Combinatorial Optimization

Frank de Zeeuw

EPFL

2012
Today

- **Introduction**
- **Graph problems** - What combinatorial things will we be optimizing?
- **Algorithms** - What kind of solution are we looking for?
- **Linear Programming** - The main tool that we will use.
- **Example: Matchings**
This course

- **Problem sets:**
  - weekly problem set
  - discussed in class on Tuesday
  - optionally hand in one problem of your choice on Tuesday (not for first set)
  - each handed-in problem counts for 1% of your final grade, if that improves it (i.e. it counts if > final exam grade)

- **Final Exam:** more information when the time comes
Lecture outline (tentative)

1. Introduction
2. Trees
3. Paths
4. Flows
5. Matchings
6. Matchings
7. Integer Programming
8. Matroids
9. Matroids
10. Matroids
11. \(\mathcal{NP}\)-hardness
12. More problems?
13. Travelling Salesman Problem
I will post rough lecture notes online that cover the lectures.

Online notes by others (see links on course website):

- Schrijver (great, but don’t cover everything)
- Plotnik (great, but very different order)
- Several others

Books (several copies in library):

- Korte, Vygen - Combinatorial Optimization (not the easiest to read, least expensive)
- Cook, Cunningham, Pulleyblank, Schrijver - Combinatorial Optimization (easier to read, very expensive)
- Schrijver - Combinatorial Optimization (encyclopedic, 3 big books, very expensive...)
- Matoušek, Gärtner - Understanding and Using Linear Programming (useful if you have it already)
The goal of this course is to look for algorithms that find certain combinatorial objects with optimal value.

Most often, the objects are subgraphs of a graph $G = (V, E)$, usually with a weight function $w : E \rightarrow \mathbb{R}$. Then we want to find a subgraph $H$ of a given type such that its total weight $w(H) = \sum_{e \in H} w(e)$ is larger/smaller than that of any other such subgraph.

Some simple examples:

- **Shortest Path Problem:** Given a weighted directed graph and $a, b \in V$, find a path $P$ from $a$ to $b$ such that $w(P)$ is minimum.
- **Maximum Spanning Tree Problem:** Given an undirected weighted graph, find a spanning tree $T$ such that $w(T)$ is maximum.
Matchings (in undirected graphs)

A matching is an $M \subset E$ such that $\forall m, m' \in M, \ m \cap m' = \emptyset$.

- It is maximal if it is not contained in any larger matching.
- It is maximum if there is no matching with more edges.

**Maximum Matching Problem**
In an unweighted graph, find a maximum matching.

**Maximum Weighted Matching Problem**
In a weighted graph, find a matching $M$ such that $w(M)$ is maximum.
Flows

A network is a directed graph $G = (V, E)$ with two special vertices $s, t$, and a capacity function $c : E \rightarrow \mathbb{R}_+$. Given a network, a flow is a function $f : E \rightarrow \mathbb{R}_+$ such that

- $f(e) \leq c(e) \ \forall e \in E$;
- $\sum_{uv \in E} f(uv) = \sum_{vw \in E} f(vw) \ \forall v \in V - \{s, t\}$.

Maximum Flow Problem

Given a network, find a flow $f$ such that $\sum_{sv \in E} f(sv)$ is maximum.
A Hamilton cycle of a graph is a cycle that goes through all the vertices of the graph.

**Travelling Salesman Problem**

Given a weighted complete graph, find a Hamilton cycle of minimum weight.
Other problems (picked randomly, not exactly involving graphs)

- **Minimum Set Cover Problem**: Given a set $S$ with subsets $S_1, \ldots, S_m$. Find a minimum $I \subset [m]$ such that $\bigcup_{j \in I} S_j = S$.

