Integer and Combinatorial Optimization: Algorithm Complexity

John E. Mitchell

Department of Mathematical Sciences
RPI, Troy, NY 12180 USA

January 2019
Outline

1. How run-time grows as problem size grows

2. Polynomial time algorithms and the class $\mathcal{P}$
   - Examples of problems in $\mathcal{P}$
Big-\(O\) notation

**Definition**

Let \(f\) and \(g\) be functions from the set of integers or the set of reals to the set of real numbers. The function \(f(k)\) is \(O(g(k))\) whenever there exist positive constants \(C\) and \(k'\) such that

\[
|f(k)| \leq C|g(k)| \quad \forall k \geq k'.
\]
Size of an instance

The size of an instance is the number of binary characters required to store the problem.
For example, given a positive integer $x$, we set $p = \lceil \log_2(x) \rceil$. Then

$$2^p \leq x < 2^{p+1} \quad \text{and} \quad x = \sum_{i=0}^{p} \delta_i 2^i \quad \text{for some binary } \delta_i.$$

For example, $42 = 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5$. So we need $O(\log(x))$ bits to store $x$.

We don’t need to specify the base for the logarithm here, because

if $x > 0$ and $y = \log_a(x)$, $z = \log_b(x)$ then $y = z \log_a(b)$. 
Data representation

The major classification of algorithm efficiency is insensitive to the choice of data representation, for the most part.

There are two restrictions:

1. The alphabet used to represent the data must contain at least two symbols. So the space to store a positive integer $x$ is $O(\log(x))$, instead of $x$ with a unary alphabet.

2. The “data” can’t contain information that requires a lot of calculation. For example, can’t store the length of all tours as part of the representation of a traveling salesman problem.
Data representation

The major classification of algorithm efficiency is insensitive to the choice of data representation, for the most part.

There are two restrictions:

1. The alphabet used to represent the data must contain at least two symbols. So the space to store a positive integer \( x \) is \( O(\log(x)) \), instead of \( x \) with a unary alphabet.

2. The “data” can’t contain information that requires a lot of calculation. For example, can’t store the length of all tours as part of the representation of a traveling salesman problem.
Computation time

The time required for an elementary operation (addition, subtraction, multiplication, division, comparison) is counted as *unit time*.

(Need to be a little careful if multiply very large numbers together.)
Instances

Perfect matching, linear programming, and node packing are all examples of optimization problems.

An optimization problem $X$ consists of an infinite number of instances $d_1, d_2, \ldots$ where the data for instance $d_i$ is given by a binary string of length $l(d_i)$.

For example, an instance of a perfect matching problem is given by lists of the vertices and edges of the graph $G = (V, E)$. 
Worst case running time

Let $A$ be an algorithm that solves every instance of $X$ in finite time.

Let $g_A(d_i)$ be the running time of $A$ on instance $d_i$.

Measure performance of $A$ on $X$ using the **worst-case** running time, for each problem size $k$:

$$f_A(k) := \max\{g_A(d_i) : l(d_i) = k\}.$$

Advantages of using worst-case performance measure:

- Gives a guarantee of performance of the algorithm
- Independent of the probability distribution of the instances.
- Appears to be the easiest to analyze.
Worst case running time

Let $A$ be an algorithm that solves every instance of $X$ in finite time.

Let $g_A(d_i)$ be the running time of $A$ on instance $d_i$.

Measure performance of $A$ on $X$ using the \textit{worst-case} running time, for each problem size $k$:

$$f_A(k) := \max\{g_A(d_i) : l(d_i) = k\}.$$  

Advantages of using worst-case performance measure:

- Gives a guarantee of performance of the algorithm
- Independent of the probability distribution of the instances.
- Appears to be the easiest to analyze.
Worst case running time

Let $A$ be an algorithm that solves every instance of $X$ in finite time.
Let $g_A(d_i)$ be the running time of $A$ on instance $d_i$.
Measure performance of $A$ on $X$ using the worst-case running time, for each problem size $k$:

$$f_A(k) := \max\{g_A(d_i) : l(d_i) = k\}.$$

Advantages of using worst-case performance measure:

- Gives a guarantee of performance of the algorithm
- Independent of the probability distribution of the instances.
- Appears to be the easiest to analyze.
Klee-Minty

For example, the simplex algorithm solves any linear optimization problem in finite time (provided we use an anti-cycling rule).

