}$$

$(d_{sv}^{(=k)})_{k \in \{1,\dots,n\}} \leftarrow$ BellmanFord(-Moore)$'(G', c')$

$v \in \arg\min_{u \in V} \max_{0 \le k < n} (d_{su}^{(=n)} - d_{su}^{(=k)})/(n-k)$

$W \leftarrow$ Backtrack$(d_{sv}^{(=n)}, s, v)$ (shortest s-v-walk with n arcs)

return any cycle in W (or $\emptyset$ if none exists)

Theorem 4.19. We can find a cycle of minimum-mean cost in time $\mathcal{O}(nm)$.

Proof. We can introduce a vertex s and connect it to all other vertices, and use an algorithm very similar to BellmanFord(-Moore) to compute $d_c^{(=k)}(s,v)$ for all $0 \le k \le n$ and $v \in V$. Lemma 4.18 gives us a way to obtain a cycle of minimum-mean cost (see algorithm MinMeanCycle). □

Using this together with Theorem 4.15 yields the following running time.

Theorem 4.20. MinMeanCycleCancelling can be implemented to compute a min-cost b-flow in time $\mathcal{O}(m^3 n^2 \log n)$.

4.2 Successive Shortest Paths

In the previous section, we started with a b-flow that may be too expensive and repeatedly decreased its cost by augmenting along negative cycles in the residual graph. A different approach is to start with some b'-flow that is guaranteed to have minimum cost among all b'-flows, but may have $b' \neq b$, and to then repeatedly send flows along paths in order to transition to a b-flow. Such a transition is possible by the following insight.

Lemma 4.21. Let f be a minimum-cost b-flow and f' be a b'-flow that results from augmenting f along a shortest (with respect to c) path P between some vertices s and t in G_f. Then, f' is a minimum-cost b'-flow.

Proof. For the sake of contradiction, assume that f' does not have minimum cost. Then, by Theorem 4.10, c induces a negative cycle K in $G_{f'}$.

Let $\bar{E}_{PK} := \{e \in E \cup \bar{E} \mid e, \bar{e} \in P \cup K\}$. We consider the graphs $H_P = (V, P \setminus \bar{E}_{PK})$, $H_K := (V, K \setminus \bar{E}_{PK})$, and $H_{PK} = (V, E_{PK}) := (V, (P \cup K) \setminus \bar{E}_{PK})$. No arc $e \in G_{f'} \setminus G_f$ can be contained in H_{PK}, since $\bar{e} \in P$. Hence, $H_{PK} \subseteq G_f$. Therefore, as f has minimum cost, by Theorem 4.10, H_{PK} cannot contain negative cycles. We have $|\delta^+_{H_P}(v)|+|\delta^+_{H_K}(v)| = |\delta^-_{H_P}(v)|+|\delta^-_{H_K}(v)|$ for all $v \in V \setminus \{s,t\}$, and $|\delta^+_{H_P}(s)|+|\delta^+_{H_K}(s)| = |\delta^-_{H_P}(s)|+|\delta^-_{H_K}(s)|+1$. This means that $g\colon E_{PK} \to \{1,2\}$ with $g(e) = |\{H \in \{H_P, H_K\} \mid e \in H\}|$ defines an s-t-flow of value $|g| = 1$ in $H_{PK} \subseteq G_f$. We can therefore decompose g into an s-t-path P' and a set of cycles (see lecture *Algorithmic Discrete Mathematics*). Since these cycles cannot have negative costs, we have

$$c(P') \leq c(g) = c(H_P) + c(H_K) = c(P) + c(K) < c(P),$$

which contradicts our choice of P. □

We show that we can apply these transitions to reduce the balances. Note that, by definition of circulation networks, $\sum_{v \in V} b(v) = 0$, hence $b \neq 0$ implies existence of $u \in V$ with $b(u) > 0$.

Lemma 4.22. If a b-flow exists in (G, μ, b) for $b \neq 0$, then for every $u \in V$ with $b(u) > 0$ there is $v \in V$ with $b(v) < 0$ reachable from u in G.

Proof. Let R be the set of vertices reachable from u. By Proposition 4.4, we have

$$0 = \sum_{e \in \delta^+(R)} \mu(e) \geq \sum_{v \in R} b(v) > \sum_{v \in R \setminus \{u\}} b(v),$$

hence there must be $v \in R$ with $b(v) < 0$. □

Lemmas 4.21 and 4.22 immediately suggest another algorithm similar to EdmondsKarp. To start our algorithm, we need some minimum-cost b'-flow for arbitrary b'. By Theorem 4.10, such a flow can be obtained by eliminating all negative cycles. This can be accomplished easily, by fully saturating all arcs of negative costs.

Algorithm: SUCCESSIVESHORTESTPATH(G, μ, b, c)

input: circ. network (G, μ, b), costs $c\colon E \to \mathbb{R}$
output: minimum cost b-flow

$b, f \leftarrow$ MINCOSTSTARTFLOW(G, μ, b, c)
while $b \neq 0$**:**
  if $\exists$ *s-t-path in* G_f **with** $b(s) > 0$ **and** $b(t) < 0$**:**
    $P \leftarrow$ SHORTESTPATH(G_f, c, s, t)
    PUSH(P)
  else
    fail
return f

Algorithm: MINCOSTSTARTFLOW(G, μ, b, c)

input: circ. network (G, μ, b), costs $c\colon E \to \mathbb{R}$
output: balances $b'\colon V \to \mathbb{R}$,
min-cost $(b - b')$-flow f

$f \leftarrow 0$
for $e \in E$ **with** $c(e) < 0$**:**
  $f(e) \leftarrow \mu(e)$
for $v \in V$**:**
  $b'(v) \leftarrow b(v) + \mathrm{ex}_f(v)$
return b', f

Algorithm: PUSH(P)

$\gamma \leftarrow \min\{\min_{e \in P} \mu_f(e), b(s), -b(t)\}$
$b(s) \leftarrow b(s) - \gamma$, $b(t) \leftarrow b(t) + \gamma$
AUGMENT(P, γ)

We show that SUCCESSIVESHORTESTPATH is correct.

Theorem 4.23. SUCCESSIVESHORTESTPATH only fails if no b-flow exists. If it terminates without failing, it returns a minimum-cost b-flow.

Proof. Let $b^\star$ be the original value of b at the start of the algorithm, and let b_0 be the value after computation of the starting flow f_0. Since f_0 saturates all arcs of negative costs and does not use other arcs, f_0 is a minimum-cost $(b^\star - b_0)$-flow. By Lemma 4.21, f remains a minimum-cost $(b^\star - b)$-flow. Since the algorithm only properly terminates when f is a $b^\star$-flow, it follows that SUCCESSIVESHORTESTPATH returns a minimum-cost $b^\star$-flow if it terminates without failing.

Now assume thatSUCCESSIVESHORTESTPATH fails, and let f be the $(b^\star - b)$-flow at the time it does. The algorithm fails, since there is no more s-t-path with $b(s) > 0$ and $b(t) < 0$ in G_f. By Lemma 4.22, no b-flow exists in (G_f, μ_f, b) at this point. This means that no $b^\star$-flow $f^\star$ can exist in $(G, \mu, b^\star)$, since, otherwise, $f^\star \ominus f$ would be a b-flow in (G_f, μ_f, b) by Lemma 4.8. □

For integral balances and capacities, we can bound the total number of iterations of SUCCESSIVESHORTESTPATH. In case of irrational data, similarly to FORDFULKERSON, the algorithm may not terminate.

Corollary 4.24. If all balances and all capacities are integral, SUCCESSIVESHORTESTPATH computes a minimum-cost b-flow in $\mathcal{O}(B)$ augmentations, where $B := \sum_{v \in V} |b(v)| + \sum_{e \in E} \mu(e)$.

Proof. In MINCOSTSTARTFLOW, $\sum_{v \in V} |b(v)|$ may increase by at most $2 \sum_{e \in E} \mu(e)$. Afterwards, since balances and capacities are integral, in every augmentation from some vertex $s \in V$ to some vertex $t \in V$, the absolute values of $b(s)$ and $b(t)$ are reduced by at least one, while their signs remain unchanged. □

Observe that the above bound is not polynomial in n and m, since capacities and balances may have exponential values. The bound is tight, i.e., there are instances on which SUCCESSIVESHORTESTPATH requires exponentially many iterations (see Figures 4.1 and 4.2) (note that the network N_i has $2(i + 1)$ vertices).

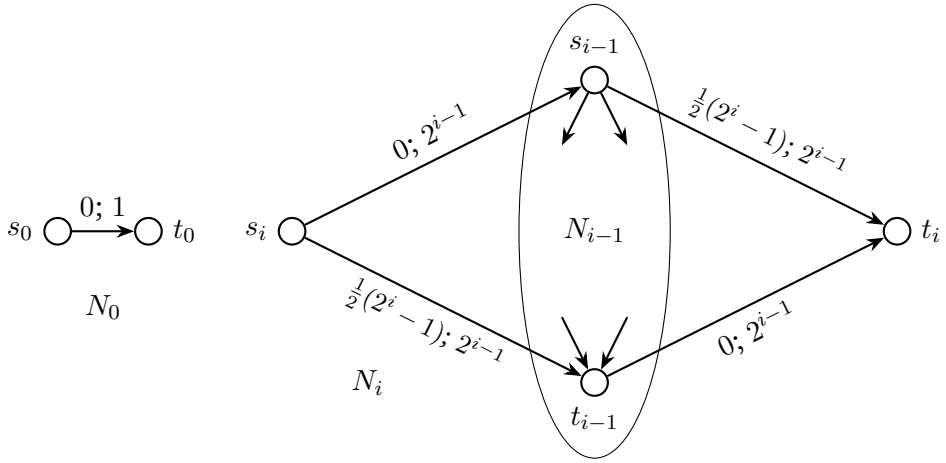

Figure 4.1: Recursive construction of the network N_i with $2(i+1)$ vertices on which SUCCESSIVESHORTESTPATH needs an exponential number of iterations. Each arc is labeled by its cost and its capacity in this order, and we set $b(s_i) = 2^i$ and $b(t_i) = -2^i$. The cost of the shortest s_i-t_i-path in iteration $j \in \{1, \dots, 2^i\}$ is exactly $j - 1$.

Proposition 4.25. SUCCESSIVESHORTESTPATH needs $2^i = \Theta(B)$ augmentations to compute a minimum-cost flow in the network N_i of Figure 4.1 for any $i \in \mathbb{N}$.

In order to bound the running time of SUCCESSIVESHORTESTPATH beyond bounding the number of iterations, we need to specify how shortest paths are computed. As shown in Section 2.2.1, we can use DIJKSTRA, provided that we have a vertex potential at hand. We cannot recompute such a potential in every iteration,

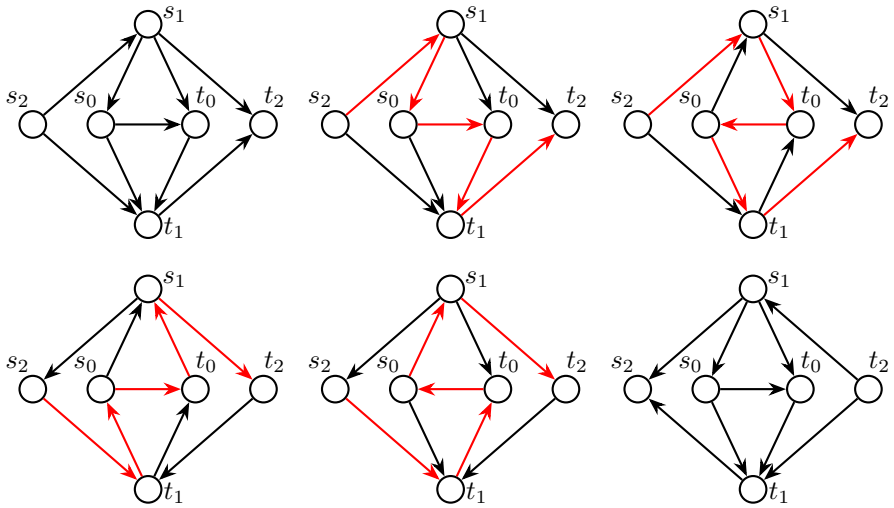

Figure 4.2: Illustration of the iterations performed by SUCCESSIVESHORTESTPATH on the network N_2 of Figure 4.1. The shortest path in each iteration is marked in red, and arcs are oriented in the direction in which they are used next. Note that, after $2^i = 4$ iterations, the configuration of the residual graph is the same as in the beginning if we switch the roles of s_2 and t_2. This allows to recursively embed the network into N_3 and so on.

since this would dominate the running time of DIJKSTRA. However, we can compute a vertex potential once and update it cheaply in every iteration.