- **Bin-Packing Problem**: Given a list of numbers $0 \leq a_1, \ldots, a_n \leq 1$, find a minimum $k \in \mathbb{N}$ for which there is an assignment $f : [n] \rightarrow [k]$ with

  $$\sum_{f(i)=j} a_i \leq 1 \quad \forall j \in [k].$$

- ...
Algorithms

We will treat algorithms informally, usually describing them 'mathematically'. For instance, here is an example I will talk about later today:

1. Set $M := \emptyset$.
2. Pick $e \in E$ such that $e \cap m = \emptyset \ \forall m \in M$; if not possible, then stop and return $M$.
3. Set $M := M \cup \{e\}$.
4. Go back to 2.

Note that this is not a precise implementation: It doesn't say how to 'pick' $e$, how to check that the intersection is empty, or how to represent the graph as a data structure. But mathematically those things are not problematic, and we will focus on the mathematical concepts here.
Since we’re not precise about implementation or data structures, we also won’t be precise about “running times” (the worst-case number of steps that the algorithm might take, as a function of the input; for example, the algorithm above has running time $O(|V|)$).

What we will care about most is whether running times are polynomial or not, i.e. if they are of the form $O(|V|^k)$ for some fixed $k$. If it is, we say the algorithm is “polynomial”, or that it is “in $\mathcal{P}$”.

In combinatorics, naive algorithms are often “exponential”, i.e. not polynomial. For instance, if it runs over all subsets of the vertices, its running time would be $O(2^{|V|})$. 
We introduce, very roughly, several notions related to polynomial running times. The first are properties of decision problems (YES/NO questions). A decision problem is

- in $\mathcal{P}$ if there is an algorithm that solves any instance in polynomial time;
- in $\mathcal{NP}$ if, when the answer to an instance is YES, there is a certificate proving it that can be checked in polynomial time;
- $\mathcal{NP}$-complete if any problem in $\mathcal{NP}$ can be polynomially reduced to it.

Also, a calculation problem (question whose answer can be represented as a number, including any optimization problem) is

- $\mathcal{NP}$-hard if any $\mathcal{NP}$-complete problem can be polynomially reduced to it.
You should be aware of the following $\mathcal{NP}$-facts:

- $\mathcal{NP}$ does not stand for “non-polynomial” (and it doesn’t mean that), but for “non-deterministically polynomial”.

- $\mathcal{P} \subset \mathcal{NP}$. To become a millionaire, prove $\mathcal{P} \neq \mathcal{NP}$ or the opposite.

- Deciding whether or not a graph has a Hamilton Cycle is $\mathcal{NP}$-complete.

- Travelling Salesman, Set Cover, Bin-Packing are all $\mathcal{NP}$-hard.

- Linear Programming is in $\mathcal{P}$, but Integer Programming is $\mathcal{NP}$-hard.
Approximation algorithms

- Consider a general minimization problem, asking for a feasible object $S$ with minimum value $v(S)$. Write $S^*$ for an optimum solution.

- A $k$-approximation algorithm (with $1 \leq k \in \mathbb{R}$) for a minimization problem is a polynomial algorithm that returns a feasible object $S$ such that

$$v(S) \leq k \cdot v(S^*).$$

- A $k$-approximation algorithm for a maximization problem is a polynomial algorithm that returns a feasible object $S$ such that

$$v(S) \geq \frac{1}{k} \cdot v(S^*).$$
A Linear Programming problem looks like

\[
\text{maximize } c^T x \quad \text{subject to}
\]
\[
Ax \leq b \quad \text{and} \quad x \geq 0
\]

with \( x \in \mathbb{R}^n \) a vector of variables, \( c \in \mathbb{R}^n \), \( b \in \mathbb{R}^m \), and \( A \in \mathbb{R}^{m \times n} \). Different versions are possible:

- “minimize” could be “maximize”
- \( Ax \leq b \) could be \( Ax \geq b \) or \( Ax = b \)
- \( x \geq 0 \) could be \( x \leq 0 \), or just \( x \in \mathbb{R}^n \)
- any mix of the above
Duality

