Linear and Combinatorial Optimization

Sara Maad Sasane

Center for Mathematical Sciences, Lund University

26 February 2019

1. Algorithm complexity
2. Problem classification
3. Algorithm complexity for min cost flow problems and related problems

Algorithm complexity

Definition

A problem is decidable *if there is an algorithm that solves the problem in finite time.*

Example

The travelling salesman problem is decidable, for example through exhaustive search.

In fact, all combinatorial optimization problems are decidable, since their feasible set is always finite.

Algorithm complexity

- We can talk about two different concepts of complexity for a given algorithm:

  Time complexity: How long does it take to run the algorithm? and how does this time grow when the size of the problem gets larger ($\to \infty$)?

  Memory complexity: How much memory does the algorithm need? and how does the memory size grow when the size of the problem gets larger ($\to \infty$)?

- Since it takes time to uses memory, the memory complexity is in general not worse than the time complexity. We will only talk about time complexity for this reason.

Algorithm complexity

When measuring the time complexity, the *ordo* notation is useful. n represents the size of the problem (e.g. the number of indata entries).

Definition (Little ordo)

$f(n) = o(g(n))$ *as* $n \to \infty$ *if*

$$\frac{f(n)}{g(n)} \to 0 \quad \text{as } n \to \infty.$$

Example

- $n^2 = o(n^3)$ *as* $n \to \infty$ *and*
- $n^k = o(2^n)$ *for every (fixed)* k.

Algorithm complexity

Definition (Big Ordo)

$f(n) = \mathcal{O}(g(n))$ *as* $n \to \infty$ *if*

$$\frac{f(n)}{g(n)} \quad \textit{is bounded for } n \geq 1.$$

Example

- $2n^2 + n + 3 = \mathcal{O}(n^2)$ *as* $n \to \infty$.

Algorithm complexity

We will make the following simplifying assumptions when talking about (theoretical) time complexity:

Basic assumption: All sorts of operations require unit time.

We let

- n be the size of the problem (i.e. the size of the matrix A in an LP problem),
- $f(n)$ be the time it takes to solve the problem with the algorithm for a problem of size n.

Note that it is the worst case scenario that is (usually) considered, and so $f(n)$ is the largest time it could take to solve a problem of size n, using the given algorithm.

Algorithm complexity

We consider different classes of algorithm, depending on how long time it takes to solve them. The most important such classes are:

Polynomial time algorithms: $f(n) = \mathcal{O}(n^k)$ for some (fixed) k.

Exponential time algorithms: $f(n) = 2^{\mathcal{O}(n)}$ as $n \to \infty$, which means that there exists a function g, defined on the positive integers, such that $f(n) = 2^{g(n)}$ and $g(n) = \mathcal{O}(n)$ as $n \to \infty$.

Algorithm complexity

- There are two related types of problems which are relevant to us. These are

  Optimization problems: Find a feasible x^* which is optimal.
  Example: LP.

  Feasibility problems: Does there exist a feasible solution?
  Example: LI (Linear Inequalities)

- From the point of view of time complexity, LP and LI are equivalent:

  LP $\Rightarrow$ LI: Solve LI by considering LP with any objective function (e.g. $z \equiv 0$).

  LI $\Rightarrow$ LP: Consider the dual problem. Solve the primal and dual problem simultaneously as an LI problem. Note that the sizes of these problems are of the same order. Feasible solution $\Rightarrow$ Primal and dual solutions are optimal.

1. Algorithm complexity

2. **Problem classification**

3. Algorithm complexity for min cost flow problems and related problems

Problem classification

- The feasibility version of a problem is usually studied when determining a problem's degree of complexity.

  Class P: A feasibility problem is in class P if there exists a polynomial time algorithm for finding its solution.

  Class NP: There exists a polynomial time algorithm for checking whether a point is a (feasible) solution.

- Clearly $P \subseteq NP$ (since an algorithm for solving a problem must have a way of checking that it has found a solution).
- The question then arises whether there are problems that are in NP but not in P. Nobody knows the answer to this question, and there is a prize of \$1,000,000 for solving the problem. You can read more about the Millennium Prize problems.

Problem reduction and *NP* completeness

- Certain problems are related so that if there is a fast algorithm for one problem in the class, one can convert the problem so that also the other problems of the same class can be solved by an algorithm of the same complexity.
- For this to be useful, the complexity for converting one of the problems to another has to be at least as good (big $\mathcal{O}$) as the complexity of the algorithm that was found.
- There is a class of problems of particular interest, called *NP-complete problems* which are the hardest problem in *NP*. This class has the property that if one of its problems can be solved in polynomial time, then all of them can be solved in polynomial time. On the other hand, nobody knows if they can be solved in polynomial time or not (i.e. it is not known if they belong to *P*).