Algorithm: SUCCESSIVESHORTESTPATH(G, μ, b, c) (potential)

input: circ. network $(G = (V, E), \mu, b)$, costs $c\colon E \to \mathbb{R}$
output: minimum cost b-flow

```
b, f ← MinCostStartFlow(G, μ, c)
π ← Potential(G_f, c)                          (or just π ← 0)
while b ≠ 0:
    if ∃ s-t-path in G_f with b(s) > 0 and b(t) < 0:
        (d_sv)_{v∈V} ← Dijkstra(G_f, c_π, s)
        P ← Backtrack((d_sv)_{v∈V}, s, t)
        Push(P)
        for v ∈ V:
            π(v) ← π(v) + min{d_sv, d_st}
    else
        fail
return f
```

We now show that vertex potentials are maintained correctly.

Theorem 4.26. SUCCESSIVESHORTESTPATH with potential only fails if no b-flow exists and otherwise returns a minimum-cost b-flow.

Proof. By Theorem 4.23, it suffices to show the invariant that π is always a vertex potential for G_f and that we choose a shortest augmenting path in G_f with respect to c in every step. By correctness of POTENTIAL (Proposition 2.19), π is a potential at the start of the algorithm. Let f_i be the b-flow at the start of iteration i, let π_i be the potential at the start of iteration i, and let P_i be the s-t-path chosen in this iteration. By correctness of DIJKSTRA, and since $c_{\pi_i} \geq 0$, we choose P_i as a shortest s-t-path with respect to c_π. Thus, P_i is a shortest path with respect to c as well (Proposition 2.18).

Assume that π_i is a vertex potential for G_{f_i} with respect to c at the start of iteration i. In iteration i, DIJKSTRA computes the values $(d_{sv})_{v\in V}$ such that $d_{sv} = d_{c_{\pi_i}}(s, v)$. We have to show that π_{i+1} is a potential, i.e., that $c_{\pi_{i+1}}(u, v) = c(u, v) + \pi_{i+1}(u) - \pi_{i+1}(v) \geq 0$ for all $(u, v) \in G_{f_{i+1}}$. First observe that

$$\begin{aligned} c_{\pi_{i+1}}(u, v) &= c(u, v) + \pi_{i+1}(u) - \pi_{i+1}(v) \\ &= c(u, v) + \pi_i(u) + \min\{d_{su}, d_{st}\} - \pi_i(v) - \min\{d_{sv}, d_{st}\} \\ &= c_{\pi_i}(u, v) + \min\{d_{su}, d_{st}\} - \min\{d_{sv}, d_{st}\}. \end{aligned} \tag{4.9}$$

Consider arcs $(u, v) \in G_{f_i} \cap G_{f_{i+1}}$. If $\min\{d_{su}, d_{st}\} = d_{st}$, then $\min\{d_{su}, d_{st}\} - \min\{d_{sv}, d_{st}\} \geq 0$, and $c_{\pi_{i+1}} \geq 0$ follows from $c_{\pi_i}(u, v) \geq 0$ by our invariant, since $(u, v) \in G_{f_i}$. If $\min\{d_{su}, d_{st}\} = d_{su}$, we obtain $c_{\pi_{i+1}}(u, v) \geq (c_{\pi_i}(u, v) + d_{c_{\pi_i}}(s, u)) - d_{c_{\pi_i}}(s, v) \geq 0$, since $(u, v) \in G_{f_i}$.

Now consider arcs $(u, v) \in G_{f_{i+1}} \setminus G_{f_i}$. We have $(v, u) \in P_i$, since only the flow along P_i was changed in iteration i. Since $c_{\pi_i} \geq 0$ and P_i is a shortest s-t-path, by optimum substructure (see lecture *Algorithmic Discrete Mathematics*), we have $\min\{d_{c_{\pi_i}}(s, u), d_{c_{\pi_i}}(s, t)\} = d_{c_{\pi_i}}(s, u)$ and $\min\{d_{c_{\pi_i}}(s, v), d_{c_{\pi_i}}(s, t)\} = d_{c_{\pi_i}}(s, v)$. Also, we have $d_{su} = d_{c_{\pi_i}}(s, u) = d_{c_{\pi_i}}(s, v) + c_{\pi_i}(v, u) = d_{sv} - c_{\pi_i}(u, v)$, and thus $c_{\pi_{i+1}}(u, v) = 0$ by (4.9). □

This allows us to bound the overall running time. Note that, exponential worst-case running time notwithstanding, SuccessiveShortestPath is the best known algorithm for the case $B = \mathcal{O}(n)$ and $\mu \in \mathbb{N}$, $b \in \mathbb{Z}$.

Corollary 4.27. If all balances and all capacities are integral, SuccessiveShortestPath with potential computes a minimum-cost b-flow in time $\mathcal{O}(nm + B(m + n \log n))$, where $B := \sum_{v \in V} |b(v)| + \sum_{e \in E} \mu(e)$.

Proof. This follows from Theorem 4.26 and Corollary 4.24, together with the running times of Potential (Proposition 2.19) and Dijkstra (Theorem 2.2). Note that, by Lemma 4.22, we can find s and t in each iteration simply by running DFS or BFS starting at any vertex s with $b(s) > 0$ (alternatively, t can be found during the execution of Dijkstra). □

Remark 4.28. Note that we can slightly improve the running time of SuccessiveShortestPath by initializing the potential to 0 instead of invoking Potential. We can do this, since we eliminate all arcs of negative costs from the initial residual graph. However, this approach no longer works for the algorithm of the next section.

4.2.1 Capacity Scaling

Intuitively, in the example of Figure 4.1, the problem is that SuccessiveShortestPath augments the flow by very small amounts in each step, even though paths of larger capacity would be available. It turns out that we can significantly improve performance by prioritizing augmenting paths that allow for large changes in flow values. We need to balance this carefully with trying to augment along cheap paths first. This can be accomplished by operating in phases, where each phase only considers paths that change the flow by at least some threshold value. Once a phase is complete, we lower the threshold and repeat the process. This general method is called *capacity scaling*.

For convenience, from now on, we assume that $\mu = \infty$, i.e., capacities are unbounded. We can transform every instance to have this property, as shown in the following lemma. Note, however, that this transformation affects the running time of all algorithms, since we introduce an additional vertex for each arc.

Lemma 4.29. Every instance $(G = (V, E), \mu, b, c)$ of the minimum-cost flow problem can be transformed into an equivalent, acyclic instance $(G' = (V \cup E, E'), \infty, b', c')$ with $n + m$ vertices and $2m$ arcs.

Proof. For every arc $e = (u, v) \in E$, we introduce arcs $(e, u) \in E'$ and $(e, v) \in E'$, with $c'(e, u) = 0$ and $c'(e, v) = c(e)$. We further define the balance by $b'(e) = \mu(e)$ for all $e \in E$ and $b'(v) = b(v) - \sum_{e \in \delta^+_G(v)} \mu(e)$ for all $v \in V$.

Let f be a b-flow in (G, μ, b, c). For $e = (u, v) \in E$, we set $f'(e, v) = f(e)$ and $f'(e, u) = \mu(e) - f(e)$. Then, f' is a b'-flow in (G', ∞, b', c') of cost $c'(f') = c(f)$. Conversely, let f' be a b'-flow in (G', ∞, b', c'). We set $f(u, v) = f'((u, v), v)$ for all $(u, v) \in E$ and obtain a b-flow f with $c(f) = c'(f')$. □

Eliminating capacities allows us to cleanly state a capacity-scaling algorithm, where the scaling aspect only reflects in our selection of source and sink in each step.

Algorithm: CapacityScaling(G, b, c)

input: circ. network $(G = (V, E), \infty, b\colon V \to \mathbb{Z})$,
costs $c\colon E \to \mathbb{R}$

output: minimum cost b-flow

```
f ← 0
γ ← 2^⌊log₂(max_{v∈V} b(v))⌋
while b ≠ 0:
    if γ < 1:
        fail
    while ∃ s-t-path in G_f with b(s) ≥ γ and b(t) ≤ −γ:
        P ← ShortestPath(G_f, c, s, t)
        Push(P, γ)
    γ ← γ/2
return f
```

Algorithm: Push(P, γ)

```
if γ < 0:
    Push(P̄, −γ)
else
    b(s) ← b(s) − γ
    b(t) ← b(t) + γ
    Augment(P, γ)
```

We now show that this algorithm is indeed still correct and achieves a better (weakly polynomial) running time than SuccessiveShortestPath.

Theorem 4.30. If $\mu = \infty$, $b(v) \in \mathbb{Z}$ for all $v \in V$, and c does not induce negative cycles, CapacityScaling computes a minimum-cost b-flow in time $\mathcal{O}(n(m + n \log n) \log b_{\max})$, where $b_{\max} := \max_{v \in V} b(v)$.

Proof. First observe that all residual capacities are either ∞ or integer multiples of γ throughout the algorithm, hence all Push operations are well-defined. Since b is integral throughout, the algorithm only fails if there exists no s-t-path with $b(s) > 0$ and $b(t) < 0$. As in the proof of Theorem 4.23, this implies that no solution exists by Lemmas 4.8 and 4.22. With this, correctness follows by Lemma 4.21 together with the fact that the initial flow ($f = 0$) is a minimum-cost circulation (i.e., 0-flow) by Theorem 4.10, since c does not induce negative cycles.

We divide the execution of CapacityScaling into maximal phases in which γ is constant and we show that the algorithm takes at most n iterations in each phase. To this end, consider f, b at the start of a phase associated with any fixed value of γ. We define

$$\begin{aligned} S &:= \{v \in V \mid b(v) \geq \gamma\}, \\ S^+ &:= \{v \in V \mid b(v) \geq 2\gamma\}, \\ T &:= \{v \in V \mid b(v) \leq -\gamma\}, \\ T^+ &:= \{v \in V \mid b(v) \leq -2\gamma\}, \end{aligned}$$

and let R be the set of all vertices that are reachable from an element of S^+ in G_f. We have $S^+ \subseteq R$ by definition, and $R \cap T^+ = \emptyset$, as the previous phase ended because no paths remain from S^+ to T^+.