In the worst case, it may require $O(2^n)$ iterations to solve a problem in $n$ nonnegative variables with $n$ inequality constraints (namely, the Klee-Minty cube).
Outline

1. How run-time grows as problem size grows

2. Polynomial time algorithms and the class \( \mathcal{P} \)
   - Examples of problems in \( \mathcal{P} \)
Feasibility problems

For example, an instance of the *perfect matching feasibility problem* consists of a graph \( G = (V, E) \) and asks the question “Does there exist a perfect matching on this graph”? The answer is either *Yes* or *No*.

**Definition**

A *feasibility problem* \( X \) consists of a set of instances \( D \) and a set of feasible instances \( F \subseteq D \). Given an instance \( d \in D \), determine whether \( d \in F \). The answer is either *Yes* or *No*.
Binary search

Many instances of optimization problems can be solved using binary search on a sequence of instances of feasibility problems.

For example, can solve the 0–1 linear optimization problem

\[ \min_x \{ c^T x : Ax \geq b, \; x \in \mathcal{B}^n \} \]

by repeatedly solving the 0–1 linear feasibility problem

Does there exist \( x \in \mathcal{B}^n \) with \( Ax \geq b, \; c^T x \leq z \)?

At each iteration, we have upper and lower bounds on the optimal value \( z^* \). The initial bounds on \( z^* \) can be taken to be

\[ z^{UB} = \sum_{j: c_j > 0} c_j, \quad z^{LB} = \sum_{j: c_j < 0} c_j \]

since \( x \) is binary. We solve the feasibility problem with

\[ z = \frac{1}{2} (z^{LB} + z^{UB}) \]. We assume all the data is integral, so the optimal value is integral, so we can maintain integral values for \( z^{LB} \), \( z^{UB} \), and \( z \).
Binary search algorithm

1. Initialize $z^{UB}, z^{LB}$.
2. If $z^{LB} = z^{UB}$, **STOP** and return the optimal value $z^* = z^{LB} = z^{UB}$.
3. Let $z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor$.
4. Solve the $0-1$ linear feasibility problem
   
   \[
   \text{Does there exist } x \in \mathcal{B}^n \text{ with } Ax \geq b, c^T x \leq z? \]

5. If answer is **YES**: update $z^{UB} \leftarrow z$, return to Step 2.
6. If answer is **NO**: update $z^{LB} \leftarrow z + 1$, return to Step 2.

Let $z^0$ be the initial value of $z^{UB} - z^{LB}$. The number of iterations of the algorithm is $O(\log_2(z^0))$, since the difference between the bounds is halved at each iteration.
Binary search algorithm

1. Initialize $z^{UB}, z^{LB}$.
2. If $z^{LB} = z^{UB}$, STOP and return the optimal value $z^* = z^{LB} = z^{UB}$.
3. Let $z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor$.
4. Solve the 0–1 linear feasibility problem

   Does there exist $x \in \mathcal{B}^n$ with $Ax \geq b$, $c^T x \leq z$?

5. If answer is YES: update $z^{UB} \leftarrow z$, return to Step 2.
6. If answer is NO: update $z^{LB} \leftarrow z + 1$, return to Step 2.

Let $z^0$ be the initial value of $z^{UB} - z^{LB}$. The number of iterations of the algorithm is $O(\log_2(z^0))$, since the difference between the bounds is halved at each iteration.
Binary search algorithm

1. Initialize \( z^{UB}, z^{LB} \).
2. If \( z^{LB} = z^{UB} \), **STOP** and return the optimal value \( z^* = z^{LB} = z^{UB} \).
3. Let \( z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor \).
4. Solve the 0–1 linear feasibility problem

   Does there exist \( x \in B^n \) with \( Ax \geq b, c^T x \leq z \)?

