Theoretical Computer Science - Bridging Course
Winter Term 2016
Sample Solution for Exercise Sheet 9

Exercise 1: The class \( \mathcal{NP} \) (2+3+2+0 points)

\( \mathcal{NP} \) is the set of languages which can be decided by a non-deterministic Turing machine whose run-time (minimum number of steps required to reach the accepting state) can be bounded by \( p(n) \), where \( p \) is a polynomial and \( n \) the size of the respective input (problem instance).

A non-deterministic Turing machine \( N \) is typically used for a non-deterministic procedure called 'guess and check'. First \( N \) 'guesses' a solution for given problem instance \( s \), which leads to the correct answer of the decision problem (\( s \in L \) or \( s \notin L \))^1. Then the solution guessed by \( N \) is verified by a deterministic Turing machine \( D \) which accepts if and only if the solution is correct (\( D \) sifts out wrong guesses).

Show that the following problems are in \( \mathcal{NP} \) by describing the solution that the non-deterministic machine is expected to guess, and giving a deterministic algorithm that verifies it in polynomial time.

Use the \( O \) notation to bound the run time. Since it is easy (i.e. possible in polynomial time) to decide whether inputs are well-formed instances, your algorithm only needs to consider well-formed inputs.

(a) \( \text{Clique} = \{\langle G, k \rangle \mid G \text{ is a Graph with a complete subgraph with } k \text{ nodes} \} \)

(b) \( \text{ISO} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are isomorphic graphs} \} \)

Remark: Two graphs \( G, H \) are isomorphic, if a bijective mapping \( f : V(G) \to V(H) \) exists such that \( u, v \in V(G) \) are adjacent in \( G \), if and only if \( f(u), f(v) \in V(H) \) are adjacent in \( H \).

(c) \( \text{3-SAT} = \{\langle \phi \rangle \mid \text{bool. formula } \phi \text{ in 3-CNF has assignment of variables s.t. } \phi \text{ evaluates to TRUE.} \} \)

Remark: \( \phi \) is in 3-CNF if it is of the form \( C_1 \land \ldots \land C_m \), where \( C_i = L_{i,1} \lor L_{i,2} \lor L_{i,3} \) are clauses of at most three literals \( L_{i,j} = x_k \) or \( L_{i,j} = \overline{x_k} \) of negated or non-negated variables \( x_1, \ldots, x_n \) of \( \phi \).

(d) Show that \( \text{2-SAT} \in \mathcal{P} \) (voluntary).

Hint: Clauses with two literals can be transformed into the form \( A \to B \).

Solution

(a) \textbf{Guess:} We use a non-deterministic Turing machine to guess a subset \( C \subseteq V \) of the nodes of \( G = (V, E) \). Then a deterministic algorithm (Turing machine) verifies that the set \( C \) in fact forms a clique of size \( k \). This is done as follows.

\textbf{Check:} First the algorithm checks if \( |C| \geq k \) which can be done in \( O(|V|) \subseteq O(n) \) (where we define \( n \) as the size of the input \( \langle G, k \rangle \)).

Second the algorithm checks whether \( \{u, v\} \in E \) for all nodes \( u, v \in C \) with \( u \neq v \). This can be done in \( O(|V|^2) \subseteq O(n^2) \).

If both checks turn out positive the algorithm accepts else rejects the guessed solution. The algorithm requires \( O(n^2) \) steps and thus \( \text{Clique} \in \mathcal{NP} \).

1A Turing machine that 'guesses' solutions can be constructed as as follows: \( N \) writes a random input sequence and halts. Since a non-deterministic Turing machine 'explores all possibilities' it will give the correct solution in polynomial time, if it exists.
(b) **Guess:** Non-deterministically we guess a mapping \( f : V(G) \to V(H) \) for the input \( \langle G, H \rangle \). To verify that the mapping is in fact a graph-isomorphism we do the following.

**Check:** *First our algorithm checks whether the mapping is bijective* by marking the nodes \( f(u) \) for every \( u \in V(G) \). If a node \( v \in V(H) \) is marked twice, \( f \) can not be injective and the algorithm rejects the guess and terminates.