Let $((s_i, t_i))_{i=1,\dots,k}$ be the pairs (s,t) chosen during the current phase. Since every augmentation is by the same amount γ, every augmentation with $s_i \in R$ and $t_i \notin R$ reduces the capacity of the cut $\delta^+_{G \cup \bar{G}}(R)$ by γ. Since $\delta^+_{G_f}(R) = \emptyset$, this means that for every augmentation across $\delta^+_{G \cup \bar{G}}(R)$ we first need to augment at least once across $\delta^-_{G \cup \bar{G}}(R)$. We obtain

$$|\{i \,|\, s_i \in R \wedge t_i \notin R\}| \leq |\{i \,|\, s_i \notin R \wedge t_i \in R\}|\,.$$

The total number k of iterations in the phase is therefore bounded by

$$\begin{aligned} k &= |\{i \,|\, s_i \in R \wedge t_i \in R\}| + |\{i \,|\, s_i \in R \wedge t_i \notin R\}| + |\{i \,|\, s_i \notin R \wedge t_i \in R\}| + |\{i \,|\, s_i \notin R \wedge t_i \notin R\}| \\ &\leq |\{i \,|\, s_i \in R \wedge t_i \in R\}| + 2 \cdot |\{i \,|\, s_i \notin R \wedge t_i \in R\}| + |\{i \,|\, s_i \notin R \wedge t_i \notin R\}| \\ &= |\{i \,|\, t_i \in R\}| + |\{i \,|\, s_i \notin R\}|\,. \end{aligned}$$

But since $R \cap T^+ = \emptyset$, we have $b(t) > -2\gamma$ for every sink $t \in R$, which means that every sink in R can be used at most once. This implies $|\{i \,|\, t_i \in R\}| \leq |T \cap R|$. Analogously, every source $s \notin R$ can only be used once, since $S^+ \setminus R = \emptyset$. This implies $|\{i \,|\, s_i \notin R\}| \leq |S \setminus R|$. We obtain

$$k \leq |T \cap R| + |S \setminus R| \leq |T| + |S| \leq n.$$

This establishes that the algorithm needs at most n iterations per phase. Obviously, there are $\mathcal{O}(\log b_{\max})$ phases overall, and every phase can be implemented as in SuccessiveShortestPath with potential. This yields the claimed running time. □

4.3 Orlin's Algorithm

In CapacityScaling, we reduce γ by factor 2 after each phase. Assuming, for the moment, that we have p augmentations per phase, we can conclude that the total magnitude of all augmentations after a γ-phase is $p \sum_{i \geq 1} 2^{-i} \gamma = p\gamma$. This means that all arcs $\bar{e} \in \bar{E}$ for which $f(e) > p\gamma$ at the end of the γ-phase remain in the residual graph from now on (and so does e, because of $\mu = \infty$). This insight allows us to improve efficiency by treating connected components formed by such arcs $e, \bar{e}$ as single vertices. Importantly, we know that there cannot be cheaper paths connecting the endpoints of $e, \bar{e}$, since this would induce negative cycles. We can therefore discard all other arcs between the components of the two endpoints of $e, \bar{e}$ before merging these components. This gradual simplification of the graph significantly improves the running time of the algorithm, as we will see. We first formalize the notion of merging components (cf. Figure 4.3).

Definition 4.31. Let $\{u, v\} \in E$ be an edge of an undirected graph $G = (V, E)$, and let $c \colon E \to \mathbb{R}$ be edge costs. The result of *contracting* $\{u, v\}$ in G, c is $G' = (V', E')$, c' with $V' := (V \cup \{x_{uv}\}) \setminus \{u, v\}$ and $E' := (E \setminus \delta_G(\{u, v\})) \cup \{\{x_{uv}, w\} \,|\, w \in \Gamma_G(\{u, v\})\}$ and $c'(e) = c(e)$ for $e \in E' \cap E$ and $c'(x_{uv}, w) := \min\{c(u, w), c(v, w)\}$ for $\{x_{uv}, w\} \in E' \setminus E$. Contracting arcs of directed graphs is defined analogously.

We do not explicitly contract arcs, but rather keep track of the forest F of all arcs that have been contracted and the subgraph $H \subseteq G$ in which all other arcs within components have been removed. To further simplify the algorithm, we refine a *representative* r for each component in F and we maintain $r_v = r$ for all vertices v of this component. When merging components we ensure that $b(v) = 0$ if v is not the representative of the new component, by augmenting flow once along the unique path between the two representatives. Finally, we further improve the algorithm by reducing γ by more than factor 2 whenever possible. With these modifications we obtain Orlin's algorithm. Note that we assume $n \geq 2$ in the following.

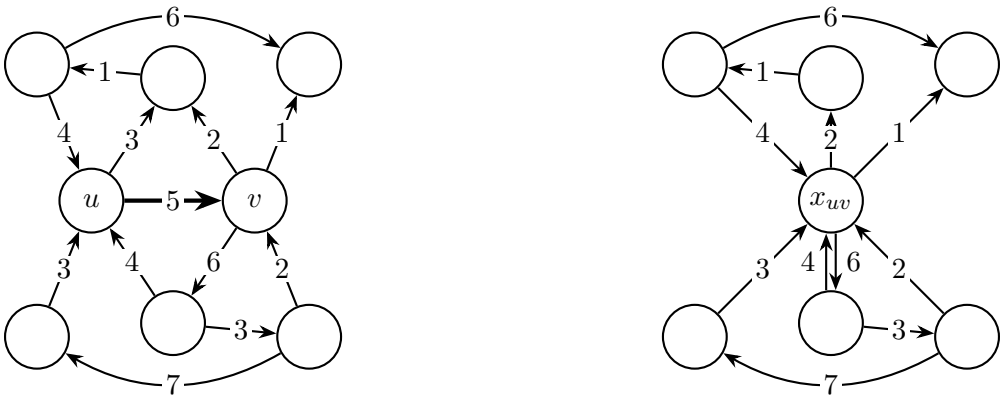

Figure 4.3: Illustration of the contraction of an arc (u, v). Left: original graph, right: contracted graph.

Algorithm: ORLIN(G, b, c)

input: circ. network $(G = (V, E), \infty, b)$, costs $c\colon E \to \mathbb{R}$
output: minimum cost b-flow

$r_v \leftarrow v \quad \forall v \in V$
$F \leftarrow \emptyset$, $H \leftarrow G$, $\varepsilon \leftarrow 1/n$
$f \leftarrow 0$, $\gamma \leftarrow \max_{v \in V} |b(v)|$
while $b \neq 0$**:**
  while $\exists v \in V$ **with** $|b(v)| > (1-\varepsilon)\gamma$**:**
    if $\exists s$-t-*path in* G_f **with** $b(s) > (1-\varepsilon)\gamma \wedge b(t) < -\varepsilon\gamma$
        or $b(s) > \varepsilon\gamma \wedge b(t) < -(1-\varepsilon)\gamma$**:**
      $P \leftarrow$ SHORTESTPATH(H_f, c, s, t)
      PUSH(P, γ)
    else
      fail
  if $f(e) = 0 \, \forall e \in H \setminus F$**:**
    $\gamma \leftarrow \min\{\gamma/2, \max_{v \in V} |b(v)|\}$
  else
    $\gamma \leftarrow \gamma/2$
  while $\exists e \in H \setminus F$ **with** $f(e) > 6n\gamma$**:**
    CONTRACT(e)
return f

Algorithm: CONTRACT(e)

input: arc $e \equiv (u, v) \in H_f$

$F \leftarrow F \cup \{e, \bar{e}\}$
$Q \leftarrow r_u$-r_v-path in F
PUSH$(Q, b(r_u))$
for $e' \equiv (x, y) \in H \setminus F$ **with** $\{r_x, r_y\} = \{r_u, r_v\}$**:**
  $H \leftarrow H \setminus \{e'\}$
for $y \in V$ **with** $r_y = r_u$**:**
  $r_y \leftarrow r_v$

An important consequence of our contraction strategy is that all arcs that have not been contracted or removed carry a flow that is an integer multiple of γ. This means we do not have to worry that our PUSH operations may not be well-defined.

Observation 4.32. Throughout the execution of ORLIN, it holds that $f(e)$ is an integer multiple of γ for all $e \in H \setminus F$.

Proof. As long as $f(e) \neq 0$ for some $e \in H \setminus F$, we halve γ in each iteration. Since we only augment by γ in H_f and since CONTRACT only uses arcs in F, it is ensured that $f(e)$ remains an integer multiple of γ for all $e \in H \setminus F$. Note that we can reduce γ by more than half if $f(e) = 0$ for all $e \in H \setminus F$ without violating this invariant. □

We divide the execution of ORLIN into maximal phases in which γ stays constant. For $\gamma' \in \mathbb{R}$, the *γ'-phase* is the phase in which $\gamma = \gamma'$. We first relate the number of augmentations outside of and within CONTRACT.

Lemma 4.33. In each phase of ORLIN, the number of augmentations outside of CONTRACT is not larger than the number of augmentations within CONTRACT plus the number of vertices v with $|b(v)| > (1 - \varepsilon)\gamma$ at the beginning of the phase.

Proof. Consider the potential function $\Phi := \sum_{v \in V} k_v := \sum_{v \in V} \lceil |b(v)|/\gamma - (1 - \varepsilon) \rceil \geq 0$. We show that during each augmentation along some s-t-path outside of CONTRACT, this potential Φ is reduced by at least 1. Without loss of generality, let $b(s) > (1 - \varepsilon)\gamma$ and $b(t) < -\varepsilon\gamma$. In case $|b(s)| \geq \gamma$, we reduce $|b(s)|/\gamma$ by 1 and therefore also k_s by 1. Otherwise, we reduce k_s by 1 to 0. If $b(t) \leq -\gamma/2$, we do not increase $|b(t)|$, therefore k_t does not increase. Otherwise, we have $b(t) \in (-\gamma/2, -\varepsilon\gamma)$, which implies $|b(t)|/\gamma \in (\varepsilon, 1/2)$ and $|b(t) + \gamma|/\gamma \in (1/2, 1 - \varepsilon)$ with $\varepsilon \leq 1/2$ for $n \geq 2$. Hence, k_t stays 0.

Now consider an augmentation by δ within CONTRACT. We increase $|b(r_v)|$ by at most δ and hence k_{r_v} by at most $\lceil \delta/\gamma \rceil$. On the other hand, we reduce k_{r_u} by at least $\lfloor \delta/\gamma \rfloor$. The increase of Φ is therefore at most 1.

At the beginning of the phase, Φ is bounded by the number of vertices v with $|b(v)| > (1 - \varepsilon)\gamma$, since $|b(v)| \leq (1 - \varepsilon)2\gamma$ and thus $k_v \leq 1$, otherwise the previous phase would not have ended. The number of augmentations outside of CONTRACT is therefore bounded by the number of such vertices plus the number of augmentations within CONTRACT. □

We now establish that the algorithm is well-defined in the sense that it maintains a $(b^\star - b)$-flow, where b^* denotes the initial value of b.

Lemma 4.34. ORLIN maintains the invariant that f is a $(b^\star - b)$-flow with $f(e) > 0$ for all $e \in G \cap F$.

Proof. To show that f is a b'-flow for some b', we need to show that $f(e) \geq 0$ for all $e \in G$. Arcs in $G \setminus H$ do not change their flow values. Consider an arc $e \in H \setminus F$. By Observation 4.32, the flow along e, and thus the residual capacity of $\bar{e}$, is always an integer multiple of γ. In CONTRACT we only augment along arcs in F. Since, otherwise, we only augment flow by γ and only along arcs of positive residual capacity, we never reduce $f(e)$ below 0. For $e \in G \cap F$ we additionally need to show that we never reduce $f(e)$ to 0. We show that even the total magnitude of *all* augmentations after e was added to F would not suffice to reduce $f(e)$ to 0.

We show that the total magnitude of all augmentations during a γ-phase and all subsequent phases is at most $6n\gamma$ along each arc. We know that there are at most $n-1$ CONTRACT operations overall, since only arcs between vertices in different components of F are contracted. Let k be the number of CONTRACT operations in a γ'-phase with $\gamma' \leq \gamma$. We know that, at the beginning of the γ'-phase, we have $|b(v)| \leq 2(1-\varepsilon)\gamma' < 2\gamma$ for all vertices $v \in V$, because the previous phase ended. It follows that the total magnitude of all augmentations along some arc within CONTRACT during the γ'-phase is at most $2k\gamma$, i.e., less than 2γ on average per CONTRACT operation (we ignore flow changes that are undone in the same phase and use that F corresponds to a forest). In total, the magnitude of all remaining augmentations along a fixed arc inside CONTRACT is strictly less than $2n\gamma$.

By Lemma 4.33, in each γ'-phase with $\gamma' \leq \gamma$, the number of augmentations outside of CONTRACT is at most equal to the number of other augmentations plus the number of vertices v with $|b(v)| > (1-\varepsilon)\gamma'$, i.e., at most $2n-1$. Since each such augmentation is by γ', the magnitude of these augmentations is less than $2n\gamma'$.

