10 Advanced Topics in Complexity Theory

- What to do with a problem that is *intractable* and does not accept a deterministic exact solution in polynomial time
- Relax the problem:
  1. Instead of solving it exactly, *approximate* the solution
  2. Instead of using a deterministic algorithm, use a *probabilistic* (a.k.a. randomized) algorithm
- Approximation algorithm finds a solution that is *guaranteed* to be close to the optimal exact solution
- Probabilistic algorithm comes up with the exact solution with a *high probability*
- Sometimes it may fail to give the correct answer (Monte Carlo) or may have a high time requirement (Las Vegas)

10.1 Approximation Algorithms

- Let us examine a problem, where we are given
  - A ground set $U$ with $m$ elements
  - A collection of subsets of the ground set $S = \{ S_1, \ldots, S_n \}$ s.t. it is a *cover* of $U$: $US = U$

- The aim is to find a *subcover* $S' \subseteq S$,
  $US' = U$,

  containing as few subsets as possible

- This problem is known as the *Minimum Set Cover* (minSC)

- One of the oldest and most studied combinatorial optimization problems
The corresponding decision problem
- Given: a ground set \( U \), cover \( S \) and a natural number \( k \)
- Question: Does \( U \) have a subcover \( S' \subseteq S \) s.t. \( |S'| \leq k \)?

**Theorem** The decision version of minimum set cover problem is \( \text{NP-complete} \).

**Proof.** Obviously \( \text{minSC} \in \text{NP} \): Let us guess from the given cover \( S \) a subcover \( S' \) containing \( k \) subsets and verify deterministically in polynomial time that we really have a subcover.

Polynomial time reduction \( \text{VC} \leq_{\text{p}} \text{minSC} \) is easy to give. Let \( (G, k) \)
be an instance of the vertex cover in which \( G = (V, E) \). We choose the mapping \( f \):

\[
f((V, E), k) = (E, V_E, k),
\]

where \( V_E \) is the collection of edges connected to the nodes of \( G \).
In other words, for each \( v \in V \) has a corresponding set
\[
\{ e \in E \mid e = (v, w) \}.
\]

Clearly \( f \) is computable in polynomial time and is a reduction. \( \square \)
Hence, minSC is an intractable problem – we do not know of a polynomial time algorithm for solving it. Therefore, we attempt to find a polynomial time algorithm that does not necessarily give the best possible (optimal) solution, but can be shown always to be at most a function of the input length worse than the optimal solution. Such an algorithm is called an approximation algorithm.

Let us denote by
- \( \text{Opt} \), the cost of the solution given by an optimal algorithm and
- \( \text{App} \), that of the solution given by an approximation algorithm.
Since minSC is a minimization problem, $\frac{\text{App}}{\text{Opt}} \geq 1$

- The closer to 1 this ratio is, the better the solution produced approximates the optimal solution.

- From an approximation algorithm one requires that the fraction is bounded by a function of the length $n$ of the input:

$$\frac{\text{App}}{\text{Opt}} \leq \rho(n)$$

- $\rho(n)$ is the approximation ratio of the algorithm.
  - The algorithm is called an $\rho(n)$-approximation algorithm.
  - At the best the approximation ratio does not depend at all on the length $n$ of the input, but is constant.

Let us examine the following algorithm for vertex cover:

- We will show that it is an 2-approximation algorithm for the problem.

**Input:** An undirected graph $G = (V, E)$

**Output:** Vertex cover $C$

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E$
3. while $E' \neq \emptyset$ do
   a. Let $(u, v)$ be any edge of the set $E'$;
   b. $C \leftarrow C \cup \{u, v\}$;
   c. Remove from $E'$ all edges connected to nodes $u$ and $v$;
4. od
5. return $C$;
Selection of the first random edge: \((b, c)\)

We remove other edges connected with nodes \(b\) and \(c\)
The next random choice: \((e, f)\) and
Removal of other edges connected with its nodes

The only remaining choice \((d, g)\)

We end up with a cover of 6 nodes, while the optimal one has 3 nodes (e.g., \(b, d, e\))
**Theorem 10.1** The above given algorithm is polynomial time 2-approximation algorithm for vertex cover.

