1
Bridging Continuous and Discrete Optimization

A large part of algorithm design is concerned with problems that optimize or enumerate over discrete structures such as paths, trees, cuts, flows, and matchings in objects such as graphs. Important examples include the following:

(i) Given a graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$, find a **flow** on the edges of G of maximum value from s to t while ensuring that each edge has at most one unit flow going through it.
(ii) Given a graph $G = (V, E)$, find a **matching** of maximum size in G.
(iii) Given a graph $G = (V, E)$, count the number of **spanning trees** in G.

Algorithms for these fundamental problems have been sought for more than a century due to their numerous applications. Traditionally, such algorithms are **discrete** in nature, leverage the rich theory of **duality** and **integrality**, and are studied in the areas of algorithms and combinatorial optimization; see the books by Dasgupta et al. (2006), Kleinberg and Tardos (2005), and Schrijver (2002a). However, classic algorithms for these problems have not always turned out to be fast enough to handle the rapidly increasing input sizes of modern-day problems.

An alternative, **continuous** approach for designing faster algorithms for discrete problems has emerged. At a very high level, the approach is to first formulate the problem as a convex program and then develop continuous algorithms such as gradient descent, the interior point method, or the ellipsoid method to solve it. The innovative use of convex optimization formulations coupled with algorithms that move in geometric spaces and leverage linear solvers has led to faster algorithms for many discrete problems. This pursuit has also significantly improved the state of the art of algorithms for convex optimization. For these improvements to be possible, it is often crucial to abandon an entirely combinatorial viewpoint; simultaneously, fast convergence of continuous algorithms often leverage the underlying combinatorial structure.

1.1 An Example: The Maximum Flow Problem

We illustrate the interplay between continuous and discrete optimization through the $s-t$-maximum flow problem on undirected graphs.

The maximum flow problem. Given an undirected graph $G=(V,E)$ with $n:=|V|$ and $m:=|E|$, we first define the **vertex-edge incidence matrix** $B\in\mathbb{R}^{n\times m}$ associated to it. Direct each edge $i\in E$ arbitrarily and let i^+ denote the head vertex of i and i^- denote its tail vertex. For every edge i, the matrix B contains a column $b_i := e_{i^+} - e_{i^-} \in \mathbb{R}^n$, where $\{e_j\}_{j\in[n]}$ are the standard basis vectors for $\mathbb{R}^n$.

Given $s\neq t\in V$, an $s-t$-flow in G is an assignment $x:\ E\to\mathbb{R}$ that satisfies the following **conservation of flow** property: For all vertices $j\in V\setminus\{s,t\}$, we require that the **incoming** flow is equal to the **outgoing** flow, i.e.,

$$\langle e_j, Bx\rangle = 0.$$

An $s-t$-flow is said to be **feasible** if

$$|x_i|\leq 1$$

for all $i\in E$, i.e., the magnitude of the flow in each edge respects its capacity (1 here). The objective of the $s-t$-maximum flow problem is to find a feasible $s-t$-flow in G that maximizes the flow out of s, i.e., the value

$$\langle e_s, Bx\rangle.$$

The $s-t$-maximum flow problem was not only used to encode various real-world routing and scheduling problems; also many fundamental combinatorial problems such as finding a maximum matching in bipartite graph were shown to be its special cases; see Schrijver (2002a,b) for an extensive discussion.

Combinatorial algorithms for the maximum flow problem. An important fact about the $s-t$-maximum flow problem is that there always exists an **integral** flow that maximizes the objective function. As it will be explained later in this book, this is a consequence of the fact that the matrix B is **totally unimodular**: Every square submatrix of B has determinant 0, 1, or -1. Thus, we can restrict

$$x_i\in\{-1,0,1\}$$

for each $i\in E$, making the search space for the optimal $s-t$-maximum flow discrete. Because of this, the problem has been traditionally viewed as a combinatorial optimization problem.

One of the first combinatorial algorithms for the $s-t$-maximum flow problem was presented in the seminal work by Ford and Fulkerson (1956). Roughly speaking, the **Ford-Fulkerson method** starts by setting $x_i = 0$ for all edges i and checks if there is a path from s to t such that the capacity of each edge on it is 1. If there is such a path, the method adds 1 to the flow value of the edges that point (from head to tail) in the direction of this path and subtracts 1 from the flow values of edges that point in the opposite direction. Given the new flow value on each edge, it constructs a **residual graph** where the capacity of each edge is updated to reflect how much additional flow can still be pushed through it, and the algorithm repeats. If there is no path left between s and t in the residual graph, it stops and outputs the current flow values.