We only insert $e \in E$ into F once $f(e) > 6n\gamma$. Note that γ is at least halved after every phase. The total magnitude in the γ-phase and all subsequent phases is thus less than

$$2n\gamma + \sum_{i=0}^{\infty} 2n\frac{\gamma}{2^i} = 6n\gamma.$$

It follows that we do not augment enough flow along $\bar{e}$ to ever reduce the flow along e to 0.

Finally, observe that we update b whenever we push flow such that $\mathrm{ex}_f(v) = -(b^\star(v) - b(v))$ for all $v \in V$. Hence, f is a $(b^\star - b)$-flow. □

The following is the key lemma for the correctness of ORLIN. We show that our simplification strategy via contractions does not introduce errors.

Lemma 4.35. ORLIN maintains the invariant that f is a minimum-cost $(b^\star - b)$-flow, provided that c does not induce negative cycles.

Proof. We show that the invariant is maintained and, additionally, shortest paths in H_f always are shortest paths in G_f. Both holds initially for $f = 0$, since we assume that c does not induce negative cycles (Theorem 4.10).

We show that, as long as f has minimum cost, every shortest u-v-path P in H_f is also a shortest path in G_f. Note that, since f has minimum cost, c cannot induce negative cycles in G_f (Theorem 4.10), and shortest paths in H_f, G_f exist. Assume there was a u-v-path P' in G_f shorter than P. We can replace every arc $e \in P' \setminus H_f$ by a (unique) path P_e in F to obtain a u-v-walk W in H_f. By assumption, we thus have $c(P') < c(P) \leq c(W)$. This means that there is an arc $e \in P' \setminus H_f$ for which $c(P_e) > c(e)$. By Lemma 4.34 and since capacities are unbounded, all reverse arcs of arcs $e' \in P_e$ have a positive residual capacity, thus $\bar{P}_e \subseteq G_f$. We consider the cycle $e \oplus \bar{P}_e \subseteq G_f$. The cost of this cycle is

$$c(e) + c(\bar{P}_e) = c(e) - c(P_e) < 0,$$

hence c induces a negative cycle in G_f. This is a contradiction to optimality of f (Theorem 4.10), and we conclude that P must be a shortest path in G_f.

Consequently, in every step, we augment along a shortest path in G_f. This is also true within CONTRACT, since every path in F is a shortest path in H_f and thus in G_f. By Lemma 4.21, it follows that the resulting flow still has minimum cost, which means that our invariant is maintained. □

Finally, we conclude that ORLIN computes the correct result by refining Lemma 4.22.

Theorem 4.36. If c does not induce negative cycles, ORLIN only fails if no b-flow exists and otherwise returns a minimum-cost b-flow.

Proof. If ORLIN terminates without failing, the statement follows from $b = 0$ and Lemma 4.35. Now assume, without loss of generality, that the algorithm fails because there is a vertex s with $b(s) > (1-\varepsilon)\gamma$ but no vertex t with $b(t) < -\varepsilon\gamma$ that is reachable from s in G_f. Let R be the set of all vertices reachable from s in G_f. Since capacities are unbounded, we have $\delta_G^+(R) = \emptyset$ (and not just $\delta_{G_f}^+(R) = \emptyset$). Moreover, we have $f(e) = 0$ for all $e \in \delta_G^-(R)$, since $\delta_{G_f}^+(R) = \emptyset$. Because f is a $(b^\star - b)$-flow (Lemma 4.34), it follows that $\sum_{v\in R}(b^\star(v) - b(v)) = \sum_{v\in R}(-\mathrm{ex}_f(v)) = \sum_{e\in\delta_G^+(R)} f(e) - \sum_{e\in\delta_G^-(R)} f(e) = 0$. We obtain

$$\sum_{v\in R} b^\star(v) = \sum_{v\in R} b(v) = b(s) + \sum_{v\in R\setminus\{s\}} b(v) > (1-\varepsilon)\gamma - (n-1)\varepsilon\gamma \overset{\varepsilon=1/n}{=} 0,$$

but $\delta_G^+(R) = \emptyset$ and thus $\sum_{e\in\delta_G^+(R)} \mu(e) = 0 < \sum_{v\in R} b^\star(v)$. By Proposition 4.4, this means that no $b^\star$-flow exists in $(G, \infty, b^\star)$, and ORLIN can safely fail. □

To bound the running time of the algorithm, much like for the previous algorithms, we need to establish that we make progress over time.

Lemma 4.37. If we have $|b(z)| > (1-\varepsilon)\gamma$ for $z \in V$ at some point during the execution of ORLIN, then the connected component of z in F grows within the next $\lceil 2\log_2 n + \log_2 m\rceil + 3$ phases.

Proof. Let $|b(z)| > (1-\varepsilon)\gamma_1$ for some $z \in V$ at the beginning of the γ_1-phase for $\gamma_1 > 0$. Let $\gamma_0 \geq 2\gamma_1$ be the value of γ in the previous phase (or $\gamma_0 := 2\gamma_1$ if we are in the first phase) and let γ_2 be the value of γ after $k := \lceil 2\log_2 n + \log_2 m\rceil + 2$ additional phases. Since γ is reduced by factor 2 or more after each phase, we have

$$\gamma_1 \geq 2^k\gamma_2 \geq 2^{2\log_2 n} 2^{\log_2 m} 2^2 \gamma_2 = 4n^2 m\gamma_2. \tag{4.10}$$

Let f_1 and b_1 be the values of f and b, respectively, *at the beginning* of the γ_1-phase, and let f_2 and b_2 be the values *at the end* of the γ_2-phase. Let further $b^\star$ denote the original value of b, and let C_z be the connected component of z in F at the beginning of the γ_1-phase. Assume that C_z does not grow in the next k phases after the γ_1-phase. In CONTRACT we ensure that $b(v) = 0$ for all v with $r_v \neq v$. Therefore, we have $r_z = z$ and $b_1(v) = 0$ for all $v \in C_z \setminus \{z\}$ at the beginning of the γ_1-phase. Since f_1 is a $(b^\star - b_1)$-flow (Lemma 4.34), we further have

$$\Big(\sum_{v\in C_z} b^\star(v)\Big) - b_1(z) = \sum_{v\in C_z}(b^\star(v) - b_1(v)) = \sum_{v\in C_z}(-\mathrm{ex}_{f_1}(v)) = \sum_{e\in\delta_G^+(C_z)} f_1(e) - \sum_{e\in\delta_G^-(C_z)} f_1(e) =: r\gamma_0,$$

with $r \in \mathbb{Z}$ by Observation 4.32, since $(\delta_G^-(C_z) \cup \delta_G^+(C_z)) \subseteq H \setminus F$. If $r \cdot b_1(z) \geq 0$, we obtain

$$\Big|\sum_{v\in C_z} b^\star(v)\Big| = |r\gamma_0 + b_1(z)| \geq |b_1(z)| > (1-\varepsilon)\gamma_1 \overset{n\geq 2}{\geq} \frac{1}{n}\gamma_1, \tag{4.11}$$

and otherwise (i.e., if $r \cdot b_1(z) < 0$) we have $r \neq 0$ and thus (note that $|b_1(z)| < (1-\varepsilon)\gamma_0$, since the previous phase ended)

$$\Big|\sum_{v\in C_z} b^\star(v)\Big| = |r\gamma_0 + b_1(z)| = |r\gamma_0| - |b_1(z)| > \gamma_0 - (1-\varepsilon)\gamma_0 = \frac{1}{n}\gamma_0 > \frac{1}{n}\gamma_1. \tag{4.12}$$

Note that we need $\varepsilon > 0$ in (4.12) to get a strictly positive right-hand side.

Since f_2 is a $(b^\star - b_2)$-flow (Lemma 4.34), we obtain as above

$$\big(\sum_{v\in C_z} b^\star(v)\big) - b_2(z) = \sum_{e\in\delta_G^+(C_z)} f_2(e) - \sum_{e\in\delta_G^-(C_z)} f_2(e). \tag{4.13}$$

Together with $|b_2(z)| \leq (1-\varepsilon)\gamma_2$ (since the γ_2-phase ended), this yields

$$\begin{aligned}
\sum_{e\in\delta_G^+(C_z)\cup\delta_G^-(C_z)} f_2(e) \quad &\geq \quad \Big|\sum_{e\in\delta_G^+(C_z)} f_2(e) - \sum_{e\in\delta_G^-(C_z)} f_2(e)\Big| \\
&\overset{(4.13)}{=} \quad \Big|\big(\sum_{v\in C_z} b^\star(v)\big) - b_2(z)\Big| \\
&\geq \quad \Big|\sum_{v\in C_z} b^\star(v)\Big| - \Big|b_2(z)\Big| \\
&\overset{(4.11),(4.12)}{>} \quad \frac{1}{n}\gamma_1 - (1-\varepsilon)\gamma_2 \\
&\overset{(4.10)}{\geq} \quad 4nm\gamma_2 - (1-\varepsilon)\gamma_2 \\
&> \quad (4nm-1)\gamma_2 \\
&> \quad m(6n\frac{\gamma_2}{2}).
\end{aligned}$$

But this means that at least one arc in $\delta_G^+(C_z) \cup \delta_G^-(C_z)$ carries a flow of at least $6n\gamma$ at the beginning of the next phase and is therefore contracted. This causes the component of z in F to grow. □

Now, we have everything we need to bound the running time of Orlin. Note that it is crucial that we reduce γ by more than half whenever possible.

Theorem 4.38. Orlin can be implemented to run in time $\mathcal{O}((m + n\log n)\, n\log n)$.

Proof. We can make use of a vertex potential as in SuccessiveShortestPath with potential. This allows to compute shortest paths in time $\mathcal{O}(m + n\log n)$. It remains to bound the number of augmentations and phases. There are at most $n-1$ augmentations within Contract, since we merge two distinct components in each Contract operation.

We count the maximum number of phases until some arc is contracted. As soon as an augmentation is performed outside of Contract, by Lemma 4.37, the number of phases until the next Contract is bounded by $\mathcal{O}(\log m) = \mathcal{O}(\log n)$. Before that, however, there may be additional phases without vertices v satisfying

$|b(v)| > (1-\varepsilon)\gamma$. If there is at least one arc $e \in H \setminus F$ with $f(e) > 0$, then this arc carries at least γ units of flow (Observation 4.32). After $\lceil \log_2 n \rceil + 3$ phases without change in flow, we have $f(e) \geq 2^{\lceil \log_2 n \rceil + 3}\gamma > 6n\gamma$, since we reduce γ by at least half after each phase. Then, e can be contracted. On the other hand, if $f(e) = 0$ for all arcs $e \in G \setminus F$, the algorithm sets $\gamma = \max_{v \in V} |b(v)|$, which leads to an augmentation in the next phase. It follows that after at most $\mathcal{O}(\log n)$ phases some arc is contracted, which means that we have $\mathcal{O}(n \log n)$ phases overall.

By Lemma 4.33, the number of augmentations outside of CONTRACT is at most $n - 1$ plus the number of pairs (γ, z) with $|b(z)| > (1-\varepsilon)\gamma$ at the beginning of the γ-phase. At such a point in time, it holds that $r(z) = z$, since only the representative of a connected component may have $b(z) \neq 0$. By Lemma 4.37, the connected component grows within the next $\mathcal{O}(\log m + \log n) = \mathcal{O}(\log n)$ phases. A connected component can therefore be responsible for at most $\mathcal{O}(\log n)$ augmentations, before it changes. The total number of different connected components throughout the course of the algorithm is $2n - 1$, since we start with n components and reduce the number of components by one in each CONTRACT operation, until we are left with a single component. Hence, the total number of augmentations is $\mathcal{O}(n \log n)$. □

5 Maximum Cardinality Matchings

We now turn to the maximum matching problem on general graphs. Elementary results and algorithms for bipartite graphs were discussed in the lecture *Algorithmic Discrete Mathematics*. We will present a better algorithm for bipartite graphs and extend results to general graphs. We begin by recalling basic definitions.