Suppose $x$ satisfies $Ax \leq b$, or in rows $b_i \geq \sum_j A_{ij}x_j$. Then for any $y_i \geq 0$ also

$$
\sum_i y_i b_i \geq \sum_i y_i \left( \sum_j A_{ij}x_j \right) = \sum_i \sum_j A_{ij}x_i y_j = \sum_j x_j \left( \sum_i A_{ij}y_i \right).
$$

If $\sum_i A_{ij}y_i \geq c_j$, then $\sum_i y_i b_i \geq \sum_j c_j x_j$, so $b^T y \geq \max c^T x$. This gives the dual LP problem (to the primal problem we started with):

$$
\begin{align*}
\text{minimize} & \quad b^T y \\
\text{subject to} & \quad A^T y \geq c \text{ and } y \geq 0
\end{align*}
$$

Note that we need $y \geq 0$, otherwise the inequalities would have flipped.
**Duality**

**Primal:**

\[
\text{maximize } c^T x \quad \text{subject to} \quad Ax \leq b \quad \text{and} \quad x \geq 0
\]

**Dual:**

\[
\text{minimize } b^T y \quad \text{subject to} \quad A^T y \geq c \quad \text{and} \quad y \geq 0
\]

By the previous slide, if \( x \) is a primal feasible solution and \( y \) a dual feasible solution, then \( c^T x \leq b^T y \) (this is called weak duality).

**Duality Theorem:**

\( x^* \) is a primal optimal solution and \( y^* \) is a dual optimal solution if and only if \( c^T x^* = b^T y^* \).

Note that if the primal looks different, so will the dual. For instance, if the primal has \( Ax = b \), then the dual does not have \( y \geq 0 \) (\( y \) is free). To avoid memorizing all the different combinations, you should be able to redo the previous slide.
Theorem

Feasible $x$ and $y$ are optimal solutions to the primal and dual problems in the forms above if and only if

$$\forall j : \left( \sum_i A_{ij}x_i = b_j \text{ or } y_j = 0 \right).$$

In other words, either the $j$th primal constraint is tight, or $y_j = 0$. By symmetry, something similar holds for the dual constraint.
The most natural algorithm for solving LP problems is the simplex algorithm. It uses that $Ax \leq b$ describes a polytope in $\mathbb{R}^n$, and optimal solutions are among the vertices of the polytope.

In the algorithm, you start at any vertex, then repeatedly move along an edge to a better vertex, eventually ending up at an optimal one.

But the simplex algorithm does not have a polynomial running time. (actually, some but not all versions are proven to be non-polynomial, while no version is known to be polynomial)

There are other algorithms, like ellipsoid algorithms, which are proven to be polynomial. So solving linear programs is in $\mathcal{P}$.

In this course, we will treat these algorithms as “black boxes”, i.e. we should be aware of the fact that they exist, but we won’t look any closer at them.
An Integer Programming problem looks like

$$\text{maximize } c^T x \quad \text{subject to} \quad Ax \leq b, \ x \geq 0, \ \text{and} \ x \in \mathbb{Z}^n$$

- Solving IP problems is $\mathcal{NP}$-hard. So unless $\mathcal{P} = \mathcal{NP}$, there is no polynomial algorithm for it.
- There is no duality theorem for an IP problem (directly), but weak duality still holds.
- The *relaxation* of an IP problem is the LP problem you get by removing the condition $x \in \mathbb{Z}^n$. An optimum for the relaxation provides a bound for the IP optimum, but usually there is a gap.
All the combinatorial optimization problems that we will see are in fact IP problems.

For some problems, we can prove that the relaxation has the same optima as the IP problem (we say the polytope is \textit{integral}), which immediately implies that there is a polynomial algorithm.

We’ll still study these more closely, to look for nicer algorithms than an indirect LP algorithm, and because we’ll learn much that we’ll need for harder problems.
Main tools

The techniques that we will use to find algorithms for combinatorial optimization are, roughly speaking:

- **Greedy heuristic** (matroids)
- **Linear Programming** (especially duality)
- **Graph Theory** (a.k.a. insight...)
Greedy Algorithm: Repeatedly add any edge that you can add without losing the matching property. More formally:

1. Set $M := \emptyset$.
2. Pick $e \in E$ such that $e \cap m = \emptyset \ \forall m \in M$; if not possible, then stop and return $M$.
3. Set $M := M \cup \{e\}$.
4. Go back to 2.