The fact that the algorithm always outputs a maximum $s-t$-flow is nontrivial and a consequence of **duality** – in particular, of the **max-flow min-cut theorem** that states that the maximum amount of flow that can be pushed from s to t is equal to the minimum number of edges in G whose deletion leads to disconnecting s from t. This latter problem is referred to as the $s-t$-minimum cut problem and is the **dual** of the $s-t$-maximum flow problem. Duality gives a way to certify that we are at an optimal solution and, if not, suggests a way to improve the current solution.

It is not hard to see that the Ford-Fulkerson method generalizes to the setting of nonnegative and integral capacities: Now the flow values are

$$x_i \in \{-U, \ldots, -1, 0, 1, \ldots, U\}$$

for some $U \in \mathbb{Z}_{\geq 0}$. However, the running time of the Ford-Fulkerson method in this general capacity case depends linearly on U. As the number of bits required to specify U is roughly $\log U$, this is not a polynomial time algorithm.

Following the work of Ford and Fulkerson (1956), a host of combinatorial algorithms for the $s-t$-maximum flow problem were developed. Roughly, each of them augments the flow in the graph iteratively in an increasingly faster, but combinatorial, manner. The first polynomial time algorithms were by Dinic (1970) and by Edmonds and Karp (1972), who used breadth-first search to augment flows. This line of work culminated in an algorithm by Goldberg and Rao (1998) that runs in $\widetilde{O}\left(m \min\left\{n^{2/3}, m^{1/2}\right\} \log U\right)$ time. Note that unlike the Ford-Fulkerson method, these latter combinatorial algorithms are polynomial time: They find the exact solution to the problem and run in time that depends polynomially on the number of bits required to describe the input. However, since the result of Goldberg and Rao (1998), there was no real progress on improving the running times for algorithms for the $s-t$-maximum flow problem until 2011.

Convex programming-based algorithms. Starting with the paper by Christiano et al. (2011), the last decade has seen striking progress on the $s-t$-maximum flow problem. One of the keys to this success has been to abandon combinatorial approaches and view the $s-t$-maximum flow problem through the lens of continuous optimization. At a very high level, these approaches still maintain a vector $x \in \mathbb{R}^m$ which is updated in every iteration, but this update is dictated by continuous and geometric quantities associated to the graph and is not constrained to be a feasible $s-t$-flow in the intermediate steps of the algorithm. Here, we outline one such approach for the $s-t$-maximum flow problem from the paper by Lee et al. (2013).

For this discussion, assume that we are also given a value F and that we would like to find a feasible $s-t$-flow of value F.[1] Lee et al. (2013) start with the observation that the problem of checking if there is a feasible $s-t$-flow of value F in G is equivalent to determining if the intersection of the sets

$$\{x \in \mathbb{R}^m : Bx = F(e_s - e_t)\} \cap \{x \in \mathbb{R}^m : |x_i| \leq 1,\ \forall i \in [m]\} \qquad (1.1)$$

is nonempty. Moreover, finding a feasible $s-t$-flow of value F is equivalent to finding a point in this intersection. Note that the first set in Equation (1.1) is the set of all $s-t$-flows (a linear space) and the second set is the set of all vectors that satisfy the capacity constraints, in this case the ℓ_∞-ball of radius one, denoted by B_∞, which is a polytope.

Their main idea is to reduce this nonemptiness testing problem to a convex optimization problem. To motivate their idea, suppose that we have convex sets K_1 and K_2 and the goal is to find a point in their intersection (or assert that there is none). One way to formulate this problem as a convex optimization problem is as follows: Find a point $x \in K_1$ that minimizes the distance to K_2. As K_1 is convex, for this formulation to be a convex optimization problem, we need to find a convex function that captures the distance of a point x to K_2. It can be checked that the squared Euclidean distance has this property. Alternatively, one could consider the convex optimization problem where we switch the roles of K_1 and K_2: Find a point $x \in K_2$ that minimizes the distance to K_1. Note here that, while the squared Euclidean distance to a set is a convex function, it is nonlinear. Thus, at this point it may seem like we are heading in the wrong direction. We started off with a combinatorial problem that is a special type of a linear programming problem, and here we are with a nonlinear optimization formulation for it.

[1] Using a solution to this problem, we could solve the $s-t$-maximum flow problem by performing a binary search over F.

Thus the following questions arise: Which formulation should we choose? And why should this convex optimization approach lead us to faster algorithms?