Definition 5.1. A *matching* in an undirected graph $G = (V, E)$ is a set $M \subseteq E$ of pairwise disjoint edges. A vertex $v \in V \setminus \bigcup_{e \in M} e$ is *(M-)exposed* and we write $v \notin M$. A vertex $v \in \bigcup_{e \in M} e$ is *(M-)covered* and we write $v \in M$. The matching M is

- *perfect* if all vertices of G are M-covered (i.e., if $|M| = \frac{1}{2}|V|$),
- *maximal* if there is no matching $M' \supsetneq M$ in G,
- *maximum (cardinality)* if it maximizes the number of edges in the matching, i.e., if $|M| = \max\{|M'| \mid M' \text{ is a matching in } G\}$.

With this, the maximum matching problem is defined as follows. We sometimes explicitly speak of maximum *cardinality* matchings to distinguish them from maximum *weight* matchings.

Maximum (Cardinality) Matching Problem	
input:	graph $G = (V, E)$
problem:	find maximum cardinality matching $M \subseteq E$ of G

As before, from now on we fix an undirected graph $G = (V, E)$. We recall the following characterization of maximum matchings.

Definition 5.2. Let M be a matching in G. A u-v-path P in G is *(M-)alternating* if $P \setminus M$ is a matching. If, additionally, $u, v \notin M$, then P is *(M-)augmenting*.

Theorem 5.3. Let M be a matching in G. Then, M is a maximum matching in G if and only if there is no M-augmenting path in G.

5.1 Bipartite Graphs Revisited

We begin by revisiting the maximum matching problem in bipartite graphs. This problem can be reduced to the maximum flow problem by introducing the following auxiliary network (see Figure 5.1).

Definition 5.4. Let $G = (U \cup V, E)$ be a bipartite graph with $s, t \notin (U \cup V)$. The *flow network* $N_G = (H, \mu, s, t)$ *induced by* G is given by $H = (V_H, E_H)$ with vertices $V_H = U \cup V \cup \{s, t\}$ and arcs $E_H = \{(u, v) \in U \times V \mid \{u, v\} \in E\} \cup (\{s\} \times U) \cup (V \times \{t\})$, and $\mu(e) = 1$ for all $e \in E_H$.

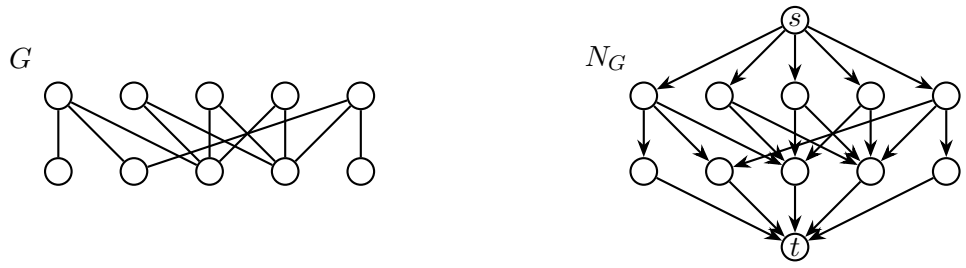

Figure 5.1: Left: bipartite graph G. Right: flow network N_G induced by G ($\mu = 1$).

With this, the maximum matching problem can easily be solved by invoking any maximum flow algorithm. For example, we may use the following algorithms. Note that FordFulkerson and Dinic are guaranteed to compute a maximum flow f with $f(e) \in \{0, 1\}$ for all edges e, since $\mu(e) \in \{0, 1\}$.

Algorithm: FordFulkersonMatching(G)	**Algorithm:** DinicMatching(G)
input: bipartite graph $G = (U \cup V, E)$	**input**: bipartite graph $G = (U \cup V, E)$
output: maximum card. matching $M \subseteq E$ in G	**output**: maximum card. matching $M \subseteq E$ in G
$(H, \mu, s, t) \leftarrow N_G$	$(H, \mu, s, t) \leftarrow N_G$
$f \leftarrow$ FordFulkerson(H, μ, s, t)	$f \leftarrow$ Dinic(H, μ, s, t)
return $\{\{u, v\} \in E \mid (u, v) \in U \times V \wedge f((u, v)) = 1\}$	**return** $\{\{u, v\} \in E \mid (u, v) \in U \times V \wedge f((u, v)) = 1\}$

Importantly, in this application, FordFulkersonMatching performs just as well as DinicMatching in terms of running time.

Theorem 5.5. Both FordFulkersonMatching and DinicMatching solve the maximum matching problem in time $\mathcal{O}(nm)$.

Proof. The first part of the statement follows from the fact that FordFulkerson has running time $\mathcal{O}(m \cdot |f^\star|)$ (see lecture *Algorithmic Discrete Mathematics*) and $|f^\star| < n$ in N_G.

The running time of DinicMatching follows from the fact that N_G has unit capacities, implying that BlockingFlow only uses saturating pushs. With this observation, the analysis in the proof of Theorem 3.9 yields a running time of $\mathcal{O}(m)$ for BlockingFlow. Together with Proposition 3.4, we obtain the claimed bound on the running time of DinicMatching. □

We can further simplify Dinic for flow networks induced by bipartite graphs by realizing that every vertex except s and t can only be part of a single augmenting path, since their indegree our outdegree is 1 and all arcs have unit capacity. This allows to replace BlockingFlow by a simple variant of DFS without the need to topologically sort the graph. The resulting improvement yields the Hopcroft-Karp algorithm:

Algorithm: HopcroftKarp(G)

input: bipartite graph $G = (U \cup V, E)$
output: maximum cardinality matching $M \subseteq E$ in G

$(H, \mu, s, t) \leftarrow N_G$
$f \leftarrow 0$
while $\exists$ *s-t-path in* H_f**:**
  $H_{f,\mathrm{L}} \leftarrow (V, \{(v,w) \in H_f \mid d_{H_f}(s,w) = d_{H_f}(s,v) + 1\})$
  $f' \leftarrow$ BipartiteBlockingFlow$(H_{f,\mathrm{L}}, \mu_f, s, t)$
  $f \leftarrow f + f'$
return $\{\{u,v\} \in E | (u,v) \in U \times V \wedge f((u,v)) = 1\}$

In BipartiteBlockingFlow we simply use the vertex-disjoint (except for s and t) paths found by DFS as augmenting paths.

Algorithm: BipartiteBlockingFlow(G, μ, s, t)

input: network $(G = (V, E), \mu = 1, s, t)$ with $\max\{d^+(v), d^-(v)\} \leq 1 \, \forall v \in V$
output: blocking flow

$f \leftarrow 0$
while $P \leftarrow$ *DFS*$(G, s, t) \neq \emptyset$**:**
  Augment$(P, 1)$
return f

Algorithm: DFS(G, u, t)

input: graph $G = (V, E)$, vertices $u, t \in V$
output: u-t-path in G along unmarked vertices

if $u = t$**:**
  return (t)
mark u (t is never marked)
for $v \in \Gamma(u)$ *with* v **not marked:**
  if $P \leftarrow$ *DFS*$(G, v, t) \neq \emptyset$**:**
    return $(u) \oplus P$
return $\emptyset$

We show that this method of computing blocking flows is correct for flow networks.

Proposition 5.6. BipartiteBlockingFlow computes a blocking flow in time $\mathcal{O}(m)$.

Proof. We show that BipartiteBlockingFlow computes a blocking flow. The running time then follows, because every vertex is visited at most once and only if it is reachable from s.

For the sake of contradiction, assume that there is still an s-t-path P in $G \cap G_f$ upon termination of BipartiteBlockingFlow. Then, at least one vertex of $P \setminus \{s,t\}$ must be marked, otherwise DFS would have still found some s-t-path. Let $v \in V \setminus \{s,t\}$ the last such vertex along P. Then, a v-t-path without marked vertices exists. This means that DFS either did not visit all neighbors of v, or $t \in \Gamma(v)$. In either case, DFS must have found a v-t-path at some point, which means that BipartiteBlockingFlow must have augmented flow over some path P' that contains v. From $\mu = 1$ it follows that P and P' do not share arcs. Since both P and P' contain v, this implies that v has both in- and out-degree greater one, contradicting the assumption on the input of the algorithm. □

With Proposition 3.4, this again immediately gives a running time of $\mathcal{O}(nm)$, which does not yet improve upon FordFulkersonMatching. But we can further strengthen the analysis. Note that DinicMatching achieves the same running time.

Theorem 5.7. HopcroftKarp computes a maximum matching in bipartite graphs in time $\mathcal{O}(m\sqrt{n})$.

Proof. We have already seen that the length of a shortest s-t-path in G_f increases in every iteration of Dinic (Proposition 3.4). Consider the situation after iteration $\sqrt{n}+2$. Let M be the bipartite matching in G corresponding to the current flow in H, and let $M^\star$ be a maximum matching in G. The symmetric difference $M \triangle M^\star$ has at most one edge of M and at most one of $M^\star$ at every vertex. This allows to partition $M \triangle M^\star$ into both M- and $M^\star$-alternating cycles and paths. The number of M-augmenting paths in this partition is exactly $|M^\star| - |M|$, since there cannot be any $M^\star$-augmenting paths (Theorem 5.3). We also know that every M-augmenting path corresponds to an augmenting path in H_f, which, in turn, has length at least $\sqrt{n}+2$. It follows that every M-augmenting path in $M \triangle M^\star$ has length at least $\sqrt{n}$. Since these paths are disjoint, there can be at most $\sqrt{n}$ such paths, and, hence, $|M^\star| - |M| \leq \sqrt{n}$. Since the size of the matching increases in every iteration (Proposition 3.4), HopcroftKarp needs at most $\sqrt{n}$ more iterations to reach a maximum cardinality matching. Overall, the number of iterations is bounded by $2\sqrt{n}+2 = \mathcal{O}(\sqrt{n})$.

To apply Proposition 5.6 and conclude that each iteration takes time $\mathcal{O}(m)$, which yields the claimed running time, we need to show that the algorithm maintains the invariant that degrees in $H_{f,\mathrm{L}}$ are bounded (and not just in H). For this, it is sufficient to prove the invariant for $H_f \supseteq H_{f,\mathrm{L}}$. It initially holds, because H is induced by a bipartite graph, and it is maintained because every augmentation is by one unit of flow and $\mu = 1$, which implies that in- and outdegrees remain unchanged in H_f. □

5.2 Characterization of Matchings

We turn to matchings in general (not necessarily bipartite) undirected graphs. We first give an algebraic characterization of graphs that admit perfect matchings. This characterization relies on the following matrix similar to an adjacency matrix, but with variables as its entries. Recall that an *orientation* of an undirected graph is a directed graph that is obtained by assigning a direction to each edge.

Definition 5.8. For $\boldsymbol{x} = (x_e)_{e \in E}$ and an arbitrary orientation G' of G, the *Tutte matrix* $T(\boldsymbol{x}) = (t_{u,v})_{u,v \in V}$ is given by

$$t_{u,v} := \begin{cases} x_e, & \text{if } e = (u,v) \in G', \\ -x_e, & \text{if } e = (v,u) \in G', \\ 0, & \text{otherwise.} \end{cases}$$

It is a beautiful mathematical connection that the determinant of the Tutte matrix vanishes if and only if the underlying undirected graph has no perfect matching.

Theorem 5.9 (Tutte)**.** An undirected graph G *does not* have a perfect matching if and only if all terms in the polynomial $\det T(x)$ cancel out (symbolically).