After the marking process is finished, the algorithm verifies if all nodes in \( V(H) \) have been marked. If not \( f \) can not be surjective and the algorithm rejects and halts. The marking process can accomplished in \( O(|V(G)| + |V(H)|) \subseteq O(n) \) (again \( n \) denotes the size of input \( \langle G, H \rangle \)).

*Next the algorithm checks the property* \( \{u, v\} \in E(G) \iff \{f(u), f(v)\} \in E(H) \). For all edges \( \{u, v\} \in E(G) \) we check \( \{f(u), f(v)\} \in E(H) \). If \( \{f(u), f(v)\} \notin E(H) \) the algorithm rejects and halts since \( \{u, v\} \in E(G) \) has not corresponding edge in \( H \). Else the edge \( \{f(u), f(v)\} \) is marked. This process takes at most \( O(|E(G)| \cdot |E(H)|) \subseteq O(n^2) \) time.

If there is an unmarked edge \( e \in E(H) \) after the marking process, then edge \( e \) has no corresponding edge in \( E(G) \), thus \( f \) is no graph isomorphism and the algorithm rejects and stops. However, if all checks turn out positive, then \( G \) and \( H \) are in fact isomorphic via \( f \) and the algorithm accepts.

The whole verification process takes at most \( O(n^2) \) time, thus \( ISO \in \mathbb{NP} \).

(c) **Guess:** For an input \( \langle \phi \rangle \) we guess an interpretation \( I : \{x_1, \ldots, x_n\} \to \{T, F\} \) of the atoms \( x_1, \ldots, x_n \) occurring in \( \phi \).

**Check:** Our deterministic verification process checks if \( \phi \) evaluates to \( T \) (true) with interpretation \( I \) (\( I \models \phi \)). This can be done easily by checking for each clause \( C_i \) (in \( O(m) \)) whether one of the literals \( L_{i,j} \) evaluates to \( T \).

If one of the \( m \) clauses is false then \( I \) is rejected. If all clauses evaluate to \( T \) under \( I \) then \( \phi \) evaluates to \( T \) too and is therefore satisfiable and thus \( I \) is accepted. This whole process takes \( O(n \cdot m) \) time and is thus polynomial in the size of the input. Hence \( 3\text{-SAT} \in \mathbb{NP} \).

(d) The result \( 2\text{-SAT} \in \mathbb{P} \) may be surprising since \( 3\text{-SAT} \in \mathbb{NP} \). This is due to the fact that \( 2\text{-SAT} \) can be boiled down to a graph problem that can be solved in poly. time. To show \( 2\text{-SAT} \in \mathbb{P} \), we have to devise a deterministic algorithm that checks if a boolean formula \( \phi \) in \( 2\text{-CNF} \) is satisfiable.

**Removal of 1-clauses in** \( \phi \): First, we check if there are 1-clauses \( C_i = x_j \) or \( C_i = \overline{x_j} \) in \( \phi \) which consist of just one literal. If yes, we have to interpret the according atom \( x_j \) as \( T \) (true) for \( C_i = x_j \) and as \( F \) (false) for \( C_i = \overline{x_j} \) in order to satisfy \( \phi \).

Thus at first our algorithm repeatedly searches \( \phi \) for 1-clauses \( x_j \) or \( \overline{x_j} \) removes them, and replaces all other occurrences of \( x_j \) with its according interpretation \( T \) if the removed 1-clause was \( x_j \) and \( F \) if it was \( \overline{x_j} \). After each removal of a 1-clause, \( \phi \) is simplified such that occurrences of \( T, F \) are removed using the respective logical rules (this may cause further literals or clauses to vanish).

If during simplification a 1-clause happens to evaluate to \( F \), then \( \phi \) is unsatisfiable and has to be rejected. If all clauses are removed we have found an interpretation that satisfies \( \phi \) and hence our algorithm accepts. Each removal of a 1-clause can be accomplished in \( O(n + m) \) time and since there are \( m \) clauses the total time for 1-clause removal is bounded from above by \( O(m(n + m)) \).

**Check satisfiability of** \( \phi \) **in** \( 2\text{-CNF without 1-clauses:** If the algorithm did not terminate up to this point, then \( \phi \) contains only 2-clauses. We transform \( \phi \) into a directed graph \( G_\phi = (V, E) \) as follows. All negated and non-negated atoms \( x_1, \ldots, x_n, \overline{x_n} \) form the set of nodes \( V \).