Lee et al. (2013) considered the following convex optimization formulation for the $s-t$-maximum flow problem:

$$\begin{aligned} \min_{x\in\mathbb{R}^m} \quad & \mathrm{dist}^2(x, B_\infty) \\ \text{such that} \quad & Bx = F(e_s - e_t), \end{aligned} \tag{1.2}$$

where $\mathrm{dist}(x, B_\infty)$ is the Euclidean distance of x to the set $B_\infty := \{y \in \mathbb{R}^m : \|y\|_\infty \leq 1\}$. As the optimization problem above minimizes a convex function over a convex set, it is indeed a convex program. The choice of this formulation, however, comes with a foresight that relies on an understanding of algorithms for convex optimization.

A basic method to minimize a convex function is **gradient descent**, which is an iterative algorithm that, in each iteration, takes a step in the direction of the negative gradient of the function it is supposed to minimize. While gradient descent does so in an attempt to optimize the function locally, the convexity of the objective function implies that a local minimum of a convex function is also its global minimum. Gradient descent only requires oracle access to the gradient, or first derivative of the objective function and is, thus, called a **first-order** method. It is really a meta-algorithm and, to instantiate it, one has to fix its parameters such as the step-size and must specify a starting point. These parameters, in turn, depend on various properties of the program including estimates of smoothness of the objective function and those of the closeness of the starting point to the optimal point.

For the convex program in Equation (1.2), the objective function has an easy-to-compute first-order oracle. This follows from the observation that it decomposes into a sum of squared-distances, one for each coordinate, and each of these functions is quadratic. Moreover, the objective function is **smooth**: The change in its gradient is bounded by a constant times the change in its argument; one can visually inspect this in Figure 1.1.

One problem with the application of gradient descent is that the convex program in (1.2) has constraints $\{x \in \mathbb{R}^m : Bx = F(e_s - e_t)\}$ and, hence, the direction gradient descent asks us to move can take us out of this set. A way to get around this is to project the gradient of the objective function onto the subspace $\{x \in \mathbb{R}^m : Bx = 0\}$ at every step and move in the direction of the projected gradient. However, this projection step requires solving a least squares problem that, in turn, reduces to the numerical problem of solving a linear system of equations. While one can appeal to the Gaussian elimination

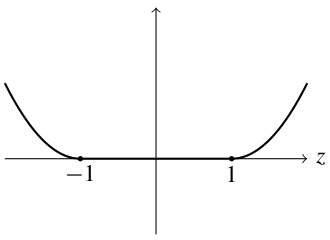

Figure 1.1 The function $\text{dist}^2(z, [-1,1])$.

method for this latter task, it is not fast enough to warrant improvements over combinatorial algorithms mentioned earlier. Here, a major result discovered by Spielman and Teng (2004) implies that such a projection can, in fact, be computed in time $\widetilde{O}(m)$. This is achieved by noting that the linear system that arises when projecting a vector onto the subspace $\{x \in \mathbb{R}^m : Bx = 0\}$ is the same as solving **Laplacian systems** that are of the form $BB^\top y = a$ (for a given vector a), where B is a vertex-edge incidence matrix of the given graph. Such a result is not known for general linear systems and (implicitly) relies on the combinatorial structure of the graph that gets encoded in the matrix B.

Thus, roughly speaking, in each iteration the projected gradient descent algorithm takes a point x_t in the space of all $s-t$-flows of value F, moves toward the set B_∞ along the negative gradient of the objective function, and then projects the new point back to the linear space; see Figure 1.2 for an illustration. While each iterate is an $s-t$-flow, it is not a feasible flow.

A final issue is that such a method may not lead to an exact solution but only an approximate solution. Moreover, in general, the number of iterations depends inverse polynomially on the quality of the desired approximation. Lee et al. (2013) proved the following result: There is an algorithm that, given an $\varepsilon > 0$, can compute a feasible $s-t$-flow of value $(1-\varepsilon)F$ in time $\widetilde{O}(mn^{1/3}\varepsilon^{-2/3})$. If we ignore the ε in their bound, this improved the result of Goldberg and Rao (1998) mentioned earlier.

We point out that the combinatorial algorithm of Goldberg and Rao (1998) has the same running time even when the input graph is directed. It is not clear how to generalize the gradient descent–based algorithm for the $s-t$-maximum flow problem presented above to run for directed graphs.