Problem reduction and *NP*-completeness

To define the *NP*-complete problems, we first define another class of problems: the *NP*-hard problems.

Definition

- *A problem H is NP-hard if every problem in NP can be reduced in polynomial time to H.*
- *A problem is NP-complete if it belongs to NP and is NP-hard.*

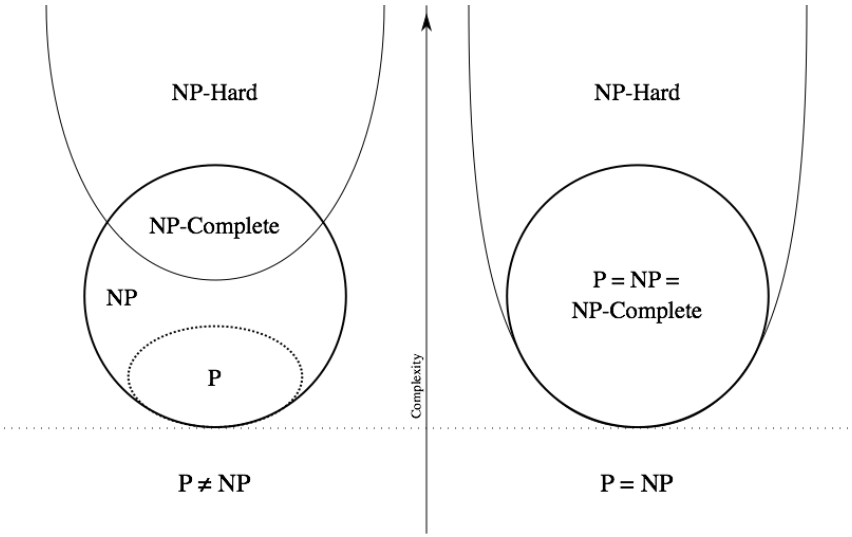

Figure 1: There are two different scenarios depending on whether or not $P = NP$. Most experts believe that $P \neq NP$ (left part of the figure).

Problem reduction and *NP*-completeness

Example (Examples of *NP*-complete problems)

- *The decision version of the travelling salesman problem is NP-complete.(By the decision version of TSP we mean the problem of finding a certain path with total length less than or equal to a given number.)*
- *The optimization version of TSP is NP-hard but* not *NP-complete in general, but*
- *TSP where the edge lengths are assumed to be integers is NP-complete.*
- *The Knapsack Problem is NP-complete.*
- 0–1 *Linear Programming is NP-complete.*
- *Integer linear programming is NP-complete.*

Problem reduction and *NP*-completeness

Theorem

The simplex method is not a polynomial time algorithm:

- *For every $d > 1$, there is an LP problem with $2d$ equations, $3d$ variables and integer coefficients of size ≤ 4 such that the simplex method will need $2^d - 1$ iterations to find the optimum.*
- *Furthermore, for every pivoting rule in the simplex algorithm, it is possible to construct a problem which takes exponential time to solve with the simplex algorithm.*

On the other hand, the simplex method still perform well in practice for most problems. Remember that algorithm complexity is about the time it takes to solve a problem in the worst case.

Problem reduction and *NP*-completeness

Question: Does *LP* belong to *NP*?
Answer: Yes, it is easy to check that the linear inequalities are satisfied.

- Let $m = \mathcal{O}(d)$ and $n = \mathcal{O}(d)$ as $d \to \infty$, where m is the number of inequality constraints and n is the number of variables.
- The number of operations needed for checking that a solution is feasible and optimal is:

  Primal inequalities: $m \times (n + (n-1) + 1) = 2mn = \mathcal{O}(d^2)$,
  Dual inequalities: $n \times (m + (m-1) + 1) = 2mn = \mathcal{O}(d^2)$,
  Dual obj.$\leq$ Primal obj.:

  $$n + m + (n + m - 1) + 1 = 2(n + m) = \mathcal{O}(d).$$

  Totally there are $\mathcal{O}(d^2)$ operations, and hence there is a polynomial time algorithm for verifying that a solution is optimal (and feasible).

Problem reduction and *NP*-completeness

To summarize, we have seen that

- the simplex method is *not* a polynomial time algorithm, but
- LP belongs to the class *NP*.

