Efficient Algorithms and Problem Complexity

– Dealing with NP-Completeness –

Frank Drewes
Department of Computing Science
Umeå University
Today’s Menu

1. Side Step: Constructing Witnesses if P=NP

2. Some Ways to Cope with NP-Completeness
Is the Restriction to Decision Problems Justified?

Almost all NP-complete problems are not really what we want to solve:

- The answer \( G \in \text{HAM} \) is insufficient. 
  We rather want to find the cycle.
- The answer \((G, k) \in \text{CLIQUE}\) is insufficient.
  We rather want to find the largest clique.
- The answer \(S \in \text{INTEGER PROGRAMMING}\) (the system \(S\) of linear inequalities has an integer solution) is insufficient. 
  We rather want to find the integer solution.
- The answer \(\varphi \in \text{SAT}\) is insufficient.  
  We rather want to find a satisfying assignment.
- ...

That is, we want to solve a function problem.
Is the Restriction to Decision Problems Justified?

What does $P = NP$ or $P \neq NP$ tell us about what really interests us?

A function problem is at least as hard as its decision problem

$\Rightarrow$ if $P \neq NP$ then the function problems are hard as well, no chance.

But what if $P = NP$? Can we then solve the corresponding function problems efficiently as well?
Example: SAT

Suppose \( P = NP \), and let \( M \) solve \( SAT \) in polynomial time.

How to find a satisfying assignment:

```plaintext
findAssignment(\( \varphi \)) where \( \varphi \in SAT \) with variables \( x_1, \ldots, x_n \)

for \( i = 1, \ldots, n \) do
  if \( M(\varphi\langle x_i \leftarrow true \rangle) \) then
    \( ASS[i] \leftarrow true \)
    \( \varphi \leftarrow \varphi\langle x_i \leftarrow true \rangle \) // replace \( x_i \) by \( true \) in \( \varphi \)
  else
    \( ASS[i] \leftarrow false \)
    \( \varphi \leftarrow \varphi\langle x_i \leftarrow false \rangle \) // replace \( x_i \) by \( false \) in \( \varphi \)

return \( ASS \)
```

Running time is polynomial (approx. \( n \) times the running time of \( M \)).
Finding Arbitrary Certificates if $P = NP$

Let $R \in P$ be a polynomially bounded relation such that

$$A = \{x \mid \exists y \in \{0, 1\}^* \text{ such that } (x, y) \in R\}.$$  

Notice: $A' = \{(x, y_0) \mid \exists y_1 \in \{0, 1\}^* \text{ such that } (x, y_0y_1) \in R\}$ is in NP.

```
findCertificate(x) where x \in A
    y_0 \leftarrow \epsilon
    \textbf{while} (x, y_0) \notin R \textbf{ do} \quad \text{ // polynomial since } R \in P
    \quad \textbf{if} (x, y_00) \in A' \textbf{ then} \quad \text{ // polynomial since } A' \in NP = P
    \quad \quad y_0 \leftarrow y_00
    \textbf{else}
    \quad \quad y_0 \leftarrow y_01
    \textbf{return} \ y_0
```

Running time is again polynomial (if $P = NP$). [WHY?]
Side Step: Constructing Witnesses if P=NP

Solving the Optimization Version of TSP if P = NP

Let $M$ be an $n \times n$-matrix of distances $\leq d_{\text{max}}$. Assume that the decision version $\text{TSP}_D$ of the Travelling Salesman Problem is in $P$.

**Step 1:** Find the length $k$ of the shortest tour by binary search, similar to finding a certificate. Takes $O(\log(d_{\text{max}}n))$ iterations (polynomial!).

**Step 2:**

\[
\text{findTour}(x)
\]

\[
\text{for } (i,j) \in \{1, \ldots, n\}^2 \text{ do}
\]

\[
M_{i,j} \leftarrow k + 1
\]

\[
\text{if } (M, k) \notin \text{TSP}_D \text{ then restore } M_{i,j}
\]

\[
\text{return } \text{ tour given by distances } \leq k \text{ in } M
\]

Only $n^2$ iterations $\Rightarrow$ running time is polynomial if $\text{TSP}_D \in P$. 
Some Ways to Cope with NP-Completeness

- **Brute Force**
  Try to find algorithms running in time $O(c^n)$ for a small $c$.

- **Randomization**
  Use algorithms that find the correct solution with high probability.

- **Approximation**
  Find a solution to an optimization problem not too far away from the optimum.

- **Fixed-parameter algorithms**
  Try to find algorithms that are efficient if certain parameters are small (next lecture).
Ex.: Brute Force Algorithm for INDEPENDENT SET$_F$

Recall: An independent set in a graph $G$ is a set $V'$ of vertices such that none of them are connected by an edge.

We consider the function problem INDEPENDENT SET$_F$.

<table>
<thead>
<tr>
<th>INDEPENDENT SET$_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
</tbody>
</table>

The straightforward solution tries all subsets of the set of vertices. $\Rightarrow$ time $O(2^n)$. 
Ex.: Brute Force Algorithm for INDEPENDENT SET

Notation: \( N(v) = \) set of neighbors of \( v \) in a graph

\[
\text{largestIndependentSet}(G) \quad \text{where} \quad G = (V, E)
\]

if \( E = \emptyset \) then
    return \( |V| \)
else
    Let \( v \in V \) with \( N(v) \neq \emptyset \)
    \[ k \leftarrow \text{largestIndependentSet}(G - \{v\}) \quad \text{// deselect } v \]
    \[ l \leftarrow \text{largestIndependentSet}(G - (\{v\} \cup N(v))) \quad \text{// select } v \]
    return \( \max(k, l + 1) \)

Recurrence: \( T(0) = T(1) = 1 \) and \( T(n + 2) = T(n + 1) + T(n) + O(n^2) \)
\( \Rightarrow \) analysis of the recurrence yields \( T(n) \in O(1.62^n) \).