Proof. Let $V = \{1, \ldots, n\}$ and let S_n denote the set of all permutations of V. By the Leibniz formula for determinants, we have

$$\det T(\boldsymbol{x}) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \prod_{v \in V} t_{v,\pi(v)}.$$

First assume that a perfect matching M exists in G. Consider the permutation $\pi(u) = v$ and $\pi(v) = u$ for all $\{u, v\} \in M$. For this and no other permutation we have

$$\prod_{v \in V} t_{v,\pi(v)} = \prod_{e \in M} (-x_e^2).$$

Therefore, this term does not cancel with any other term in $\det T(\boldsymbol{x})$.

Now, conversely, assume that $\det T(\boldsymbol{x}) \neq 0$. Let $S'_n := \{\pi \in S_n \mid \prod_{v \in V} t_{v,\pi(v)} \neq 0\}$ denote the set of all permutations that contribute to $\det T(\boldsymbol{x})$. We interpret every permutation $\pi \in S_n$ as a directed graph H_π with arcs (u, v) between every $u, v \in V$ with $v = \pi(u)$. Every vertex of H_π has in- and out-degree 1, and, hence, H_π consists of cycles – the permutation cycles of π. For all $\pi \in S'_n$ we have that H_π is a subgraph of $G' \cup \bar{G}'$, where G' is an arbitrary orientation of G.

We show that the contributions of all permutations $\pi \in S'_n$ that contain at least one odd cycle in H_π cancel out in $\det T(\boldsymbol{x})$. Let $r \in S_n$ be the permutation which changes the orientation of the first odd cycle in π, i.e., the odd cycles in H_π and $H_{r \circ \pi}$ containing the smallest vertex have opposite orientations and all other cycles are the same.

We first show that $\operatorname{sgn}(\pi) = \operatorname{sgn}(r \circ \pi)$, i.e., π can be translated to $r \circ \pi$ with an even number of swaps. Let $v_1, \dots, v_{2k+1}$ be the first odd cycle in H_π. For $i = 1, \dots, k$, we swap (in this order) $v_{2i-1}$with v_{2k} and, analogously, v_{2i} with v_{2k+1}. Let $v_{-1} := v_{2k}$ and $v_0 := v_{2k+1}$. Now, for all $i \in \{1, \dots, k\}$, we ensured that $r(v_i) = v_{i-2}$, since v_{i-2} was first swapped with v_{2k} or v_{2k-1} and then (from there) with v_i. Hence, we have

$$r(\pi(v_i)) = r(v_{i+1}) = v_{i-1},$$

i.e., the first odd cycle is flipped. As we used an even number of swaps, we have $\operatorname{sgn}(\pi) = \operatorname{sgn}(r(\pi))$.

Since we changed the orientation of an odd cycle, the sign of an odd number of entries of $\prod_{v \in V} t_{v,\pi(v)}$ changed. In particular, $\pi \in S'_n$ implies $r \circ \pi \in S'_n$. Consequently, $\prod_{v \in V} t_{v,\pi(v)} = -\prod_{v \in V} t_{v,r(\pi(v))}$ and the contributions of π and $r \circ \pi$ cancel out. Furthermore, $r \circ r \circ \pi = \pi$, therefore we can pair the elements of S'_n with odd cycles, such that the contributions of each pair to $\det T(\boldsymbol{x})$ cancels out.

Now, since $\det T(\boldsymbol{x}) \neq 0$, a permutation with only even cycles must exist. But then we can construct a perfect matching by taking every second edge of each cycle (ignoring orientations). □

While the above characterization is beautiful, it is often difficult to apply. We now derive a much more useful characterization. To this end, we need some definitions.

Definition 5.10. Let $G = (V, E)$ be an undirected graph and $X \subseteq V$. Then $q(X) := q_G(X)$ denotes the number of connected components with an odd number of vertices in the induced subgraph $G[V \setminus X]$.

In order to inductively characterize graphs that admit perfect matchings, we need a way of splitting a graph into smaller parts that allows us to relate the existence of a perfect matching in the original graph to the existence of a perfect matching in every part. Sets of vertices $X \subseteq V$ with $q(X) = |X|$ allow us to do this nicely, because they split the graph into parts in such a way that every part of odd cardinality in $V \setminus X$ needs to have exactly one edge to X in every perfect matching. All other edges need to be within parts. This justifies the following definitions.

Definition 5.11. A *barrier* is a set of vertices $X \subseteq V$ with $q(X) = |X|$.

Definition 5.12. An undirected graph $G = (V, E)$ is called *factor critical* if $G - v$ has a perfect matching for all $v \in V$.

The following lemma will be useful.

Lemma 5.13. For all undirected graphs $G = (V, E)$ and all $X \subseteq V$ it holds that $|V| \pm q(X) \pm |X|$ is even.

Proof. Let $\mathcal{Z}$ be the partition of V into the vertex sets of the connected components of $G - X$ and the set X. Then, $q(X) \pm |X|$ is even if and only if $\mathcal{Z}$ contains an even number of odd (cardinality) sets. Since $\mathcal{Z}$ is a partition of V, this is the case if and only if $|V|$ is even. It follows that $|V| \pm q(X) \pm |X|$ is always even. □

We now present a characterization, similar to the marriage theorem for bipartite graphs.

Theorem 5.14 (Tutte)**.** An undirected graph $G = (V, E)$ has a perfect matching if and only if

$$q(X) \leq |X| \quad \forall X \subseteq V. \tag{5.1}$$

Proof. If a perfect matching M exists in G, then, for every $X \subseteq V$, every odd component of $G - X$ must have a matching edge to a distinct element of X. Hence, $q(X) \leq |X|$.

Conversely, assume that the Tutte condition (5.1) holds. We show, by induction over $n = |V|$, that a perfect matching exists. For $n = 2$, by the Tutte condition, we have $q(\emptyset) = 0$ so there must be an edge between the two vertices. Thus, a perfect matching exists.

Now consider the case $n > 2$. The Tutte condition implies $q(\emptyset) = 0$, hence $\emptyset$ is a barrier and n must be even. We can therefore choose a barrier X of maximum size. By definition, $G - X$ has $|X|$ odd connected components. Let $x \in V$ be a vertex of an even component. By the Tutte-condition, we have $q(X \cup \{x\}) \leq |X| + 1$, but it also holds that $q(X \cup \{x\}) \geq q(X) + 1 = |X| + 1$. Hence, $X \cup \{x\}$ is a barrier – a contradiction to the maximality of X. Hence, $G - X$ cannot have even components.

We claim that every connected component of $G - X$ is factor critical. Let C be such a component and $v \in C$ be one of its vertices. Assume that $C - v$ did not have a perfect matching. By induction, it then follows that the Tutte condition is violated for the graph $C - v$. In other words, there must be a set of vertices $Y \subseteq C - v$ with $q_{C-v}(Y) > |Y|$. Since $G - X$ has no even components, C is odd and $C - v$ is even. By Lemma 5.13, $q_{C-v}(Y) - |Y|$ must be even as well. It follows that

$$q_{C-v}(Y) \geq |Y| + 2.$$

Since X, Y, and $\{v\}$ are pairwise disjoint, it follows that

$$\begin{aligned} q(X \cup Y \cup \{v\}) &= q(X) - 1 + q_C(Y \cup \{v\}) \\ &= |X| - 1 + q_{C-v}(Y) \\ &\geq |X| - 1 + |Y| + 2 \\ &= |X \cup Y \cup \{v\}|. \end{aligned}$$

By the Tutte condition, we also have $q(X \cup Y \cup \{v\}) \leq |X \cup Y \cup \{v\}|$. Therefore, $X \cup Y \cup \{v\}$ is a barrier – a contradiction to maximality of X. We conclude that every component of $G - X$ is factor critical.

Now consider the bipartite graph $G' = (X \cup \mathcal{Z}, E')$, where $\mathcal{Z}$ is the set of components of $G - X$ and E' contains an edge between $x \in X$ and $C \in \mathcal{Z}$ if and only if $x \in \Gamma_G(C)$. If there is no perfect matching in G', then, by the marriage theorem (see *Algorithmic Discrete Mathematics*), there exists $\mathcal{A} \subseteq \mathcal{Z}$ with $|\Gamma_{G'}(\mathcal{A})| < |\mathcal{A}|$. Since the components in $\mathcal{Z}$ are not connected in $G - X$, the set $\Gamma_{G'}(\mathcal{A}) \subseteq X$ isolates every component in $\mathcal{A}$ from the rest of G. Since every $C \in \mathcal{A} \subseteq \mathcal{Z}$ is an odd component of $G - X$, it follows that $q_G(\Gamma_{G'}(\mathcal{A})) \geq |\mathcal{A}|$. We conclude that $q_G(\Gamma_{G'}(\mathcal{A})) > |\Gamma_{G'}(\mathcal{A})|$, which is a contradiction to the Tutte condition.

Thus, there must be a perfect matching in G'. Since every $C \in \mathcal{Z}$ is factor critical, we can transform this perfect matching into a perfect matching $M^\star$ of G: For every edge $\{x, C\}$ in the matching, let $v \in C$ with $\{x, v\} \in E$ be a vertex of C that shares an edge with x in G. We include this edge in $M^\star$. Because C is factor critical, there is a perfect matching in $C - v$, which we can also include in $M^\star$. □

While Tutte's theorem gives a characterization for the existence of perfect matchings, it does not apply for graphs that do not admit perfect matchings. We now extend this characterization to capture maximum matchings in general. Note that the following theorem subsumes Tutte's theorem for perfect matchings, since then $|M^\star| = n/2$ and $q(X) \leq |X|$, $q(\emptyset) = 0$.

Theorem 5.15 (Berge-Tutte)**.** For all undirected graphs $G = (V, E)$ it holds that

$$2|M^\star| + \max_{X \subseteq V} \{q(X) - |X|\} = |V|,$$

where $M^\star$ denotes a maximum cardinality matching in G.

Proof. For every $X \subseteq V$, every matching M must leave at least $q(X) - |X|$ vertices exposed, since every odd component of $G - X$ either has an exposed vertex or a matching edge to a vertex of X. It follows that $2|M| + q(X) - |X| \leq |V|$.

Now let

$$k := \max_{X \subseteq V} \{q(X) - |X|\}$$

and $H = (V', E')$ be the graph G with k additional vertices that each have edges to all other vertices. Assume that H has a perfect matching $M_H^\star$. Then, for a maximum matching $M^\star$ in G, it holds that $|M^\star| \geq |M_H^\star| - k$, since we can construct a matching in G from $M_H^\star$ by removing all matching edges that contain a vertex of $V' \setminus V$. Since $M_H^\star$ is perfect, we have

$$2|M^\star| + k \geq 2|M_H^\star| - k = |V'| - k = |V|.$$

It remains to show that H always has a perfect matching. Assume this was not the case. By Tutte's theorem (Theorem 5.14), it follows that there is a set $Y \subseteq V'$ with $q_H(Y) > |Y|$. By Lemma 5.13, $|V'| = |V| + k$ is even. Hence, $q_H(\emptyset) = 0$ and thus $Y \neq \emptyset$. It follows that $q_H(Y) > |Y| \geq 1$, hence $H - Y$ consists of multiple components. But then Y must contain all vertices in $V' \setminus V$, since these are connected to all vertices. We obtain a contradiction to the definition of k (with $X = Y \cap V$):

$$q_G(Y \cap V) = q_H(Y) > |Y| = |Y \cap V| + k.$$ □

5.3 Edmonds' Algorithm

After having characterized the existence of matchings, we now focus on finding maximum matchings in general graphs algorithmically. We approach this problem greedily as follows: While there is still an unmatched vertex r, we take any neighbor $v \in \Gamma(r)$. If v is unmatched, we can augment the matching by adding the edge $\{r, v\}$ and restart. Otherwise, we can store the path from r to v's matching partner z. Let $T_{\text{even}} = \{r, z\}$. We can repeat the process with any neighbor of T_{even} that we have not considered yet. If this neighbor is unmatched, we found an augmenting path, which means that we can augment the matching and restart. Otherwise, we can store another path, add its endpoint to T_{even} in the same way, and repeat the process looking for unconsidered neighbors of T_{even}. In this way, we greedily build up the following structure (see Figure 5.2).