Using the hint, each 2-clause \( C_i \) can be written as \( C_i = L_{i,1} \lor L_{i,2} = \overline{L_{i,1}} \rightarrow L_{i,2} = \overline{L_{i,1}} \rightarrow L_{i,1} \). For each clause we add directed edges \( (L_{i,1}, L_{i,2}), (L_{i,2}, L_{i,1}) \) to \( E \). Formula \( \phi \) is unsatisfiable iff \( x_j \in V \) exists, s.t. there are paths from \( x_j \) to \( \overline{x_j} \) and from \( \overline{x_j} \) to \( x_j \) in \( G_\phi \) (see Lemma 1 below).

To check whether \( G_\phi \) has such a pair of paths, we conduct breadth first searches (BFS, cf. solution of exercise sheet 8) starting from all nodes \( x_j, \overline{x_j} \) (\( j = 1, \ldots, n \)) of \( G_\phi \) to find the respective negated node. Whenever we find a path for which we previously found one in the other direction, we reject \( \phi \) and terminate. After the searches are finished we accept \( \phi \). Taken together, the at most
2n searches have a running time of $O(n(m + n))$. The total running time of the algorithm is $O((m + n)^2)$. Hence 2-SAT $\in \mathcal{P}$. It remains to show the equivalency we stated above.

**Lemma 1.** Formula $\phi$ in 2-CNF consisting only of 2-clauses is unsatisfiable if and only if $x_j \in V$ exists, such that there are paths from $x_j$ to $\overline{x_j}$ and from $\overline{x_j}$ to $x_j$ in $G_\phi$.

**Proof.** On one hand, if there are paths in $G_\phi$ as stated in the lemma, we have chains of implications $x_j \rightarrow \ldots \rightarrow \overline{x_j}$ and $\overline{x_j} \rightarrow \ldots \rightarrow x_j$. Due to the transitivity of the operation ‘$\rightarrow$’, the formula $\phi$ entails $(x_j \rightarrow \overline{x_j}) \land (\overline{x_j} \rightarrow x_j) \equiv x_j \leftrightarrow \overline{x_j}$ which is unsatisfiable.

On the other hand, if $\phi$ is unsatisfiable, then for every interpretation there must be a 2-clause $L' \lor L''$ that cannot be satisfied without making another 2-clause false. Thus $L'$ and $L''$ must be entailed by a series of implications

$$L'_1 \rightarrow \ldots \rightarrow L'_k \rightarrow L', \quad L''_1 \rightarrow \ldots \rightarrow L''_l \rightarrow L''$$

(1)

since otherwise there nothing prevents us from setting either $L_1$ or $L_2$ as true, thus satisfying $\phi$.

However, if all literals $L'_1, \ldots, L'_k$ are set to false, which would still give us the opportunity to interpret $L'$ as true thus satisfying $\phi$, a contradiction. Therefore there must be two literals $L'_v, L'_w$ $(1 \leq v < w \leq k)$ for which a clause $L'_v \lor L'_w$ exists, which becomes false if we set $L'_1, \ldots, L'_k$ to false. Analogously we have a 2-clause $L''_x \lor L''_y$ $(1 \leq x < y \leq l)$. From these 2-clauses we derive

$$L'_v \rightarrow L'_w, \quad L''_x \rightarrow L''_y.$$ 

(2)

Using contraposition ($A \rightarrow B \iff \overline{B} \rightarrow \overline{A}$) we can ‘reverse’ the paths given in Equation 1, obtaining

$$L' \rightarrow L'_k \rightarrow \ldots \rightarrow L'_1, \quad L'' \rightarrow L''_l \rightarrow \ldots \rightarrow L''_1$$

(3)

Moreover, from the original 2-clause $L' \lor L''$ we derive

$$L' \rightarrow L'', \quad L'' \rightarrow L'.$$ 

(4)

Now we have everything we need to show that there is a path from $L'$ to $L'$.

$$L' \rightarrow L'_k \rightarrow \ldots \rightarrow L'_{v+1} \rightarrow L'_v \rightarrow L'_w \rightarrow L'_{w+1} \rightarrow \ldots \rightarrow L'_k \rightarrow L'. $$