**Proof.** The time complexity of the algorithm, using adjacency list representation for the graph, is \( O(V + E) \), and thus uses a polynomial time.

The set of nodes \( C \) returned by the algorithm obviously is a vertex cover for the edges of \( G \), because nodes are inserted into \( C \) in the loop of row 3 until all edges have been covered.

Let \( A \) be the set of edges chosen by algorithm in row 3a. In order to cover the edges of \( A \) any vertex cover — in particular also the optimal vertex cover — has to contain at least one of the ends of each edge in \( A \).

Because the end points of the edges in \( A \) are distinct by the design of the algorithm, \( |A| \) is a lower bound for the size of any vertex cover.

In particular,

\[ \text{Opt} \geq |A|. \]

The above algorithm always selects in row 3a an edge whose neither end point is yet in the set \( C \). Hence,

\[ \text{App} = |C| = 2|A|. \]

Combining the above equations yields

\[ \text{App} = 2|A| \leq 2\text{Opt}, \]

and therefore

\[ \text{App}/\text{Opt} \leq 2. \]

\( \square \)
• Also set cover has a simple greedy approximation algorithm
• Neither this nor any other polynomial time deterministic algorithm can attain a constant approximation ratio

**Input:** Ground set $U$ and its cover $S$  
**Output:** Set cover $C$

1. $X \leftarrow U$;  
2. while $X \neq \emptyset$ do
   a. select $S' \in S$ s.t. $|S' \cap X|$ is maximized;
   b. $X \leftarrow X \setminus S'$;
   c. $C \leftarrow C \cup \{ S' \}$;
3. od;
4. return $C$;
Greedy: 4 subsets

Optimal: 3 subsets
The greedy algorithm can quite easy to implement to run in polynomial time in the length of the input $|U|$ and $|S|$

- The loop in row 2 is executed at most $\min(|U|, |S|)$ times and the body of the loop itself can be implemented to require time $O(|U| \cdot |S|)$
- Altogether the time requirement thus is $O(|U| \cdot |S| \min(|U|, |S|))$
- It is also possible to give a linear time implementation for the greedy approximation algorithm for set cover

The collection $C$ returned by the algorithm is obviously a set cover, because the loop of row 2 is executed until there are no more elements to cover.

In order to relate the cost of the set cover returned by the greedy algorithm, we set cost 1 to each of the chosen sets

- Let $S_i$ be the set selected by the greedy algorithm at round $i$
- We distribute the cost of $S_i$ evenly among all those elements in it that now become covered for the first time
- Let $c_u$ denote the cost assigned on element $u \in U$
- Each element gets assigned a cost only once, the first time it is covered by some set
- If $u$ is first covered by the set $S_i$, the cost assigned to it is:

$$c_u = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}$$
• Each set selected by the greedy algorithm is assigned cost 1 so that
\[ \text{App} = |C| = \sum_{u \in U} c_u \]
• On the other hand, the cost of the optimal cover \( C^* \) is
\[ \sum_{S' \in C^*} \sum_{u \in S'} c_u \]
• Because each \( u \in U \) belongs to at least one \( S' \in C^* \), we have
\[ \sum_{S' \in C^* \cup S} c_u \geq \sum_{u \in U} c_u \]
• Combining the above given yields
\[ \text{App} \leq \sum_{S' \in C^* \cup S} c_u \]

• Let \( H(k) \) denote the \( k \)-th harmonic number
\[ H(k) = \sum_{j=1}^{k} \frac{1}{j} = 1 + \frac{1}{2} + \cdots + \frac{1}{k} \]
• We define \( H(0) = 0 \)
• Next we show that for any \( S' \in S \) it holds
\[ \sum_{S' \in S} c_s \leq H(|S'|) \]
• Then, by the previous inequality,
\[ \text{App} \leq \sum_{S' \in C^*} H(|S'|) \leq |C^*| \cdot H(\max\{|S'|: S' \in S\}) = \text{Opt} \cdot H(\max\{|S'|: S' \in S\}) \]
Lemma For each $S' \in \mathcal{S}$ it holds

$$\sum_{u \in S'} c_u \leq H(|S'|)$$

Proof. Let $S' \in \mathcal{S}$ be arbitrary and $i = 1, 2, \ldots, |\mathcal{C}|$. Furthermore, let

$$n_i = |S' \setminus (S_1 \cup S_2 \cup \ldots \cup S_i)|$$

be the number of those elements of $S'$ that have not yet been covered when the greedy algorithm has chosen sets $S_1, S_2, \ldots, S_i$ to the set cover.

Let $n_0 = |S'|$.

Let $k$ be the smallest index s.t. $n_k = 0$; i.e., every element of $S'$ belongs to at least one of the sets $S_1, S_2, \ldots, S_k$.

Then $n_{i-1} \geq n_i$ and $S_i, i = 1, 2, \ldots, k$, covers $n_{i-1} - n_i$ elements for the first time.

Now

$$\sum_{u \in S'} c_u = \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{|S_i \setminus (S_1 \cup \ldots \cup S_{i-1})|}.$$ 

Since $S_i$ is chosen greedily, it must cover at least as many elements as the set $S'$ (or otherwise $S'$ should have been selected). Hence,

$$|S_i \setminus (S_1 \cup \ldots \cup S_{i-1})| \geq |S' \setminus (S_1 \cup \ldots \cup S_{i-1})| = n_{i-1}$$

Which further yields

$$\sum_{u \in S'} c_u \leq \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{n_{i-1}}.$$
$$\sum_{n \in S} c_n \leq \sum_{i=1}^{k} (n_{i-1} - n_i) \frac{1}{n_{i-1}}$$

$$= \sum_{i=1}^{k} \sum_{j=n_{i-1}}^{n_i} \frac{1}{j}$$

$$\leq \sum_{i=1}^{k} \sum_{j=n_{i-1}+1}^{n_i} \frac{1}{j}$$

because \( j \leq n_{i-1} \). Moreover,

$$= \sum_{i=1}^{k} \left( \sum_{j=1}^{n_{i-1}} \frac{1}{j} - \sum_{j=1}^{n_i} \frac{1}{j} \right)$$

$$= \sum_{i=1}^{k} (H(n_{i-1}) - H(n_i))$$

$$= H(n_0) - H(n_k),$$

since the other terms in the sum cancel each other out.

We have chosen \( n_k = 0 \) and defined \( H(0) = 0 \). Therefore, further

$$= H(n_0) - H(0)$$

$$= H(n_0)$$

$$= H(|S'|)$$

and we have proved the lemma.

- For the harmonic number \( H(k) \) it holds \( \ln k < H(k) \leq \ln (k + 1) \)
- From the above results it follows:

**Theorem** For the greedy algorithm of the set cover problem it holds that

$$\frac{\text{App}}{\text{Opt}} \leq H(\max \{|S'|: S' \in S\}) \leq \ln |U| + 1$$
• In some applications $\max \{ |S'| : S' \subseteq S \}$ is a small constant
• Then the solution returned by the greedy algorithm is only a small constant away from the optimal one
• In particular, if subsets $S'$ have an upper bound $d$ for their size, $App/Opt \leq H(d)$

E.g., when the nodes of the graph of vertex cover have maximum degree 3, then
• the solution returned by the greedy set cover algorithm is at most $H(3) = 11/6 < 2$ times as large as the optimal cover

Feige, 1996: no polynomial-time algorithm can approximate $\text{minSC}$ within $(1-\epsilon) \ln m$, for any $\epsilon > 0$, unless $\text{NP} \subseteq \text{DTIME}(n^{\log \log n})$
• Hence, it is not possible to find an approximation algorithm for $\text{minSC}$ that would be significantly better than the greedy one

Slavík, 1996: A more exact upper bound for the approximation ratio of the greedy algorithm is
$$\ln m - \ln \ln m + \Theta(1)$$
• In fact this is also a lower bound for the approximation ratio of the greedy algorithm
• $\ln m - \ln \ln m + \Theta(1)$ is thus the asymptotically exact approximation ratio of the greedy algorithm
10.2 Probabilistic Algorithms