The results of Christiano et al. (2011) and Lee et al. (2013) were further improved using increasingly sophisticated ideas from continuous optimization and finally led to a nearly linear time algorithm for the undirected $s-t$-maximum flow problem in a sequence of work by Sherman (2013), Kelner et al. (2014), and Peng (2016). Remarkably, while these improvements

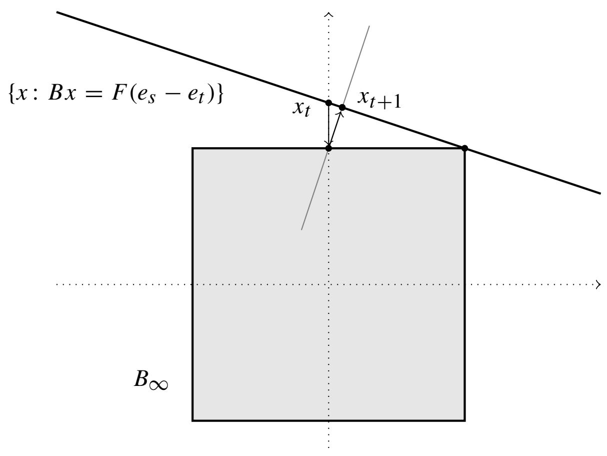

Figure 1.2 An illustration of one step of the projected gradient descent in the algorithm by Lee et al. (2013).

abandoned discrete approaches and used algorithms for convex optimization, beating the running times of combinatorial algorithms leveraged the underlying combinatorial structure of the $s-t$-maximum flow problem.

The goal of this book is to enable a reader to gain an in-depth understanding of algorithms for convex optimization in a manner that allows them to apply these algorithms in domains such as combinatorial optimization, algorithm design, and machine learning. The emphasis is to derive various convex optimization methods in a principled manner and to establish precise running time bounds in terms of the input length (and not just on the number of iterations). The book also contains several examples, such as the one of $s-t$-maximum flow presented earlier, that illustrate the bridge between continuous and discrete optimization. Laplacian solvers are not discussed in this book. The reader is referred to the monograph by Vishnoi (2013) for more on that topic.

The focus of Chapters 3–5 is on basics of convexity, computational models, and duality. Chapters 6–8 present three different first-order methods: gradient descent, mirror descent and multiplicative weights update method, and **accelerated gradient descent.** In particular, the discussion here is presented in detail as an application in Chapter 6. In fact, the fastest version of the method of Lee et al. (2013) uses the accelerated gradient method. Chapter 7 also draws a connection between **mirror descent** and the **multiplicative weights update** method and shows how the latter can be used to design a fast (approximate) algorithm for the bipartite maximum matching problem. We remark that the algorithm of Christiano et al. (2011) relies on the multiplicative weights update method.

Beyond approximate algorithms? The combinatorial algorithms for the $s-t$-maximum flow problem, unlike the first-order convex optimization–based algorithms described above, are exact. One can convert the latter approximate algorithms to exact ones, but it may require setting a very small value of ε making the overall running time non-polynomial. The remainder of the book is dedicated to developing algorithms for convex optimization – **interior point** and **ellipsoid** – whose number of iterations depend **poly-logarithmically** on ε^{-1} as opposed to polynomially on ε^{-1}. Thus, if we use such algorithms, we can set ε to be small enough to recover exact algorithms for combinatorial problems at hand. These algorithms use deeper mathematical structures and more sophisticated strategies (as explained later). The advantage in learning these algorithms is that they work more generally – for linear programs and even convex programs in a very general form. Chapters 9–13 develop these methods, their variants, and exhibit applications to a variety of discrete optimization and counting problems.

1.2 Linear Programming

The $s-t$-maximum flow problem on undirected graphs is a type of linear program: a convex optimization problem where the objective function is a linear function and all the constraints are either linear equalities or inequalities. In fact, the objective function is to maximize the flow value $F \geq 0$ constrained to the set of feasible $s-t$-flows of value F; see Equation (1.1).

A linear program can be written in many different ways and we consider its **standard form**, where one is given a matrix $A \in \mathbb{R}^{n\times m}$, a constraint vector $b \in \mathbb{R}^m$, and a cost vector $c \in \mathbb{R}^n$, and the goal is to solve the following optimization problem:

$$\begin{aligned} &\max_{x\in\mathbb{R}^m} \langle c, x\rangle \\ &\text{such that } Ax = b, \\ &\qquad\qquad x \geq 0. \end{aligned}$$

Typically we assume $n \leq m$ and, hence, the rank of A is at most n. Analogous to the $s-t$-maximum flow problem, linear programming has a rich duality theory, and in particular the following is the **dual** of the above linear program:

$$\min_{y\in\mathbb{R}^n} \langle b, y\rangle$$
$$\text{such that } A^\top y \geq c.$$

Note that the dual is also a linear program and has n variables.

Linear programming duality asserts that if there is a feasible solution to both the linear program and its dual, then the optimal values of these two linear programs are the same. Moreover, it is often enough to solve the dual if one wants to solve the primal and vice versa. While duality has been known for linear programming for a very long time (see Farkas [1902]), a polynomial time algorithm for linear programming was discovered much later. What was special about the $s-t$-maximum flow problem that led to a polynomial time algorithm for it before linear programming?

As mentioned earlier, one crucial property that underlies the $s-t$-maximum flow problem is integrality. If one encodes the $s-t$-maximum flow problem as a linear program in the standard form, the matrix A turns out to be totally unimodular: Determinants of all square submatrices of A are $0, 1$, or -1. In fact, in the case of the $s-t$-maximum flow problem, A is just the vertex-edge incidence matrix of the graph G (which we denoted by B) that can be shown to be totally unimodular. Because of linearity, one can always assume without loss of generality that the optimal solution is an extreme point, i.e., a **vertex** of the polyhedra of constraints (not to be confused with the vertex of a graph). Every such vertex arises as a solution to a system of linear equations involving a subset of rows of the matrix A. The total unimodularity of A, then, along with Cramer's rule from linear algebra, implies that each vertex of the polyhedra of constraints has integral coordinates.

While duality and integrality do not directly imply a polynomial time algorithm for the $s-t$-maximum flow problem, the mathematical structure that enables these properties is relevant to the design of efficient algorithms for this problem. It is worth mentioning that these ideas were generalized in a major way by Edmonds (1965a,b) who figured out an integral polyhedral representation for the **matching problem** and gave a polynomial time algorithm for optimizing linear functions over this polyhedron.

For general linear programs, however, integrality does not hold. The reason is that for a general matrix A, the determinants of submatrices that show up in the denominator of the vertices of the associated polyhedra may not be 1 or -1. However, for A with integer entries, these determinants cannot be more than $\text{poly}(n, L)$ in magnitude, where L is the number of bits required to encode A, b and c. This is a consequence of the fact that determinant of a matrix with integer entries bounded by 2^L is no more than $n!\,2^{nL}$. While there was a

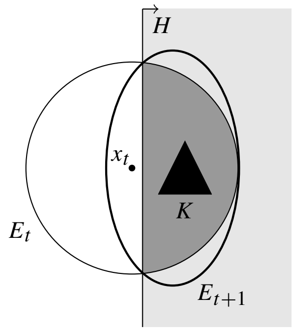

Figure 1.3 Illustration of one step of the ellipsoid method for the polytope K.

combinatorial algorithm for linear programming, e.g., the **simplex method** of Dantzig (1990) that moved from one vertex to another, none was known to run in polynomial time (in the bit complexity) in the worst case.

Ellipsoid method. In the late 1970s, a breakthrough occurred and a polynomial time algorithm for linear programming was discovered by Khachiyan (1979, 1980). The ellipsoid method is a geometric algorithm that checks if a given linear program is feasible or not. As in the case of the $s-t$-maximum flow problem, solving this feasibility problem implies an algorithm to optimize a linear function via a binary search argument.

In iteration t, the ellipsoid method approximates the feasible region of the linear program with an **ellipsoid** E_t and outputs the center (x_t) of this ellipsoid as its guess for a feasible point. If this guess is incorrect, it requires a **certificate** – a hyperplane H that separates the center from the feasible region. It uses this **separating hyperplane** to find a new ellipsoid (E_{t+1}) that encloses the intersection of E_t and the halfspace of H in which the feasible region lies; see Figure 1.3. The key point is that the update ensures that the volume of the ellipsoid reduces at a fast enough rate and only requires solving a linear system of equations to find the new ellipsoid from the previous one. If the **volume** of the ellipsoid becomes so small that it cannot contain any feasible point, we can safely assert the infeasibility of the linear program.

The ellipsoid method belongs to the larger class of **cutting plane methods** as, in each step, the current ellipsoid is cut by an affine halfspace and a new ellipsoid that contains this intersection is determined. The final running time of Khachiyan's ellipsoid method was a polynomial in n, L and, importantly, in $\log \frac{1}{\varepsilon}$: The algorithm output a point $\hat{x}$ in the feasible region such that