(1)

And similarly we obtain a path from $L'$ to $L'$ by going via $L''$ using Equation 4.

$$L' \rightarrow L'' \rightarrow L''_l \rightarrow \ldots \rightarrow L''_{x+1} \rightarrow L''_x \rightarrow L''_y \rightarrow L''_{y+1} \rightarrow \ldots \rightarrow L''_l \rightarrow L'' \rightarrow L'. $$

(4)

Exercise 2: The class $\mathcal{NP}C$ (3+4 points)

Let $L_1, L_2$ be languages (problems) over alphabets $\Sigma_1, \Sigma_2$. Then $L_1 \leq_p L_2$ ($L_1$ is polynomially reducible to $L_2$), iff a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ exists, that can be calculated in polynomial time and

$$\forall s \in \Sigma_1 : s \in L_1 \iff f(s) \in L_2.$$ 

Language $L$ is called $\mathcal{NP}$-hard, if all languages $L' \in \mathcal{NP}$ are polynomially reducible to $L$, i.e.

$$L \mathcal{NP} \text{-hard} \iff \forall L' \in \mathcal{NP} : L' \leq_p L.$$ 

The reduction relation '$\leq_p$' is transitive ($L_1 \leq_p L_2$ and $L_2 \leq_p L_3 \Rightarrow L_1 \leq_p L_3$). Therefore, in order to show that $L$ is $\mathcal{NP}$-hard, it suffices to reduce a known $\mathcal{NP}$-hard problem $\overline{L}$ to $L$, i.e. $\overline{L} \leq_p L$.

Finally a language is called $\mathcal{NP}$-complete ($\iff : L \in \mathcal{NP}$), if
1. $L \in \mathcal{NP}$ and
2. $L$ is $\mathcal{NP}$-hard.

(a) Show $\text{HALFCLIQUE} := \{(G) \mid \text{Graph } G \text{ has clique of size at least } \lceil n/2 \rceil \} \in \mathcal{NP}$.
Use that $\text{CLIQUE} := \{(G,k) \mid \text{Graph } G \text{ has clique of size at least } k \} \in \mathcal{NP}$.

Hint: Describe an algorithm (with poly. run-time!) that transforms $G$ and $k$ into a graph $G'$ by adding nodes and connecting them with edges in a suitable manner, s.t. a clique of size $k$ in $G$ becomes a clique of size $\lceil n/2 \rceil$ in $G'$ and vice versa(!).

(b) Show $\text{DOMINATINGSET} := \{(G,k) \mid \text{Graph } G \text{ has a dominating set of size at most } k \} \in \mathcal{NP}$.
Use that $\text{VERTEXCOVER} := \{(G,k) \mid \text{Graph } G \text{ has a vertex cover of size at most } k \} \in \mathcal{NP}$.

Remark: A dominating set is a subset of nodes of $G$ such that every node not in the subset is adjacent to some node in the subset. A vertex cover is a subset of nodes of $G$ such that every edge of $G$ is adjacent to a node in the subset.

Hint: Transform a Graph $G$ into a Graph $G'$ such that a vertex cover of $G$ will result in a dominating set $G'$ and vice versa(!). Note that a dominating set is not necessarily a vertex cover (consider isolated notes).

### Solution

(a) Guess and Check: We show $\text{HALFCLIQUE} \in \mathcal{NP}$ and guess a subset of nodes $C \subseteq V$ of $G = (V,E)$. Then we check $|C| \geq \lceil n/2 \rceil$ $(n := |V|)$ and if $C$ induces a complete subgraph in $G$. This is possible in $O(n+m)$ $(m := |E|)$ time which is polynomial in the size of the input $(G,k)$.

Polynomial reduction of $\text{CLIQUE}$ to $\text{HALFCLIQUE}$: The hard part is showing that $\text{HALFCLIQUE}$ is $\mathcal{NP}$-hard. For this purpose we show that $\text{HALFCLIQUE}$ is a harder problem than $\text{CLIQUE}$, and since the latter is known to be $\mathcal{NP}$-hard, we obtain the same result for $\text{HALFCLIQUE}$.