5. If answer is **YES**: update \( z^{UB} \leftarrow z \), return to Step 2.
6. If answer is **NO**: update \( z^{LB} \leftarrow z + 1 \), return to Step 2.

Let \( z^0 \) be the initial value of \( z^{UB} - z^{LB} \). The number of iterations of the algorithm is \( O(\log_2(z^0)) \), since the difference between the bounds is halved at each iteration.
Binary search algorithm

1. Initialize $z^{UB}, z^{LB}$.
2. If $z^{LB} = z^{UB}$, STOP and return the optimal value $z^* = z^{LB} = z^{UB}$.
3. Let $z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor$.
4. Solve the 0–1 linear feasibility problem:
   
   Does there exist $x \in B^n$ with $Ax \geq b, c^T x \leq z$?

5. If answer is YES: update $z^{UB} \leftarrow z$, return to Step 2.
6. If answer is NO: update $z^{LB} \leftarrow z + 1$, return to Step 2.

Let $z^0$ be the initial value of $z^{UB} - z^{LB}$. The number of iterations of the algorithm is $O(\log_2(z^0))$, since the difference between the bounds is halved at each iteration.
Binary search algorithm

1. Initialize $z^{UB}, z^{LB}$.
2. If $z^{LB} = z^{UB}$, **STOP** and return the optimal value $z^* = z^{LB} = z^{UB}$.
3. Let $z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor$.
4. Solve the 0–1 linear feasibility problem

   Does there exist $x \in \mathbb{B}^n$ with $Ax \geq b$, $c^T x \leq z$?

5. If answer is **YES**: update $z^{UB} \leftarrow z$, return to Step 2.
6. If answer is **NO**: update $z^{LB} \leftarrow z + 1$, return to Step 2.

Let $z^0$ be the initial value of $z^{UB} - z^{LB}$. The number of iterations of the algorithm is $O(\log_2(z^0))$, since the difference between the bounds is halved at each iteration.
Binary search algorithm

1. Initialize \( z^{UB}, z^{LB} \).
2. If \( z^{LB} = z^{UB} \), **STOP** and return the optimal value \( z^* = z^{LB} = z^{UB} \).
3. Let \( z = \left\lfloor \frac{1}{2}(z^{LB} + z^{UB}) \right\rfloor \).
4. Solve the 0–1 linear feasibility problem

\[
\text{Does there exist } x \in B^n \text{ with } Ax \geq b, c^T x \leq z?
\]

5. If answer is **YES**: update \( z^{UB} \leftarrow z \), return to Step 2.
6. If answer is **NO**: update \( z^{LB} \leftarrow z + 1 \), return to Step 2.

Let \( z^0 \) be the initial value of \( z^{UB} - z^{LB} \). The number of iterations of the algorithm is \( O(\log_2(z^0)) \), since the difference between the bounds is halved at each iteration.
The Class $\mathcal{P}$

**Definition**

*Algorithm A is a polynomial time algorithm for the feasibility problem $X$ if $f_A(k)$ is $O(k^p)$ for some fixed $p$.  

$\mathcal{P}$ denotes the class of feasibility problems that can be solved in polynomial time.

*Problem $X \in \mathcal{P}$ if and only if there is a polynomial time algorithm for $X$.  


Outline

1. How run-time grows as problem size grows

2. Polynomial time algorithms and the class \( \mathcal{P} \)
   - Examples of problems in \( \mathcal{P} \)
Minimum spanning tree with upper bound

Given a graph \( G = (V, E) \), edge weights \( w_e \) for \( e \in E \), and an integer \( z \), does there exist a spanning tree with total weight no greater than \( z \)?

Use a greedy algorithm.

Weight \( \leq 25? \)

Solution:

YES (weight 23)
Minimum spanning tree with upper bound

Given a graph $G = (V, E)$, edge weights $w_e$ for $e \in E$, and an integer $z$, does there exist a spanning tree with total weight no greater than $z$?