This does *not* show that $P \neq NP$, since there are other algorithms for LP which are of polynomial time. Some of these are:

- **The ellipsoid method (Khachiyan, 1979):** The first polynomial time algorithm found for *LP*. Only of theoretical importance since the algorithm takes too long for practical problems to be useful. It is $\mathcal{O}(n^4)$.
- **Karmarkar's algorithm (Karmarkar, 1984):** This was the first polynomial time algorithm that was practically useful for solving actual problems. It is rather complicated to learn and it is $\mathcal{O}(n^{3.5})$ as $n \to \infty$.
- **Central path algorithms (popular since the 1990's):** The best algorithms seem to be $\mathcal{O}(n^3)$.

The ellipsoid method

- We will present the idea of the ellipsoid method.
- The algorithm solves the feasibility problem, so first the LP problem has to be rephrased as a feasibility problem using the primal together with the dual.
- We will consider a general feasibility problem of Linear Programming, i.e. a problem of Linear Inequalities.

The ellipsoid method

Algorithm idea:

1. Start with a large ellipsoid which contains the feasible set.
2. Is the centre of the ellipsoid feasible?
   - If yes, then stop.
   - If no, then construct a hyperplane through this point so that half of the ellipsoid contains the feasible set. This is possible because of convexity.
3. Compute the ellipsoid with the smallest volume containing this half ellipsoid.
4. Repeat 2. and 3. until some stopping criterion is fulfilled.

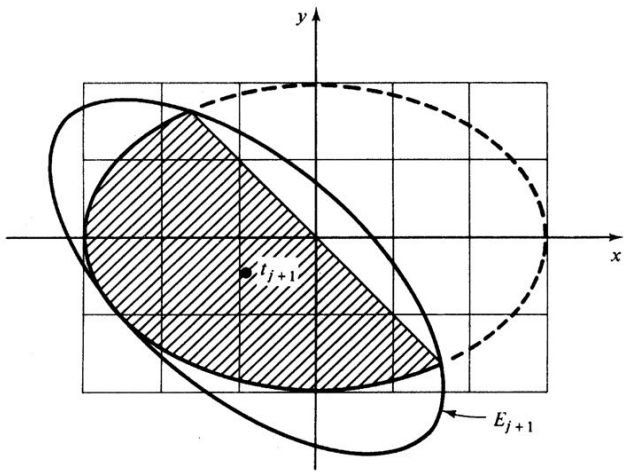

Figure 2: In the ellipsoid method, a sequence of ellipsoids is constructed.

Central path methods

We start by writing the LP problem in standard form and then adding slack variables:

$$\begin{aligned} &\text{maximize } \mathbf{c}^{\mathbf{T}}\mathbf{x} \\ &\text{subject to } \begin{cases} \mathbf{A}\mathbf{x} + \mathbf{x_s} = \mathbf{b} \\ \mathbf{x}, \mathbf{x_s} \geq 0. \end{cases} \end{aligned}$$

Central path methods

Algorithm idea:

1. **Log barrier trick:** The purpose of this step is to keep the variables away from the boundary. Instead of optimizing the original objective function, we optimize

$$\text{maximize } \mathbf{c}^{\mathbf{T}}\mathbf{x} + \mu \sum_{j=1}^{n} \log x_j + \mu \sum_{i=1}^{m} \log x_{s,i}.$$

   μ is a parameter that is fixed in each step of the algorithm. Initially it is large, and it will be reduced later. Intuitively, the added terms terms penalize points which are close to the boundary of the feasible set (since $\log x \to -\infty$ as $x \to 0$).

Central path methods

2. **Lagrange multipliers take care of the equality constraints:** Let

$$L(\mathbf{x}, \mathbf{x_s}, \mathbf{y}) = \mathbf{c}^\mathbf{T}\mathbf{x} + \mu \sum_{j=1}^{n} \log x_j + \mu \sum_{i=1}^{m} \log x_{s,i} + \mathbf{y}^\mathbf{T}(\mathbf{b} - \mathbf{A}\mathbf{x} - \mathbf{x_s}).$$

Here, $\mathbf{y}^\mathbf{T}$ is a row vector of Lagrange multipliers $y_1, \ldots, y_m$, one for each equality constraint. We have now got an unconstrained optimization problem to solve, which should be easier than what we had before.