In order to show that $\text{HALFCLIQUE}$ is in fact harder than $\text{CLIQUE}$ we make the polynomial reduction $\text{CLIQUE} \leq_p \text{HALFCLIQUE}$. For this purpose we define a function $f$ which maps instances $(G,k)$ of $\text{CLIQUE}$ to instances $(G')$ of $\text{HALFCLIQUE}$ (as usual we neglect strings that do not represent well-formed instances), and is computable in polynomial time.

For $G = (V,E)$ we define the graph $(G') = f((G,k))$ as follows. If $k \geq \lceil n/2 \rceil$ $(n := |V|)$ we have to decrease the size of the clique in relation to the number of nodes in $G'$. Thus we define $G'$ as $G$ with the difference that we add $2k - n$ isolated nodes to $G'$.

If $k < \lceil n/2 \rceil$ we increase the size of a potential $k$-clique in $G$ such that it becomes a clique of size at least $\lceil n/2 \rceil$ in $G'$. For this purpose we define $G'$ as $G$ and add a number of $x$ nodes to $G'$, connect them among themselves and each of them to all other nodes. Any $k$-clique in $G$ becomes a $(k+x)$-clique in $G'$ thus we need to choose $x$ such that $\lceil (n+x)/2 \rceil = k+x$ which is synonymous to $\lceil (n-2k-x)/2 \rceil = 0$. Therefore $x = n-2k$ does the trick.

In both cases, the construction of $G'$ can obviously be done in linear time in the input size. It remains to prove the equivalency

$$(G,k) \in \text{CLIQUE} \iff f((G,k)) = (G') \in \text{HALFCLIQUE}.$$ 

If $k \geq \lceil n/2 \rceil$ and $G$ has a $k$-clique, then we add isolated nodes to $G'$ which has a total of $2k - n + n = 2k$ nodes thus the $k$-clique in $G$ becomes a half clique in $G'$. And obviously if $G'$ has a half-clique it must be of size $2k/2 = k$ which can not contain any of the isolated nodes we added to $G'$ (except in the trivial case $k = 1$), thus $G$ has a $k$-clique.

If $k < \lceil n/2 \rceil$ and $G$ has a $k$-clique, then $G'$ has a $(k+x) = (n-k)$-clique. Since $G'$ has $n+x = 2n - 2k$ nodes the $(n-k)$-clique forms a half-clique in $G'$. Conversely, if $G'$ has a half-clique, it must be of size $n-k$ and contains at most all of the $x$ 'omni-connected' nodes we added to $G'$. Thus $G$ has a clique of size at least $n-k-x = k$. 

4
We showed \( \text{HALFCLIQUE} \in \mathcal{NP} \) and that \( \text{HALFCLIQUE} \) is \( \mathcal{NP} \)-hard, thus \( \text{HALFCLIQUE} \in \mathcal{NP} \cap \mathcal{NP} \).

(b) **Guess and Check:** We show that \( \text{DOMINATINGSET} \in \mathcal{NP} \). Given a subset \( D \subseteq V \) of \( G = (V, E) \), we can verify in \( \mathcal{O}(mn) \) \( (n := |V|, m := |E|) \) if all nodes are in \( D \) or adjacent to a node in \( D \).

**Polynomial reduction of \text{VERTEXCOVER} to \text{DOMINATINGSET}:** We define a function \( f \) that can be computed in polynomial time and transforms an instance \( (G, k) \) of \( \text{DOMINATINGSET} \) into an instance \( f((G, k)) = (G', k) \), such that \( G \) has a vertex cover of size \( k \), iff \( G' \) has a vertex cover of size \( k \), i.e.,

\[
(G, k) \in \text{VERTEXCOVER} \iff f((G, k)) = (G', k) \in \text{DOMINATINGSET}.
\]

For \( G = (V, E) \) we construct \( G' = (V', E') \) as follows. Initially, we set \( V' := V, E' := E \). For each edge \( \{u, v\} \in E \) we add an additional node \( w \) to \( V' \) and add the edges \( \{u, w\}, \{u, w\} \) to \( E' \) (i.e., \( G' \) has a triangle with nodes \( u, v, w \)). Furthermore we remove all isolated nodes from \( V' \). The construction of \( G' \) can be accomplished in \( \mathcal{O}(m + n) \), since we add at most \( \mathcal{O}(m) \) nodes and edges and remove at most \( \mathcal{O}(n) \). It remains to prove the equivalency stated above.