Use a greedy algorithm.

Weight $\leq 25$?

Solution: YES (weight 23)
Shortest path with upper bound

Given a graph $G = (V, E)$, edge weights $w_e$ for $e \in E$, source node $s$ and sink node $t$, and an integer $z$, does there exist a path from $s$ to $t$ with total length no greater than $z$? Use Dijkstra’s algorithm if edge weights are nonnegative.

Path $\leq 6$? NO
Perfect matching feasibility problem

Given a graph $G = (V, E)$, does there exist a perfect matching?
Use Edmonds blossom (alternating path) algorithm.

No alternating path between unmatched vertices 2 and 6
Linear programming with bound

Given \( m \times n \) matrix \( A \), vectors \( b \in R^m \) and \( c \in R^n \), and a scalar \( z \), does there exist a nonnegative \( x \in R^n \) satisfying \( Ax = b \) and \( c^T x \leq z \)?

Use the ellipsoid algorithm or some interior point algorithms.
Exponential functions

**Definition**

The function $f(k)$ is exponential if there exist positive constants $c_1, c_2 > 0$, constants $d_1, d_2 > 1$, and a positive constant $k'$ such that

$$c_1 d_1^k \leq f(k) \leq c_2 d_2^k \quad \forall k \geq k'.$$

For example, let $b$ be the largest integer for an instance of an optimization problem $X$, so this requires storage $O(\log_2(b))$.

Assume the total storage is also $O(\log_2(b))$.

If algorithm $A$ runs in time $O(b)$ then it requires exponential time for this instance, since

$$b = 2^{\log_2(b)}.$$
The knapsack problem

The knapsack problem $\max\{c^T x : a^T x \leq b, x \text{ binary}\}$ can be solved in $O(b)$ iterations using dynamic programming.

However, the storage requirement is only $O(\log(b))$, assuming $b$ is the largest integer in the problem definition.

So the runtime is exponential in the storage requirement.
Max Flow

Consider a **max flow** problem on the following graph. We seek to maximize flow from node 1 to node 4. The arc capacities are indicated.
Solve with augmenting path method

\[
\begin{align*}
\text{take } t &= 1 \\
(u_e, x_e)
\end{align*}
\]

Next iteration:

\[
\begin{align*}
\text{take } t &= 1 \\
(u_e, x_e)
\end{align*}
\]

Need \(2M\) iterations to get to optimal solution, which is exponential in the storage requirement. To get polynomial algorithm, need to select augmenting path with fewest edges.
Solve with augmenting path method

\[ t = 1 \]
\[ (u_e, x_e) \]

Next iteration:

\[ t = 1 \]
\[ (u_e, x_e) \]

Need \( 2M \) iterations to get to optimal solution, which is exponential in the storage requirement. To get polynomial algorithm, need to select augmenting path with fewest edges.
Solve with augmenting path method

Next iteration:

Need $2M$ iterations to get to optimal solution, which is exponential in the storage requirement. To get polynomial algorithm, need to select augmenting path with fewest edges.
Polynomial time algorithms and the class \( \mathcal{P} \) Examples of problems in \( \mathcal{P} \)

**Solve with augmenting path method**

1. \( t \)
2. 2
3. 4
4. \( t \)

\[
\begin{align*}
1 & \quad t \\
2 & \quad t \\
3 & \quad t \\
4 & \quad t \\
\end{align*}
\]

**Next iteration:**

1. \( t \)
2. 2
3. \( t \)
4. \( t \)

\[
\begin{align*}
1 & \quad t \\
2 & \quad t \\
3 & \quad t \\
4 & \quad t \\
\end{align*}
\]

**Need 2M iterations to get to optimal solution, which is exponential in the storage requirement. To get polynomial algorithm, need to select augmenting path with fewest edges.**

\[
\begin{align*}
t & = 1 \\
( u_e, x_e )
\end{align*}
\]

\[
\begin{align*}
t & = 1 \\
( u_e, x_e )
\end{align*}
\]