Central path methods

3. When $L(\mathbf{x}, \mathbf{x_s}, \mathbf{y})$ is maximal, then the partial derivatives with respect to x_j, $x_{s,i}$ and y_i vanish ($j = 1, \ldots, n$, $i = 1, \ldots, m$). We compute all these derivatives. Taking the gradient with respect to the x_j variables, we obtain

$$\mathbf{c} + \mu \begin{bmatrix} 1/x_1 \\ \vdots \\ 1/x_n \end{bmatrix} - \mathbf{A^T y} = \mathbf{0}.$$

We let the vector above with entries $\mu \cdot 1/x_j$ be denoted by $\mathbf{y_s}$, i.e. $x_j y_{s,j} = \mu$ for $j = 1, \ldots, n$. The above equations can then be written in the simpler form

$$\mathbf{A^T y} - \mathbf{y_s} = \mathbf{c}.$$

3. (cont.) Taking the gradient with respect to the $\mathbf{x_s}$ variables, we obtain

$$\mu \begin{bmatrix} 1/x_{s,1} \\ \vdots \\ 1/x_{s,m} \end{bmatrix} - \mathbf{y} = \mathbf{0} \qquad \Longleftrightarrow \qquad \begin{cases} y_1 x_{s,1} = \mu, \\ \vdots \\ y_m x_{s,m} = \mu. \end{cases}$$

Finally, we take the gradient with respect to the $\mathbf{y}$ variables, and obtain the equations

$$\mathbf{b} - \mathbf{A}\mathbf{x} - \mathbf{x_s} = \mathbf{0}.$$

Central path methods

3. (cont.) To summarize, the equations that arise from putting all the partial derivatives to 0 are

$$Ax + x_s = \mathbf{b},$$
$$A^T y - y_s = \mathbf{c},$$
$$\begin{cases} y_1 x_{s,1} = \mu, \\ \quad \vdots \\ y_m x_{s,m} = \mu \end{cases} \quad \text{and} \quad \begin{cases} x_1 y_{s,1} = \mu, \\ \quad \vdots \\ x_n y_{s,n} = \mu. \end{cases}$$

The conditions involving μ are called μ-complementary slackness because of the resemblance of the CS conditions.

Central path methods

4. Solve the system of equations numerically, for example with Newton's method for a large μ. Then decrease μ and use the optimal solution of the previous step as a starting point for the next iteration.
   The optimal solution for each μ lies on a path which is called the central path. Decrease $\mu \searrow 0$ (without attaining 0). The sequence will converge to an optimal solution of the original problem. Note that the μ-complementary slackness conditions will look more and more like complementary slackness.

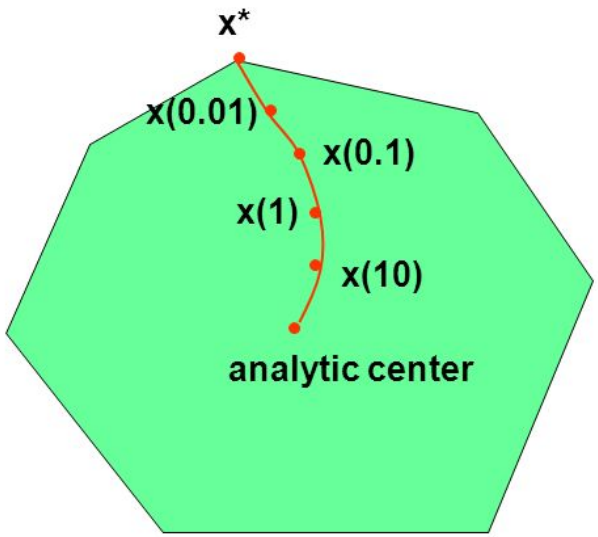

Figure 3: The optimal solutions lie on the central path for each μ. The analytic center corresponds to $\mu = \infty$.

1. Algorithm complexity

2. Problem classification

3. **Algorithm complexity for min cost flow problems and related problems**

Minimum cost flow problems

The minimum cost flow problem is the problem of shipping fixed amounts (b_i) in a network where each edge is assigned not only a capacity c_{ij} but also a (shipping) cost a_{ij}. For this problem, each node is either a source (i.e. no inflow), a sink (i.e. no outflow) or an intermediate node (i.e inflow = outflow), and there can be more than one source or sink. For each node, the source/sink strength is given as a number b_i with the property that $b_i > 0$ if node i is a sink, $b_i < 0$ if node i is a source, and $b_i = 0$ if node i is an intermediate node. A requirement for the existence of a feasible solution is that $\sum_{i=1}^{n} b_i = 0$. The minimization problem is

$$\text{minimize } \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_{ij}$$

$$\text{subject to } \begin{cases} \sum_{j=1}^{n} x_{ji} - \sum_{j=1}^{n} x_{ij} = b_i, & i = 1,\ldots,n, \\ 0 \le x_{ij} \le c_{ij}, & i,j = 1,\ldots,n. \end{cases}$$

Complexity for min cost flow problems

- Min cost flow problems can be solved with the simplex method, or
- a version of the simplex method which is adapted for networks. (See e.g. Holmgren's book, p. 313–317). The complexity of the best such algorithm seems to be $\mathcal{O}(VE \log V \log(VC))$, where V is the number of nodes (vertices), E is the number of edges and C is the maximum cost of any edge.
- By the Kruskaal–Hoffman theorem, the simplex method will find an integer solution if the indata has integer entries.
- Other problems that we have seen can be put in the framework of min cost flow problems, and so these problems belong to P too.

The transportation problem

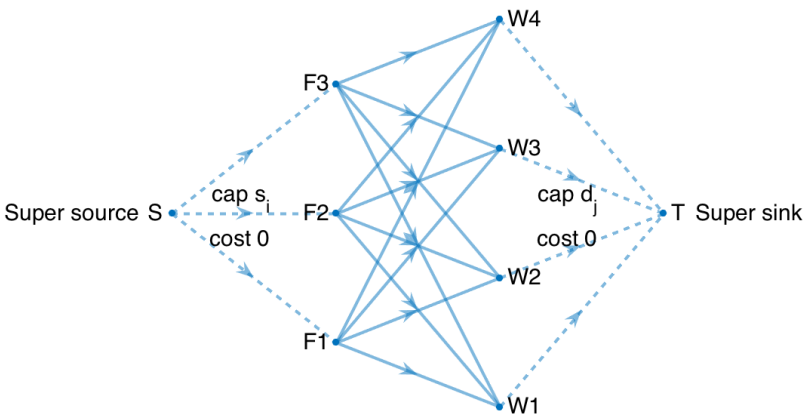

Figure 4: The transportation problem as a minimum cost flow problem

- The figure shows the resulting graph when the transportation problem is viewed as a minimal cost flow problem.
- The edges between node F_i and node F_j have capacities ∞ and costs c_{ij} (the transportation costs given in the transportation problem).
- Since the min cost flow problem belongs to P, the same is true for the transportation problem.

The assignment problem

- Since the assignment problem can be considered as a special case of a transportation problem, and the transportation problem belongs to P, the assignment problem belongs to P too.
- The Hungarian algorithm as it was formulated originally in 1957 (by Kuhn, based on the work by König and Egerváry) is $\mathcal{O}(n^4)$, but there are modifications of it (e.g. by Edmonds and Karp) that are $\mathcal{O}(n^3)$.
- In 2006 it was discovered that the assignment problem was solved already in the 19th century by Jacobi. See some more information here.

The shortest route problem

- The shortest route problem can be viewed as a minimum cost flow problem with all capacities being 0 and all $b_1 = -1$ (the starting node) and $b_n = 1$ (the end node).
- The cost on each edge is given by the distance between the respective nodes.
- Since the minimum cost flow problem belongs to P, so does the shortest route problem.
- The method that we discussed briefly in Lecture 9 is called Dijkstra's method, and this algorithm has complexity $\mathcal{O}(V^2)$, where V is the number of nodes in the network.

The maximal flow problem

- If we let all the costs a_{ij} of the minimum cost flow problem be 0 and also let $b_i = 0$ for $i = 2, \ldots, n$, then the problem looks very similar to the maximal flow problem.
- The difference is that the flow $f = \sum_{i=1}^{n} x_{1j}$ is specified in the minimum cost flow problem, but in the maximum flow problem, it is the objective function.
- If we add an edge from sink to source in the max flow problem with cost -1, and add the constraint that inflow $=$ outflow also for node 1 and n, then the objective function to be minimized in the minimum cost flow problem is $-x_{n1} = -\sum_{j=1}^{n} x_{1n}$. Moreover, the constraint sets of the two problems are the same, so after converting to a maximization problem, we see that the maximal flow problem can be reformulated as a minimum cost flow problem.
- Since the minimum cost flow problem belongs to P, so does the maximum flow problem.

The maximal flow problem

- Assuming that the capacities are all integers (or at least rational), the Ford–Fulkerson method for the maximum flow problem is $\mathcal{O}(Ef)$, where E is the number of edges and f is the value of the maximum flow.
- A specialization of Ford–Fulkerson known as the Edmonds–Karp algorithm is $\mathcal{O}(VE^2)$, where V is the number of vertices (nodes).
- The difference of the two versions of the methods is the order of how nodes are labelled (or rather: The Ford–Fulkerson doesn't have a specified order, whereas the Edmonds–Karp uses a breadth first search). For an example of the worst case scenario of the Ford–Fulkerson method, see the Wikipedia page (scroll down to the section "Integral example").
- If the capacities are not rational, then the Ford–Fulkerson algorithm may not terminate, and it may not even converge to a maximum flow. (See the same Wikipedia page as above).