\( \implies \): Let \( (G, k) \) be such, that \( G \) has a vertex cover \( C \) of size at least \( k \). We show that \( D := C \cap V' \) (\( D \) equals \( C \) minus isolated nodes) is a dominating set of size \( k \) in \( G' \). We know that for every edge \( \{u, v\} \in E \) either \( u \in C \) or \( v \in C \) (or both). Therefore, if a node \( w \) was added to \( V' \) during the construction due to an edge \( \{u, v\} \in E \) it is dominated by (adjacent to) either \( u \in D \) or \( v \in D \).

All other nodes in \( v \in V' \) also have an adjacent edge \( \{u, v\} \in E \subset E' \) (recall that we removed isolated nodes from \( V' \)). Therefore \( v \in C \) (and thus \( v \in D \)), or \( v \) is dominated by a node in \( C \) (hence dominated by one in \( D \)). Since \( |D| \leq |C| \leq k \) it holds that \( (G', k) \in \text{DOMINATINGSET} \).

\( \iff \): Let the transformed instance \( f((G, k)) = (G', k') \) be such that \( G' \) has a dominating set \( D \) with \( |D| \leq k \). We show that we can construct a vertex cover \( C \) of size at most \( k \) in the original graph \( G \) from \( D \). Let \( \{u, v\} \) be an arbitrary edge of \( G \). Due to the way \( G' \) was constructed, there is a triangle formed by the nodes \( u, v, w \) where \( w \) is only connected to \( u \) and \( v \).

This means that at least one of the three cases holds: \( w \) is dominated by \( u \in D \) or by \( v \in D \) or it holds that \( w \in D \). In the first two cases we add \( u \) or \( v \) respectively to \( C \) (whichever was in \( D \)). In the third case we simply add one of the two nodes \( u \) or \( v \) instead. In all cases \( \{u, v\} \) is covered. Since we add at most \( |D| \) nodes to \( C \) it holds that \( |C| \leq k \).

This concludes the proof and in summary we have \( \text{DOMINATINGSET} \in \mathcal{NP} \).

**Exercise 3: Complexity Classes: Big Picture (1+2+3+0+0+0 points)**

(a) Why is \( \mathcal{P} \subseteq \mathcal{NP} \)?

(b) Show that \( \mathcal{P} \cap \mathcal{NP} = \emptyset \) if \( \mathcal{P} \neq \mathcal{NP} \).

Hint: Assume that there exists a \( L \in \mathcal{P} \cap \mathcal{NP} \) and derive a contradiction to \( \mathcal{P} \neq \mathcal{NP} \).

(c) Give a Venn Diagram showing the sets \( \mathcal{P}, \mathcal{NP}, \mathcal{NPC} \) for both cases \( \mathcal{P} \neq \mathcal{NP} \) and \( \mathcal{P} = \mathcal{NP} \).

Remark: Use the results of (a) and (b) even if you did not succeed in proving those.

(d) Show that the Halting Problem \( H \) is \( \mathcal{NP} \)-hard. You can use that

\[
\text{SAT} = \{ \langle \phi \rangle \mid \text{bool. formula } \phi \text{ has assignment of variables s.t. } \phi \text{ evaluates to TRUE.} \}
\]

is \( \mathcal{NP} \)-hard. (voluntary)

Hint: For any boolean formula \( \phi \) give an algorithm \( A \) that stops if and only if \( \phi \) is satisfiable.

(e) Argue why \( H \notin \mathcal{NP} \). (voluntary)

Hint: You can use results from previous exercise sheets.

(f) Add the class of \( \mathcal{NP} \)-hard problems to the Venn Diagrams from exercise (c). (voluntary)
Solution

(a) If \( L \in \mathcal{P} \) there is a deterministic Turing machine that decides \( L \) in polynomial time. Then \( L \in \mathcal{NP} \) simply by definition since a deterministic Turing machine is a special case of a non-deterministic one.