- A.k.a. randomized algorithms
- Another way of dealing with too time consuming computation
- Certain types of problems seem to be more easily solvable by “flipping a coin” than by deterministic algorithms
- Calculating the best choice may require excessive time
- Estimating it may introduce a bias that invalidates the result
- For example, statisticians use *random sampling*
  - Instead of querying all the individuals for their political preferences might take too long
  - Randomly selected subset of voters gives reliable results at a small cost

The Class BPP

- A *probabilistic* Turing machine $N$ is a nondeterministic TM in which each nondeterministic step is called a *coin-flip step*
- Such a step has two legal next moves
- We assign a probability to each branch $b$ of $N$’s computation on input $w$ as follows:
  \[
  \Pr[b] = 2^{-k},
  \]
  where $k$ is the number of coin-flip steps that occur on branch $b$
- The probability that $N$ accepts $w$ is
  \[
  \Pr[N \text{ accepts } w] = \sum_{b \in A} \Pr[b]
  \]
  where $A$ is the set of accepting branches of computation
\[
\Pr[N \text{ rejects } w] = 1 - \Pr[N \text{ accepts } w]
\]

As usual, when a probabilistic TM recognizes a language, it must accept all strings in the language and reject all those out of the language.

Except that now we allow the machine a small probability of error.

For all \(0 \leq \varepsilon < \frac{1}{2}\) we say that \(N\) recognizes language \(A\) with probability of error \(\varepsilon\) if

- \(w \in A \Rightarrow \Pr[N \text{ accepts } w] \geq 1 - \varepsilon\)
- \(w \notin A \Rightarrow \Pr[N \text{ rejects } w] \geq 1 - \varepsilon\)

I.e., the probability of obtaining the wrong answer by simulating \(N\) is at most \(\varepsilon\).

Sometimes error probability bounds depend on the input length \(n\); e.g., exponentially small: \(\varepsilon = 2^{-n}\)

**Definition 10.4** \(BPP\) is the class of languages that are recognized by probabilistic polynomial time Turing machines with an error probability \(\frac{1}{2}\).

- Any constant error instead of \(\frac{1}{2}\) would yield an equivalent definition as long as it is in the interval \([0, \frac{1}{2}]\).
- By the virtue of the following amplification lemma we can always make the error probability exponentially small.
- A probabilistic algorithm with an error probability of \(2^{-100}\) is far more likely to give an erroneous result because of a hardware failure than because of an unlucky toss of its coin.
Amplification Lemma

**Lemma 10.5** Let $\epsilon$ be a fixed constant strictly in between 0 and $\frac{1}{2}$. Then for any polynomial $p(n)$ a probabilistic polynomial time TM $N$ that operates with error probability $\epsilon$ has an equivalent probabilistic polynomial time TM $N_2$ that operates with an error probability of $2^{-p(n)}$.

**Proof (Idea)** $N_2$ simulates $N$ by running it a polynomial number of times and taking the majority vote of the outcomes. The probability of error decreases exponentially with the number of runs of $N$ made.

---

**Primality**

- Let $\mathbb{Z}_p = \{ 0, ..., p - 1 \}$
- Every integer is equivalent modulo $p$ to some member of the set $\mathbb{Z}_p$

**Theorem 10.6 (Fermat’s little theorem)**

If $p$ is prime and $a \in \mathbb{Z}_p^*$, then $a^{p-1} \equiv 1 \pmod{p}$.

