Algorithmics I - Tutorial Sheet 4

NP-completeness

---

1. Show that the following problems are in **NP**.

   (a) Satisfiability (**SAT**)
   Instance: a Boolean expression B in CNF.
   Question: is B satisfiable?

   > **Solution:** An informal proof that **SAT** belongs to the class **NP** is as follows. A certificate for **SAT** would be a truth assignment, i.e. a function f that assigns every variable in the given Boolean formula B a truth value. To verify the certificate, we need to check that each clause of B contains at least one literal that is `true` under f. Clearly this check can be carried out in polynomial time.

   (b) **CLIQUE**
   Instance: a graph G and a positive integer k.
   Question: does G contain a clique of size $\geq k$?

   > **Solution:** An informal proof that **CLIQUE** belongs to the class **NP** is as follows. A certificate for **CLIQUE** would be a subset of vertices S. To verify the certificate, we need to check that every pair of vertices in S forms an edge in the given graph G. We would also need to check that $|S| \geq k$. Clearly both tests can be carried out in polynomial time.

   (c) Independent Set (**IND-SET**)
   Instance: a graph G and a positive integer k.
   Question: does G contain an independent set of size $\geq k$?
   (An independent set is a set of vertices with the property that no two vertices in the set form an edge.)

   > **Solution:** An informal proof that **IND-SET** belongs to the class **NP** is as follows. A certificate for **IND-SET** would be a subset of vertices S. To verify the certificate, we need to check that no pair of vertices in S forms an edge in the given graph G. We would also need to check that $|S| \geq k$. Clearly both tests can be carried out in polynomial time.

   (d) Vertex Cover (**VC**)
   Instance: a graph G and a positive integer k.
   Question: does G contain a vertex cover of size $\leq k$?
   (A vertex cover is a set of vertices with the property that every edge of the graph has a vertex in the set.)

   > **Solution:** An informal proof that **VC** belongs to the class **NP** is as follows. A certificate for **VC** would be a set of vertices S. To verify this certificate, we would need to check that every edge of G has at least one of its end-points in S. We would also need to check that $|S| \leq k$. Clearly both of tests can be carried out in polynomial time.

2. The Graph Colouring Optimisation Problem (**GCOP**) is to find the smallest value of k for which a given graph is k-colourable. Prove that the **GCOP** problem can be solved by a polynomial-time algorithm if and only if the Graph Colouring Decision Problem (**GCDP**) is a member of the class **P**.

Solution: A solution to an instance of **GCOP** automatically provides a solution to corresponding instances of **GCDP** for any value of the target value k, by simply comparing the minimum number of colours returned by **GCOP** with k. Therefore, if **GCOP** is polynomial-time solvable, then so is **GCDP**. On the other hand, suppose we have a polynomial-time algorithm for **GCDP**. We show that the minimum number of colours that suffice to colour the vertices of G can be found in polynomial time. This is achieved by applying the **GCDP** algorithm repeatedly for a sequence of target values, namely $k = 1, 2, \dots, n$ until the answer '`yes`' is returned. At this point, the current value of k is the solution to the instance of **GCOP**. Now, if the complexity of the **GCDP** algorithm is $\mathcal{O}(p(n))$ (where $p(\cdot)$ is a polynomial and n is the number of vertices of the graph), then, since the **GCDP** algorithm is called with at most n distinct values of k, the complexity of the **GCOP** algorithm is $\mathcal{O}(n{\cdot}p(n))$, and hence is also polynomial in n.

Note: the $\mathcal{O}(n{\cdot}p(n))$ complexity could be reduced to $\mathcal{O}(p(n){\cdot}\log n)$ by performing a binary search on the possible values of k, but this is not necessary for the purposes of the question.

3. Suppose there is a polynomial-time algorithm for the **CLIQUE** decision problem. Show how that algorithm could be used to find, in polynomial time, an actual clique of maximum size in a given graph.

Solution: Suppose we have a polynomial-time algorithm for **CLIQUE**. Think of this a 'black box' which, given a graph G and a target k, determines in polynomial-time whether G has a clique of size greater than or equal to k.

The first step is to determine the size of the largest clique in G. To do this, make a sequence of calls to the black box, with target values that carry out a binary search on the interval $1, 2, \dots, n$, where n is the number of vertices in G. This involves $\mathcal{O}(\log n)$ calls of the black box, so is clearly completed in polynomial time, and determines the size, say m, of the largest clique.

We then search for a clique of size m as follows.

Choose an arbitrary vertex v in G. Form the graph G_v from G by deleting v and all vertices not adjacent to the vertex v, together with their incident edges. Now, G has a clique of size m that contains v if and only if G_v has a clique of size $m{-}1$. We now call the blackbox with G_v and $m{-}1$ and proceed as follows:

- if the answer is '`yes`', we can accept v as part of our maximum clique and it remains to find a clique of size $m{-}1$ in G_v;
- if the answer is '`no`', we can discard v, and it remains to find a clique of size m in $G\backslash\{v\}$ (where $G\backslash\{v\}$ is the graph obtained by deleting v and all incident edges).

In either case, we can proceed to use the black box in the same way as before. This step can be achieved in $\mathcal{O}(n^2 + p(n))$ time, where $p(n)$ is the (polynomial-time) complexity of consulting the 'black box'. The $\mathcal{O}(n^2)$ comes from constructing G_v and $G\backslash\{v\}$ as each takes $\mathcal{O}(n^2)$ when using an adjacency matrix representation.

In this second phase, each call to the black box either discards a vertex or increases the size of the growing clique. So at most n calls are needed, since a vertex cannot be both discarded and added to the clique. Hence the overall algorithm is $\mathcal{O}(n^3 + n{\cdot}p(n))$, i.e. polynomial-time as required.

4. Using the definitions in Question 1:

   (a) describe a polynomial-time reduction from **CLIQUE** to **IND-SET**;

   Solution: Given an instance (G, k) of **CLIQUE** ($G = (V, E)$ is the graph and k is the target value), construct the instance (G', k) of **IND-SET**, where G' is the complementary graph of G, i.e $\{v, w\}$ is an edge of G' if and only if it is not an edge of G. Clearly this reduction can be carried out in time polynomial in the size of G.

   The correctness of the reduction follows from showing: S is a clique in G if and only if S is an independent set in G'.

   - "$\Rightarrow$" we show if S is an independent set of G', then S is a clique of G. