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 \( U \): \( \cup S = U \)

- The aim is to find a subcover \( S' \subseteq S \), \( \cup S' = U \), 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 NP-complete.

**Proof.** Obviously minSC ∈ 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 $\text{VC} \leq_p \text{minSC}$ is easy to give. Let $\langle G, k \rangle$ be an instance of the vertex cover in which $G = (V, E)$. We choose the mapping $f$:

$$f(\langle (V, E), k \rangle) = \langle E, V_{\text{inc}} \rangle,$$

where $V_{\text{inc}}$ 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. \hfill $\square$
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

$$\text{App}/\text{Opt} \leq 2.$$

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. $X \leftarrow U; \ C \leftarrow \emptyset$;

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
The greedy algorithm can quite easily be implemented 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 $O(|U| \cdot |S| \cdot \min(|U|, |S|))$.

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:

$$c_u = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

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

$$\sum_{S \in C^*} \sum_{u \in S} c_u \geq \sum_{u \in U} c_u$$

• Combining the above given yields

$$\text{App} \leq \sum_{S \in C^*} \sum_{u \in S} c_u$$
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
\[ \sum_{a \in S'} c_a \leq H(|S'|) \]

Then, by the previous inequality,
\[
\begin{align*}
\text{App} & \leq \sum_{a \in S'} H(|S'|) \\
& \leq |C^*| \cdot H(\max \{|S'|; S' \in S\}) \\
& = \text{Opt} \cdot H(\max \{|S'|; S' \in S\})
\end{align*}
\]

**Lemma** For each $S' \in S$ it holds
\[ \sum_{a \in S'} c_a \leq H(|S'|) \]

**Proof.** Let $S' \in S$ be arbitrary and $i = 1, 2, \ldots, |G|$. 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 \( \sum_{u \in S} c_u = \sum_{i=1}^{k} \frac{1}{n_{i-1} - n_i} \). 

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,

\[
| S_i \setminus (S_1 \cup \cdots \cup S_{i-1}) | \geq | S' \setminus (S_1 \cup \cdots \cup S_{i-1}) | = n_{i-1}
\]

Which further yields

\[
\sum_{u \in S} c_u \leq \sum_{i=1}^{k} \frac{1}{n_{i-1} - n_i} \frac{1}{n_{i-1}}.
\]

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 \( \max \{|S'| : S' \in S\} \) 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, \( \frac{\text{App}}{\text{Opt}} \leq H(d) \)
- 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 
  minSC within \((1-\varepsilon) \ln m\), for any \(\varepsilon > 0\), unless \(NP \subseteq DTIME(n^{\log\log n})\)
• Hence, it is not possible to find an approximation algorithm for 
  minSC that would be significantly better than the greedy one

• Slavik, 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

Course Recap

✔ All computational problems cannot be solved algorithmically
✔ A deterministic finite automaton (DFA) has a unique minimal 
  automaton. The minimal automaton can be constructed in a 
  straightforward manner
✔ Nondeterministic finite automata (NFA) do not recognize more 
  languages than DFAs
✔ A language is regular
  ⇔ it can be recognized with a finite automaton
  ⇔ it can be described with a regular expression
  ⇔ it can be generated with a right-linear grammar
Strings of a regular language can be "pumped"
There are sensible formal languages that are not regular
Context-free languages are a proper superset of regular languages
A language is context-free if and only if it can be recognized with a pushdown automaton (PDA)

By the Church-Turing thesis any problem solvable on a computer can also be solved using a Turing machine (TM)
Variants of Turing machines – including nondeterministic Turing machines – have equal recognition power to the standard single-tape machine

The "efficiency" of different machines varies
Languages generated by unrestricted grammars are equivalent to those recognized by Turing machines
A total Turing machine (decider) halts on every input
A formal language is Turing-recognizable (TR), if it can be recognized with a TM and Turing-decidable, if it has a decider

- $A$ and $B$ decidable $\Rightarrow \bar{A}, A \cup B$ and $A \cap B$ are decidable
- $A, B \in \text{TR} \Rightarrow (A \cup B), (A \cap B) \in \text{TR}$
- $A$ decidable $\Rightarrow A, \bar{A} \in \text{TR}$
- $A \in \text{TR}$, not decidable $\Rightarrow \bar{A} \notin \text{TR}$
\( D = \{ \langle c \rangle \mid c \in \{0, 1\}^* \text{ and } L(M) \in \text{TR} \} \)

E.g., \( A_{\text{DFA}} \), \( E_{\text{DFA}} \), and \( EQ_{\text{DFA}} \) are decidable languages