- For example, $2^{7-1} = 2^6 = 64$ and $64 \mod 7 = 1$
  while $2^{6-1} = 2^5 = 32$ and $32 \mod 6 = 2$
  hence 6 is not prime

- We show that 6 is a composite number without factoring it!
• Fermat’s little theorem, thus, (almost) gives a test for primality
• We say that $p$ passes the Fermat test at $a$, if
  \[ a^{p-1} \equiv 1 \pmod{p} \]

• Call a number $p$ pseudoprime if it passes Fermat tests for all smaller $a$ relatively prime to it
• Only infrequent Carmichael numbers are pseudoprime without being prime
• If a number is not pseudoprime, it fails at least half of all Fermat tests
• Hence, we easily get a pseudoprimality algorithm with an exponentially small error probability

Pseudoprime($p$)
1. Select random $a_1, \ldots, a_k \in \mathbb{Z}_p^*$
2. Compute $a_i^{p-1} \pmod{p}$ for each $i$
3. If all computed values are 1 accept, otherwise reject

• If $p$ isn’t pseudoprime, it passes each randomly selected test with probability at most $\frac{1}{2}$
• The probability that it passes all $k$ tests is thus at most $2^{-k}$
• The algorithm operates in polynomial time

• To convert this algorithm to a primality algorithm, we should still avoid the problem with the Carmichael numbers
The number 1 has exactly two square roots, 1 and -1, modulo any prime \( p \).

For many composite numbers, including all the Carmichael numbers, 1 has four or more square roots.

For example, ± 1 and ± 8 are the four square roots of 1 modulo 21.

We can obtain square roots of 1 if \( p \) passes the Fermat test at \( a \) because:

- \( a^{p-1} \mod p \equiv 1 \) and so
- \( a^{(p-1)/2} \mod p \) is a square root of 1.

We may repeatedly divide the exponent by two, so long as the resulting exponent remains an integer.

---

Prime\( (p) \)

\% accept = input \( p \) is prime

1. If \( p \) is even, accept if \( p = 2 \), otherwise reject.
2. Select random \( a_1, \ldots, a_k \in \mathbb{Z}_p^* \).
3. For each \( i \in \{ 1, \ldots, k \} \):
   a) Compute \( a_i^{p-1} \mod p \) and reject if different from 1.
   b) Let \( p - 1 = st \) where \( s \) is odd and \( t = 2^h \) is a power of 2.
   c) Compute the sequence \( a_i^{s-2}, a_i^{s-2^2}, \ldots, a_i^{s-2^h} \) modulo \( p \).
   d) If some element of this sequence is not 1, find the last element that is not 1 and reject if that element is not \(-1\).
4. All test have been passed, so accept.
Lemma 10.7 If p is an odd prime,
\[ \Pr[\text{Prime accepts } p] = 1. \]

Proof If p is prime, no branch of the algorithm rejects. Rejection in step 3a means that \((a^p \equiv 1 \mod p) \neq 1\) and Fermat’s little theorem implies that p is composite.

If rejection happens in step 3d, there exists some \(b \in \mathbb{Z}_p^*\) s.t. \(b \not\equiv \pm 1 \mod p\) and \(b^2 \equiv 1 \mod p\). Therefore \(b^2 - 1 \equiv 0 \mod p\).

Factoring yields \((b - 1)(b + 1) \equiv 0 \mod p\), which implies that \((b - 1)(b + 1) = cp\) for some positive integer c.

Because \(b \not\equiv \pm 1 \mod p\), both \(b - 1\) and \(b + 1\) are in the interval \([0, p]\). Therefore p is composite because a multiple of a prime number cannot be expressed as a product of numbers that are smaller than it is. \(\Box\)

• The next lemma shows that the algorithm identifies composite numbers with high probability

• An important elementary tool from number theory, Chinese remainder theorem, says that a one-to-one correspondence exists between \(\mathbb{Z}_{pq}\) and \((\mathbb{Z}_p \times \mathbb{Z}_q)\) if p and q are relatively prime:
  • Each number \(r \in \mathbb{Z}_{pq}\) corresponds to a pair \((a, b)\), where \(a \in \mathbb{Z}_p\) and \(b \in \mathbb{Z}_q\) s.t.:
    • \(r \equiv a \mod p\) and
    • \(r \equiv b \mod q\)
Lemma 10.8 If p is an odd composite number, 
\[ \Pr[\text{Prime accepts } p] \leq 2^{-k}. \]

Proof Omitted, takes advantage of the Chinese remainder thm. □

- Let \( \text{PRIMES} = \{ n \mid n \text{ is a prime number in binary } \} \)
- The preceding algorithm and its analysis establishes:

Theorem 10.9 \( \text{PRIMES} \in \text{BPP} \)

Note that the probabilistic primality algorithm has one-sided error. When it rejects, we know that the input must be composite. An error may only occur in accepting the input.