(b) As the hint suggests we assume that there is a language \( L \) which is \( \mathcal{NP} \)-complete and simultaneously solvable in polynomial time by a Turing machine. We use this language \( L \) to show that \( \mathcal{NP} \subseteq \mathcal{P} \), which together with (a) implies \( \mathcal{NP} = \mathcal{P} \), i.e., a contradiction to our premise \( \mathcal{NP} \neq \mathcal{P} \). Hence \( L \) cannot exist if \( \mathcal{NP} \neq \mathcal{P} \).

So let \( L' \in \mathcal{NP} \). We want to show that \( L' \) is in \( \mathcal{P} \) to obtain the contradiction. Since \( L \) is also \( \mathcal{NP} \)-hard, we can solve the decision problem \( L' \) via \( L \) by using the polynomial reduction \( L' \leq_p L \).

In particular for any string \( s \in L' \) we have the equivalency \( s \in L' \iff f(s) \in L \), where \( f \) is induced by the reduction.

We construct a Turing machine for \( L' \) that runs in poly. time. For instance \( s \) it first computes \( f(s) \) in polynomial time and then uses the Turing machine for \( L \) as a subroutine to return the answer of \( f(s) \in L \) in polynomial time. In total, we require only polynomial time to decide \( s \in L' \) which means \( L' \in \mathcal{P} \).

(c) See Figure 1. For the case \( \mathcal{P} = \mathcal{NP} \), the notion of \( \mathcal{NP} \)-hardness becomes utterly meaningless since the class \( \mathcal{NP} \) can be polynomially reduced to every other language except \( \Sigma^* \) and \( \emptyset \). In order to show that \( L' \leq_p L \) for an \( L \neq \Sigma^*, \emptyset \) and for all \( L' \in \mathcal{NP} = \mathcal{P} \), we need show that there is a polynomially computable mapping \( f \) such that \( \forall s \in \Sigma^* : s \in L' \iff f(s) \in L \).

But such a mapping \( f \) always exists for \( L \neq \Sigma^*, \emptyset \). We simply have to use a known ‘yes-instance’ \( y \in L \) and a ‘no-instance’ \( n \notin L \). Then we define for \( s \in \Sigma^* \) that \( f(s) := y \) if \( s \in L' \) and \( f(s) := n \) if \( s \notin L' \). This obviously fulfills the above equivalency. Moreover \( f \) is polynomially computable since we can find out whether \( s \in L' \) in polynomial time.

(d) We give a mapping \( f \) that maps a given instance \( \langle \phi \rangle \) of SAT to an algorithm \( \langle A \rangle \) (i.e., a Turing machine which is an instance of \( H \)). For \( \phi \) we describe what \( A \) does. First \( A \) extracts all atoms \( x_1, \ldots, x_n \) from \( \phi \). Then it loops over all possible interpretations of these atoms and evaluates \( \phi \). If \( A \) finds an interpretation that evaluates \( \phi \) to true then \( A \) halts. If \( A \) has iterated over all interpretations without halting then \( A \) starts over, thus going into an endless loop.

Since \( A \) evaluates \( \phi \) with all possible interpretations, \( A \) halts if and only if \( \phi \) is satisfiable (otherwise it loops indefinitely). This proves the equivalency we needed to show. Furthermore \( f \) is polynomially computable since the ‘source-code’ of \( A \) has a length of \( O(n) \). Note that \( A \) itself has obviously exponential runtime in \( n \), if it terminates at all. However, this is of no concern, since we only have to show that the creation of the ‘source-code’ of \( A \) is possible in poly. time!

(e) This is due to the definition of the class \( \mathcal{NP} \), which consists of languages which have non-deterministic deciders running in poly. time. Recall that a decider does not need to be deterministic, in fact if a non-det. decider exits, there is also a deterministic one (cf. lecture). So if \( H \in \mathcal{NP} \) were true, we could decide \( H \), which we already disproved on the last exercise sheet.

(f) See Figure 1. With \( H \) we have a witness showing that the class of \( \mathcal{NP} \)-hard languages and \( \mathcal{NP}C \) are different sets.
Figure 1: Venn-Diagram of the Language classes $\mathcal{P}, \mathcal{NP}, \mathcal{NPC}, \mathcal{NP}$-hard.