\( U = \{ \langle M, w \rangle \mid w \in L(M) \} \) is decidable, not decidable

\( \hat{U} = \{ \langle M, w \rangle \mid w \not\in L(M) \} \) is decidable

\( H = \{ \langle M, w \rangle \mid M \downarrow \} \) is decidable, not decidable

\( \hat{H} = \{ \langle M, w \rangle \mid M \downarrow \} \) is decidable

Chomsky hierarchy:

finite \( \subseteq \) regular \( \subseteq \) context-free \( \subseteq \) context-sensitive \( \subseteq \) languages generated by unrestricted grammars (\( \neq \) TR)

\( \text{NE} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \neq \emptyset \} \) is decidable, not decidable

\( \text{REG} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular} \} \) is not decidable

Rice's theorem: All nontrivial semantic properties of Turing machines are undecidable

Linear bounded automaton (LRA) cannot use more work space than that already required by input

\( A_{\text{LRA}} \) is decidable, while \( E_{\text{LRA}} \) is undecidable

\( A \subseteq \Sigma^* \) is reducible to \( B \), \( B \subseteq \Gamma^* \), denoted \( A \preceq_m B \), if there exists a computable function \( f: \Sigma^* \rightarrow \Gamma^* \) s.t.

\[ x \in A \Leftrightarrow f(x) \in B \quad \forall x \in \Sigma^* \]
\( A \subseteq \{0,1\}^* \) is TR-complete, if
1. \( A \in \text{TR} \) and
2. \( B \preceq_m A \) for all \( B \in \text{TR} \)

Language \( U \) is TR-complete

In time complexity analysis of Turing machines one examines the worst case with inputs on certain length

Relating growth rates of functions: \( O, \Theta, o, \Omega \)

The number of tapes does not have a significant impact on the efficiency of a Turing machine

The efficiency difference of a deterministic and nondeterministic TM, on the other hand, is exponential

\[ P = \bigcup_{k \geq 0} \text{DTIME}(n^k + k) \]
\[ \text{EXPTIME} = \bigcup_{k \geq 0} \text{DTIME}(2^{n^k}) \]

\( P \) includes languages that can be decided in time that is polynomial in the length of the input

E.g., finding a directed path in a graph, PATH, and deciding whether two numbers are relatively prime are examples of problems in \( P \)

The corresponding nondeterministic composite classes are \( \text{NP} \) and \( \text{NEXPTIME} \)

For instance, CLIQUE and SUBSET-SUM are problems in \( \text{NP} \)

Problems belonging to \( P \) are solvable in practice
✓ P \subseteq NP. In addition NP contains problems for which no polynomial time algorithm is known
✓ A \subseteq \Sigma^* is polynomial time reducible to B, B \subseteq \Gamma^*, denoted
✓ A \leq_m B, if there exists a polynomial time computable function f: \Sigma^* \rightarrow \Gamma^* s.t.
  \forall x \in \Sigma^*\{ x \in A \iff f(x) \in B \}
✓ A \subseteq \{0,1\}^* is NP-complete, if
  1. A \in NP and
  2. B \leq_m \pi A for all B \in NP
✓ All problems in NP are polynomial time reducible to a NP-complete problem.
✓ If any NP-complete problem is in P, then P = NP

Showing that A \in NP is NP-complete:
1. Select a similar problem B that is known to be NP-complete
2. Give a polynomial time reduction f: B \leq_m \pi A; by Theorem 7.36 also A is NP-complete

NP-complete problems: SAT, CSAT, 3SAT, VC, IS, CLIQUE, Hamiltonian path, TSP, Subset-sum, and minSC

\[
\begin{align*}
\text{PSPACE} & = \bigcup_{k \geq 0} \text{DSPACE}(n^k) \\
\text{NPSPACE} & = \bigcup_{k \geq 0} \text{NSPACE}(n^k) \\
\text{P} & \subseteq \text{NP} \subseteq \left\{ \begin{array}{l}
\text{PSPACE} = \text{NPSPACE} \\
\text{EXPTIME} \subseteq \text{NEXPTIME}
\end{array} \right. \\
\end{align*}
\]
TQBF is PSPACE-complete
So are the "asymptotic" versions of chess and GO

\[ L = \text{DSPACE}(\log n) \]
\[ NL = \text{NSPACE}(\log n) \]

\[ \text{PATH} \in NL \text{ is NL-complete} \]
\[ L = \not= NL \]
\[ NL \subseteq P \]

One can try to approximate an intractable problem efficiently
Vertex cover has an efficient 2-approximation algorithm
Minimum set cover has an efficient \((\ln m + 1)\)-approximation algorithm

THE END

Keep in mind that programming video games also requires understanding computationally demanding problems …