Thus an incorrect answer can only occur when the input is a composite number. For all primes we get the correct answer.

- The one-sided error feature is common to many probabilistic algorithms, so the special complexity class \( \text{RP} \) is designated for it:

Definition 10.10 \( \text{RP} \) is the class of languages that are recognized by probabilistic polynomial time Turing machines where inputs in the language are accepted with a probability of at least \( \frac{1}{2} \) and inputs not in the language are rejected with a probability of 1.

- Our earlier algorithm shows that \( \text{COMPOSITES} \in \text{RP} \)
PRIMES $\in P$

- A generalization of Fermat's little theorem:

**Theorem A.** Let $a$ and $p$ be relatively prime and $p > 1$. $p$ is a prime number if and only if $(X - a)^p \equiv X^p - a \pmod{p}$

- $X$ is not important here, only the coefficients of the polynomial $(X - a)^p - (X^p + a)$ are significant.
- For $0 < i < p$, the coefficient of $X^i$ is $(\binom{p}{i})a^{p-i}$. Supposing that $p$ is prime, $(\binom{p}{i}) = 0 \pmod{p}$ and hence all the coefficients are zero.
- Therefore, we are left with the first term $X^p$ and the last one $-a^p$, which is $-a \pmod{p}$.
- Unfortunately, deciding the primality of $p$ based on this requires an exponential time.

Agrawal (1999): it suffices to examine the polynomial $(X - a)^p \pmod{X^{r} - 1}$

- If $r$ is large enough, the only composite numbers that pass the test are powers of odd primes.
- On the other hand, $r$ should be quite small so that the complexity of the approach does not grow too much.

Kayal & Saxena (2000): Based on an unproven conjecture, $r$ doesn’t have to be larger than $4(\log^2 p)$, in which case the complexity of the test procedure is only of the order $O(\log^4 n)$; that is, belongs to $P$.

- The only difficulty is that the result is based on an unproven claim.
A pair of odd numbers is called *Sophie Germain primes* if both $q$ and $2q + 1$ are primes (related to Fermat’s last theorem).

Agrawal, Kayal & Saxena (2002): If one can find a pair of SG primes $q$ and $2q + 1$ s.t.

$$q > 4\sqrt{2q + 1} \cdot \log p$$

then $r$ does not need to be larger than

$$2\left(\sqrt{2q + 1}\right) \cdot \log p$$

Unfortunately this test is recursive and has time requirement of $O(\log^{12} n)$ instead of the $O(\log^3 n)$ mentioned above.

---

**Deterministic-Prime($p$)**

1. if $p = a^b$ for some $b > 1$ then reject;
2. $r \leftarrow 2$;
3. while $r < p$ do
   a) if $\gcd(p, r) \neq 1$ then reject;
   b) if Deterministic-Prime($r$) then $\% \ r > 2$
      i. Let $q$ be the largest factor of $r-1$;
      ii. if $q > 4\sqrt{r} \cdot \log p$ and $p^{(r−2)/q} \neq 1 \pmod{r}$ then break;
   c) $r \leftarrow r + 1$;
4. for $a \leftarrow 1$ to $2\sqrt{r} \cdot \log p$ do
   if $(x-a)^r \neq x^r - a \pmod{x^r - 1, p}$ then reject;
5. accept the input;
• The test of row 1 removes the powers of odd primes as required by the test of Agrawal (1999)
• The loop of row 3 searches a pair of Sophie Germain primes $q$ and $r$
• Row 3a) tests for Theorem A that $p$ and $r$ are relatively prime
• The loop of row 4 examines primality using a variation of Theorem A (Agrawal, 1999) up to value $2\sqrt{r}\log p$ (AKS, 2002)

• Because Theorem A holds if and only if $p$ is prime, the decision of the algorithm is correct
• The other variations only affect the complexity of the algorithm, not its correctness