Best known: \( O(1.211^n) \) (Robson’s algorithm).
Randomized Algorithms - the Class RP

Consider a polynomial-time RAM $M$ that has an additional coin flipping operation (fair and independent coin tosses).

⇒ for an input $x$ there is a certain probability $\Pr[M(x) = 1]$ that $M$ will accept $x$.

$M$ is a Monte Carlo algorithm for a decision problem $A$ if, for all $x$,

- if $x \in A$ then $\Pr[M(x) = 1] \geq 1/2$ and
- if $x \notin A$ then $\Pr[M(x) = 1] = 0$.

Yes is always correct, but no may be wrong with probability $1/2$.

Observation: The probability of wrong answers can be reduced to $2^{-k}$ by $k$ repetitions. E.g., $k = 100$ yields a failure probability smaller than $10^{-30}$.

Note that $\text{RP} \subseteq \text{NP}$ (just replace coin tosses by nondeterminism).
Other randomized complexity classes

- **ZPP = RP ∩ coRP** – *yes/no* are always correct, but with probability 1/2 the answer may be “Don’t know”. This yields zero error probability and polynomial expected running time.

- **BPP** – both *yes* and *no* correct with probability $\geq \frac{3}{4}$. Largest class of practical interest; could in principle be implemented.

- **PP** – both *yes* and *no* correct with probability $> \frac{1}{2}$. Too large for practical use; contains both NP and coNP.
Approximation Algorithms

We may want to approximate an optimization problem (here we consider the minimization case).

<table>
<thead>
<tr>
<th>Minimization problems formalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>A minimization problem consists of a set $I$ of instances, a set $C(x)$ of solution candidates for every $x \in I$, and a measure $m: C \to \mathbb{R}<em>+$ (where $C = \bigcup</em>{x \in I} C(x)$).</td>
</tr>
<tr>
<td>For $x \in I$, the goal is to compute $y \in C(x)$ such that $m(y) = opt(x) = \min{m(y') \mid y' \in C(x)}$.</td>
</tr>
</tbody>
</table>

Suppose $M$ computes some $y \in C(x)$ for every $x \in I$, but not necessarily one with $m(y) = opt(x)$. The least upper bound of the quotients

$$m(M(x))/opt(x)$$

is the performance ratio of $M$. 
Example: BIN PACKING

We want to store \( n \) items of size \( s_1, \ldots, s_n \leq 1 \) in as few as possible “bins” of size 1.

**BIN PACKING**

- **Instance:** A sequence of positive item sizes \( s_1, \ldots, s_n \leq 1 \).
- **Candidate:** Placement of \( s_1, \ldots, s_n \) in \( m \) “bins” of size 1 each.
- **Measure:** The number \( m \) of bins used.

The decision version **BIN PACKING** (where \( m \) is given) is NP-complete.
An Approximation Algorithm for BIN PACKING

Johnsson’s algorithm: Fill the bins one by one with \( s_1, \ldots, s_n \).

\[
\text{nextFit}(s_1, \ldots, s_n)
\]

allocate array \( store[1, \ldots, n] \)  // item \( i \) is stored in bin \( store[i] \)
bin \( \leftarrow 1; \) fill \( \leftarrow 0 \)
for \( i = 1, \ldots, n \) do
  if \( \text{fill} + s_i > 1 \) then
    bin \( \leftarrow \text{bin} + 1; \) fill \( \leftarrow 0 \)
    store[\( i \)] \( \leftarrow \text{bin} \)
    fill \( \leftarrow \text{fill} + s_i \)
return \( store \)

- Running time is obviously linear.
- Performance ratio is 2: at most twice as many bins as necessary are used (next slide).
Performance Ratio of \textit{nextFit}

\begin{verbatim}
nextFit(s_1, \ldots, s_n)

allocate array store[1, \ldots, n]       // item i is stored in bin store[i]
bin \leftarrow 1; fill \leftarrow 0
for i = 1, \ldots, n do
    if fill + s_i > 1 then
        bin \leftarrow bin + 1; fill \leftarrow 0
        store[i] \leftarrow bin
        fill \leftarrow fill + s_i

return store
\end{verbatim}

Performance ratio: Suppose \textit{nextFit} uses \(b\) bins.

The sum of the sizes of items in bins \(i\) and \(i + 1\) is greater than 1
\(\Rightarrow\) the total contents of the \(b\) bins is at least \(b/2\)
\(\Rightarrow\) at least \(\lceil b/2 \rceil\) bins are needed in every solution
\(\Rightarrow\) in particular, the optimal solution uses at least \(\lceil b/2 \rceil\) bins.
Remarks Regarding \textit{nextFit}

- The performance ratio is indeed \(2\), i.e., for some instances, \textit{nextFit} uses (almost) twice as many bins as is optimal. [Can you find one?]
- It is an \textbf{online algorithm}: items are processed as they arrive.
- It is a \textbf{1-bounded-space algorithm}: at most one bin is open at a time.

These are \textbf{very useful properties}. Think, e.g., of

- containers to be filled with delivered items, or
- data items to be combined into fixed-size packets and sent over communication channels.

If we precede \textit{nextFit} by sorting \(s_1, \ldots, s_n\) decreasingly (offline!) the asymptotic performance ratio for large \(n\) becomes \(11/9\).
More on approximation and randomization can be found in the textbooks and in any reasonable book on computational complexity.