$M$ will be maximal, but need not be maximum:

- Is it a $k$-approximation for some $k$?
Integer programming formulation of the Maximum Matching Problem:

\[
\begin{align*}
\text{max} \quad & \sum_{e \in E} x_e, \text{ with } x \geq 0, \ x \in \mathbb{Z}^{|E|} \\
\text{and} \quad & \sum_{e \ni v} x_e \leq 1 \quad \forall v \in V 
\end{align*}
\]

Dual:

\[
\begin{align*}
\text{min} \quad & \sum_{v \in V} y_v, \text{ with } y_v \geq 0, \ y \in \mathbb{Z}^{|V|} \\
\text{and} \quad & y_u + y_v \geq 1 \quad \forall uv \in E
\end{align*}
\]

Then if \( y \) is optimal, we must have \( y_v \in \{0, 1\} \), and the set \( C = \{ v \in V : y_v = 1 \} \) is what is called a vertex cover: a set of vertices that touch every edge.
By weak duality, a primal feasible solution \( \{x_e\} \) and a dual feasible solution \( \{y_v\} \) always satisfy \( \sum x_e \leq \sum y_v \). In other words, any matching \( M \) and cover \( C \) satisfy \(|M| \leq |C|\). (which you can easily prove without duality)

Any maximal matching \( M \) easily gives you a cover:
\[
C = \bigcup_{m \in M} m, \text{ with } |C| = 2|M|.
\]

Proof: If \( C \) is not a cover, then \( M \) is not maximal.

Therefore if \( M^* \) is a maximum matching, then we have
\[
|M^*| \leq |C| = 2|M|,
\]
so \(|M| \geq |M^*|/2\), which means that \( M \) is a 2-approximation.
Bipartite minimum matching

Can we find an exact algorithm for the Minimum Matching Problem? Let’s try it for bipartite graphs.

- If the algorithm is not greedy, it must have some kind of backtracking followed by improvement, like:

  \[
  \begin{array}{c}
  \text{Original Matching} \\
  \text{Improved Matching}
  \end{array}
  \]

- More generally, we can do this for any augmenting path:

  \[
  \begin{array}{c}
  \text{Original Matching} \\
  \text{Improved Matching}
  \end{array}
  \]
Bipartite minimum matching

This suggests an algorithm:

Find an augmenting path, then augment on it to get a larger matching; repeat.

This will work because:

**Theorem**

If a matching in a bipartite graph $G = (V, E)$ is not maximum, then it has an augmenting path.
Bipartite minimum matching

**Theorem**

If a matching $M$ in a bipartite graph is not maximum, then it has an augmenting path.

*Proof:* Suppose $M'$ is a matching with $|M'| > |M|$. Consider $G' = (V, M \cup M')$.

We have

$$\deg_{G'}(v) = \deg_M(v) + \deg_{M'}(v) \leq 1 + 1 = 2,$$

which implies that all components of $G'$ are paths or cycles. These must have alternating $M$-edges and $M'$-edges. By bipartiteness, all cycles are even, so have as many $M$-edges as $M'$-edges. Since $|M'| > |M|$, one of the paths must have more edges from $M'$ than from $M$, which means it’s an augmenting path. □
Bipartite minimum matching

Find an augmenting path, then augment on it to get a larger matching; repeat.

This is not quite an algorithm, because we haven’t said how to find an augmenting path.
Here’s how. Given a maximal-but-not-maximum matching in a bipartite graph, the graph will look like
If we define a digraph with the edges from $S$ downward, matching-edges upward, and the others downward, like this:

Then a dipath from $S$ to $T$ will be an augmenting path. The first homework will ask you for such an algorithm.