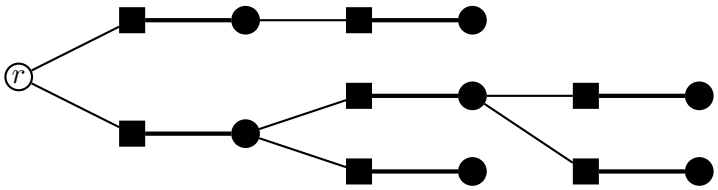

Figure 5.2: Illustration of an alternating tree. Thick edges are in the matching, square vertices are in T_{odd}, and circle-shaped vertices are in T_{even}.

Definition 5.16. An *alternating tree* with respect to a matching M in graph $G = (V, E)$ is a subtree $T = (V', E') \subseteq G$ rooted at $r \in V$ with the following properties.

(i) The root r is the only M-exposed vertex of T.

(ii) Every path in T starting at r is M-alternating.

(iii) For every leaf v of T the distance $d_T(r, v)$ is even.

We let $T_{\text{even}} \subseteq V'$ denote the set of vertices $v \in V'$ with even $d_T(r, v)$, and $T_{\text{odd}} := V' \setminus T_{\text{even}}$.

Assuming that we never encounter edges between two vertices in T_{even}, we are sure to eventually find a perfect matching if one exists. We can prove this formally, by proving that, if a perfect matching exists, we find neighbors in T_{even} until we have matched all vertices. In other words, it cannot be the case that all edges of $\delta_G(T)$ are adjacent to vertices in T_{odd}.

Observation 5.17. If an alternating tree T exists w.r.t. a matching M in G such that T_{odd} is a vertex cover of $G[T] \cup \delta_G(T)$, then G does not have a perfect matching.

Proof. Observe that $|T_{\text{odd}}| < |T_{\text{even}}|$, since r and every leaf of T are in T_{even}. If T_{odd} is a vertex cover of $G[T] \cup \delta_G(T)$, then every vertex of T_{even} is its own odd component in $G - T_{\text{odd}}$. This means that $q(T_{\text{odd}}) \geq |T_{\text{even}}| > |T_{\text{odd}}|$, i.e., the Tutte condition (5.1) is violated for $X = T_{\text{odd}}$. Therefore, by Tutte's theorem (Theorem 5.14), no perfect matching exists in G. □

Note that if we demand that G is bipartite, we can never encounter edges between vertices in T_{even}, because these edges would close an odd cycle in T. Hence, we have described an algorithm to find a perfect matching in a bipartite graph if one exists. A precise formulation of the resulting algorithm is given below. The operations AUGMENT and EXTENDTREE are illustrated in Figures 5.3 and 5.4, respectively.

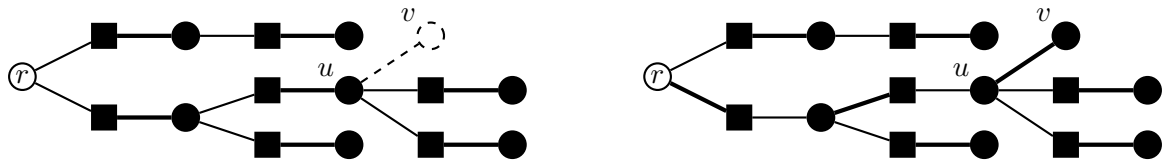

Figure 5.3: Illustration of the operation AUGMENT(u, v). Left: alternating tree before the augmentation; right: situation after the augmentation. Note that the alternating tree is *not* maintained.

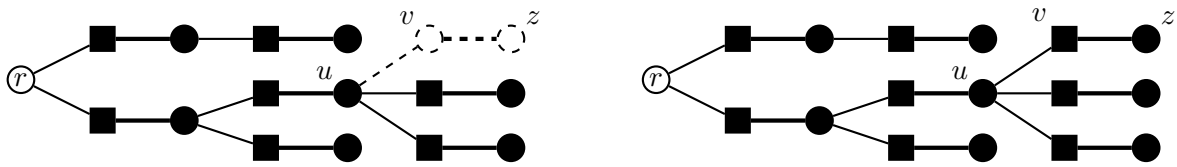

Figure 5.4: Illustration of the operation EXTENDTREE(u, v). Left: alternating tree before the extension; right: situation after the extension. Note that the alternating tree is maintained.

Algorithm: EDMONDSPERFECTBIPARTITE(G)

input: bipartite graph $G = (V, E)$
output: perfect matching in G

$M \leftarrow \emptyset$
MAKETREE$(\emptyset)$
while $\exists \{u, v\} \in E$ *with* $u \in T_{\text{even}}$ **and** $v \notin T$**:**
  if $v \notin M$**:**
    AUGMENT(u, v)
    MAKETREE$(\emptyset)$
  else
    EXTENDTREE(u, v)
if $T = \emptyset$**:**
  return M
else
  fail

Algorithm: MAKETREE(R)

if $\exists r \notin R$ *with* $r \notin M$**:**
  $T \leftarrow (\{r\}, \emptyset)$
  $T_{\text{even}} \leftarrow \{r\}$
  $T_{\text{odd}} \leftarrow \emptyset$
else
  $T \leftarrow \emptyset$, $T_{\text{even}} \leftarrow \emptyset$, $T_{\text{odd}} \leftarrow \emptyset$

Algorithm: AUGMENT(u, v)

$P \leftarrow r$-u-path in $T + \{u, v\}$
$M \leftarrow M \triangle P$

Algorithm: EXTENDTREE(u, v)

$\{z\} \leftarrow \Gamma_M(v)$
$T \leftarrow T + \{u, v\} + \{v, z\}$
$T_{\text{odd}} \leftarrow T_{\text{odd}} \cup \{v\}$
$T_{\text{even}} \leftarrow T_{\text{even}} \cup \{z\}$

We show formally that this algorithm indeed always finds a perfect matching in bipartite graphs or concludes that none exists.

Proposition 5.18. EDMONDSPERFECTBIPARTITE is correct.

Proof. We prove that EDMONDSPERFECTBIPARTITE maintains the invariant that M is a matching and T is an alternating tree with root r (or $T = \emptyset$). This is the case initially, since every vertex is M-exposed, $M = \emptyset$, and $T = (\{r\}, \emptyset)$. Now consider an iteration for edge $\{u, v\} \in E$ with $u \in T_{\text{even}}$ and $v \in V \setminus T$. First assume that v is M-exposed and let P_{ru} be the unique r-u-path in T. Then, the path $P = P_{ru} \oplus (v)$ is an M-augmenting path, since r and v are M-exposed and P_{ru} has even length. Hence, we may augment and M remains a matching. Afterwards, we either set $T = \emptyset$ or $T = (\{r'\}, \emptyset)$ for some new vertex $r' \in V$, hence the invariant

is maintained. If v is M-covered, we extend the tree T by the edges $\{u, v\}$ and $\{v, z\}$ with $\{u, v\} \notin M$ and $\{v, z\} \in M$. Hence, T remains an alternating tree and the invariant is maintained.

If the algorithm terminates successfully, then $T = \emptyset$ and, by the invariant, M is a matching. The only way that T can be set to $\emptyset$ is if no M-exposed vertex exists in MAKETREE. But then M is a perfect matching.

Finally, if the algorithm fails, then no edge $\{u, v\} \in E$ exists with $u \in T_{\text{even}}$ and $v \in V \setminus T$. Also, there cannot be any edge in $G[T_{\text{even}}]$, since such an edge would close an odd cycle, which would be a contradiction with G being bipartite. Hence, every edge in $G[T] \cup \delta_G(T)$ must contain a vertex of T_{odd}, i.e., T_{odd} is a vertex cover of $G[T] \cup \delta_G(T)$. By Observation 5.17, there is no perfect matching, hence the algorithm legitimately fails. □

To generalize EDMONDSPERFECTBIPARTITE to general graphs, we need to address edges between vertices in T_{even}. As observed above, such edges close a unique odd cycle in T. Restricted to this odd cycle, the current matching covers all vertices except for the unique vertex of the cycle closest to the root (see Figure 5.5). We can formalize this as follows.

Definition 5.19. A *blossom* w.r.t. a matching M in G is a factor critical subgraph $C = (V_C, E_C) \subseteq G$ with $|M \cap E_C| = (|V_C| - 1)/2$. The *(blossom) basis* of C is the unique $(M \cap E_C)$-exposed vertex of C.

The key idea now is to *contract* the corresponding blossom C whenever we find an edge between two vertices of T_{even}, i.e., to shrink it to a single vertex v_C and to replace the endpoints in C of all edges in $\delta(C)$ by v_C (see Figure 5.5). In the resulting graph, we eliminated the critical edge, and we can continue as before. Once we find a maximum matching in the contracted graph, we can easily recover a maximum matching in the original graph. This key insight is called the *blossom lemma*.

Lemma 5.20 (blossom lemma)**.** Let $C = (V_C, E_C)$ be a blossom with respect to matching M in G, and let G', M' be obtained from G, M, respectively, by contraction of C. Let further Q be an M-alternating path of even length in G from an M-exposed vertex to the basis b of C, such that $Q \cap C = (\{b\}, \emptyset)$. Then, M is a maximum matching in G if and only if M' is a maximum matching in G'.

Proof. Let M' *not* be a maximum matching in G'. We can extend every matching in G' to a matching in G with $\frac{|V_C|-1}{2}$ additional edges, since, by definition, C is factor critical. For maximum matchings $M'^\star$ in G' and $M^\star$ in G it therefore holds that

$$|M^\star| \geq |M'^\star| + \frac{|V_C| - 1}{2} > |M'| + \frac{|V_C| - 1}{2} = |M|.$$

This means that M is not a maximum matching in G.

Conversely, let M *not* be a maximum matching in G and consider $N := M \triangle Q$. Since Q has even length, this matching has the same cardinality as M and is therefore not maximum. Also, Q having even length implies that it has exactly one M-exposed endpoint, hence b is M-covered and therefore N-exposed. Because N is not maximum, there is an N-augmenting path P in G (Theorem 5.3). Since C contains only a single N-exposed vertex, P has an endpoint $v \notin C$. Let $\bar{P}$ be the maximal subpath of P that contains this end v and contains no vertex of C in its interior. If we contract C, as b is N-exposed in G, the resulting vertex is N'-exposed in G', where N' denotes N after contracting C. Let Q' be the path in G' obtained from Q by contracting C. In G', the path $\bar{P}$ corresponds to an N'-augmenting path. Hence, N' is not maximum in G'. But $M' = N' \triangle Q'$ has the same cardinality as N', since Q has even length and $|Q'| = |Q|$. Thus, M' is not maximum either. □

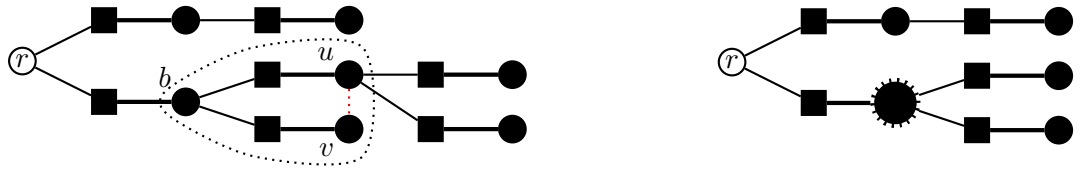

Figure 5.5: Illustration of the operation CONTRACT(u, v). Left: alternating tree before the extension; right: situation after the extension. Note that the alternating tree is maintained.

With this, we can extend EDMONDSPERFECTBIPARTITE to general graphs, as follows. The operation CONTRACT is illustrated in Figure 5.5, and an example of an execution of the algorithm can be found in Figure 5.6. In the following we denote by $V(u)$ the set of vertices of G contained in the (possibly contracted) vertex u.

Algorithm: EDMONDSPERFECT(G)

input: undirected graph $G = (V, E)$
output: perfect matching in G

$M \leftarrow \emptyset$
MAKETREE$(\emptyset)$
$G' = (V', E') \leftarrow G$
while $\exists \{u, v\} \in E'$ *with* $u \in T_{\text{even}}$ **and** $v \notin T_{\text{odd}}$ **:**
  if $v \notin T$ **and** $v \notin M$ **:**
    AUGMENT(u, v)
    $M \leftarrow$ EXPAND$(M, \emptyset)$
    $G' \leftarrow G$
    MAKETREE$(\emptyset)$
  else if $v \notin T$ **and** $v \in M$ **:**
    EXTENDTREE(u, v)
  else
    CONTRACT(u, v)
if $T = \emptyset$ **:**
  return M
else
  fail

Algorithm: CONTRACT$(C, G = (V, E))$

$E_c \leftarrow \{e \in E | e \cap C = \emptyset\}$
$E_c \leftarrow E_c \cup \{\{v_C, v\} \mid v \in \Gamma(C)\}$
$V_c \leftarrow (V \setminus C) \cup \{v_C\}$
return (V_c, E_c)

Algorithm: CONTRACT(u, v)

$C \leftarrow (v\text{-}u\text{-path in } T) \oplus (v)$
$G' \leftarrow$ CONTRACT(C, G')
$M \leftarrow$ CONTRACT(C, M)
$T \leftarrow$ CONTRACT(C, T)

Algorithm: EXPAND(M, R)

while $\exists v_C \in V' \setminus R$ *with* v_C *contracted* **:**
  $\{v\} \leftarrow \Gamma_M(v_C)$
  let $u \in C$ s.t. $\exists e \in \delta_G(V(u)) \cap \delta_G(V(v))$
  $M' \leftarrow$ max. matching in $C \setminus \{u\}$
  $M \leftarrow (M \setminus \delta_M(v_C)) \cup M' \cup \{e\}$
  $V' \leftarrow V' - v_C + C$
return M

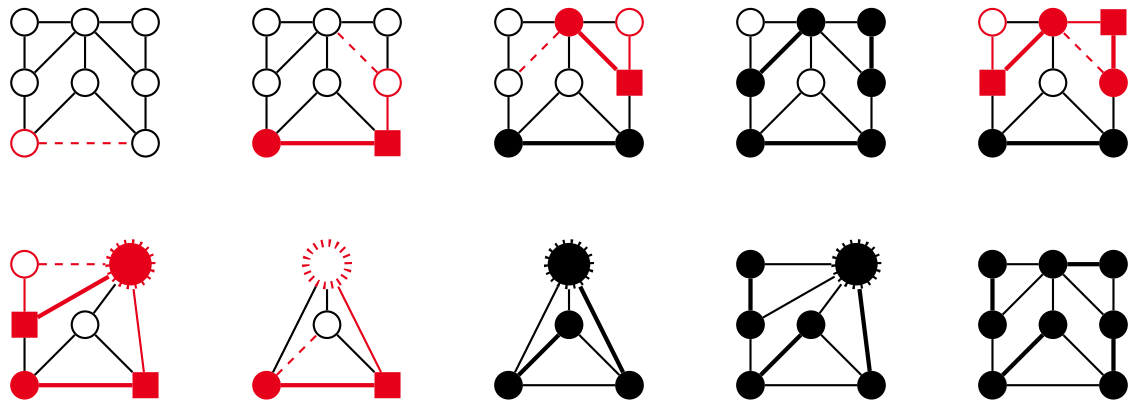

Figure 5.6: Example of an execution of EDMONDSPERFECT.

We show that this algorithm works in general to find perfect matchings.

Proposition 5.21. EDMONDSPERFECT is correct.

Proof. Again, the algorithm maintains the invariant that M is a matching in G' and that T is an alternating tree with root r. It is maintained during augmentations and extensions of T, as in the proof of Proposition 5.18. It is also maintained when a blossom C is contracted or expanded, since the basis of C is the unique vertex of C that may be part of an edge in $\delta_{G'}(C) \cap M$ (unless it coincides with r).

If the algorithm terminates successfully, we have $T = \emptyset$, which means that the last operation was MAKETREE, there is no M-exposed vertex, and $G' = G$. By the invariant, M is a perfect matching in G.

If the algorithm fails, all edges in $G'[T] \cup \delta_{G'}(T)$ have one end in T_{odd}. Then $q_{G'}(T_{\text{odd}}) \geq |T_{\text{even}}| > |T_{\text{odd}}|$, as in Observation 5.17, and G' does not have a perfect matching by Theorem 5.14. Observe that blossom bases always fall in T_{even}, thus T_{odd} does not contain contracted vertices and $T_{\text{odd}} \subseteq V$. Since blossoms are odd, expanding a contracted blossom maintains the parity of the corresponding component in $G - T_{\text{odd}}$. Hence, $q_G(T_{\text{odd}}) > |T_{\text{odd}}|$ and G does not have a perfect matching by Theorem 5.14. Hence, the algorithm legitimately fails. □

Finally, we extend the algorithm further to settings where a perfect matching need not exist. The key idea is to store completed trees, and to start new trees, until all exposed vertices are part of some tree. This yields Edmonds' famous blossom algorithm, which is considered to be one of the most beautiful algorithms in combinatorial optimization, due to the various key ideas that it combines in elegant fashion.

Algorithm: EDMONDS(G)

input: undirected graph $G = (V, E)$
output: maximum cardinality matching in G

$M \leftarrow \emptyset$, $R \leftarrow \emptyset$, MAKETREE$(\emptyset)$, $G' = (V', E') \leftarrow G$
while $\exists r \notin R$ *with* $r \notin M$ **:**
  MAKETREE(R)
  while $\exists \{u, v\} \in E'$ *with* $u \in T_{\text{even}}$ **and** $v \notin T_{\text{odd}} \cup R$ **:**
    if $v \notin T$ **and** $v \notin M$ **:**
      AUGMENT(u, v)
      $M \leftarrow$ EXPAND(M, R)
      while $\exists v_C \in V' \setminus R$ *with* v_C *contracted* **:**
        $V' \leftarrow V' - v_C + C$
      $G' \leftarrow R \cup G[V' \setminus R]$, $T \leftarrow \emptyset$
      break inner loop
    else if $v \notin T$ **and** $v \in M$ **:**
      EXTENDTREE(u, v)
    else
      CONTRACT(u, v)
  $R \leftarrow R \cup T$
return EXPAND$(M, \emptyset)$

We show that Edmonds' algorithm computes a maximum matching in general graphs.

Theorem 5.22. EDMONDS is correct.

Proof. We show that M is a maximum matching in G' upon termination of the algorithm. Since blossom bases always fall in T_{even}, by the blossom lemma (Lemma 5.20), it follows that M expanded to G is also maximum. We first observe that, during each update of R, the vertices of an alternating tree are added to R. These vertices remain unchanged afterwards, therefore the graph $G' = (V', E')$ contains all these trees $T_1, T_2, \dots$ at the end. Every tree contains exactly one exposed vertex, namely its root. Let V_{even} denote the set of vertices in T_{even} of all these trees $T \in \{T_1, T_2, \dots\}$, and let V_{odd} be defined analogously. There are no edges between vertices in T_{even} of a single tree, otherwise a blossom could have been contracted. Also there are no edges between vertices in V_{even} of different trees, otherwise the first of these trees could have been extended further. Now, if we remove all vertices of V_{odd} from G', we obtain $|V_{\text{even}}|$ components consisting of a single vertex each. If k is the number of trees, then $|V_{\text{even}}| = |V_{\text{odd}}| + k$. Hence, $q_{G'}(V_{\text{odd}}) = |V_{\text{even}}| = |V_{\text{odd}}| + k$. Since every M-exposed vertex of G' is contained in some tree, and since every tree contains exactly one M-exposed vertex, we have

$$|M| = \frac{|V'| - k}{2}.$$

By the Theorem of Berge-Tutte (Theorem 5.15) for G', it holds that

$$|M'^{\star}| = \frac{|V'| - \max_{X \subseteq V}\{q_{G'}(X) - |X|\}}{2} \leq \frac{|V'| - q_{G'}(V_{\text{odd}}) + |V_{\text{odd}}|}{2} = \frac{|V'| - k}{2} = |M|,$$

hence M is maximum in G'. □

To implement EDMONDS efficiently, we need to maintain contracted components and to maintain $\delta(T_{\text{even}})$. The former can be achieved using a union-find data structure (see lecture *Algorithmic Discrete Mathematics*), and the latter can be achieved by explicitly maintaining a list L that contains all edges in $\delta(T_{\text{even}})$ that have not yet been considered.

Algorithm: EDMONDS(G) (implementation)

input: undirected graph $G = (V, E)$
output: maximum cardinality matching in G

$M \leftarrow \emptyset$, $R \leftarrow \emptyset$
MAKETREE$(\emptyset)$
while $\exists r \notin R$ *with* $r \notin M$**:**
  MAKETREE(R)
  $L \leftarrow \{(r, r_v) \mid v \in \Gamma_G(r)\}$
  while $\exists (u, v) \in L$**:**
    $L \leftarrow L \setminus \{(u, v)\}$ (extract first)
    if *FIND*$(u) \neq$ *FIND*(v) **and** $v \notin T_{\text{odd}}$**:**
      if $v \notin T$ **and** $v \notin M$**:**
        AUGMENT(u, v)
        $M \leftarrow$ EXPAND(M, R)
      else if $v \notin T$ **and** $v \in M$**:**
        EXTENDTREE(u, v)
      else
        CONTRACT(u, v)
  $R \leftarrow R \cup T$
return M

Algorithm: EXTENDTREE(u, v) (impl.)

$\{z\} \leftarrow \Gamma_M(v)$
$T \leftarrow T + \{u, v\} + \{v, z\}$
$T_{\text{odd}} \leftarrow T_{\text{odd}} \cup \{v\}$
$T_{\text{even}} \leftarrow T_{\text{even}} \cup \{z\}$
for $w \in \{r_{w'} \mid w' \in \Gamma_G(z)\} \setminus \{v\}$**:**
  $L \leftarrow L \oplus \{(z, w)\}$

Algorithm: CONTRACT(u, v) (impl.)

$C \leftarrow (v\text{-}u\text{-path in } T) \oplus (v)$
for $\{a, b\} \in C$**:**
  UNION(a, b)

Algorithm: EXPAND(M, R)

for $v \in M$ *with* $v \notin R$**:**
  $r_v \leftarrow v$

We remark (without proof) that a union-find data structure can be implemented efficiently. Recall that $\alpha(m)$ is the inverse Ackermann function that grows extremely slowly (e.g., $\alpha(9876!) = 5$).

Proposition 5.23. A union-find data structure can be implemented such that every sequence of $\mathcal{O}(m)$ operations takes time $\mathcal{O}(m \cdot \alpha(m))$.

With this, we obtain a very efficient implementation of Edmonds' algorithm.

Theorem 5.24. EDMONDS can be implemented with a running time of $\mathcal{O}(nm \cdot \alpha(m))$.

Proof. Since every augmentation increases the size of the matching, there can be at most $n/2 = \mathcal{O}(n)$ augmentations. We consider the operations between two augmentations. Obviously, every edge is added to L at most once. It follows that we have $\mathcal{O}(m)$ FIND and $\mathcal{O}(n)$ UNION operations. The total running time of these operations is $\mathcal{O}(m \cdot \alpha(m))$ if we use a suitable union-find data structure, by Proposition 5.23.

We have $\mathcal{O}(n)$ EXTENDTREE operations that consider each edge at most twice overall. The total running time for this is $\mathcal{O}(m)$.

In every contraction, the number of vertices is reduced by the size of the contracted blossom minus 1 and the running time required is linear in the size of the blossom. The total running time for contractions therefore is $\mathcal{O}(n)$ plus the running time of the involved UNION operations.

Overall, the running time is $\mathcal{O}(nm \cdot \alpha(m))$, assuming that we implement trees by storing predecessors for each node. □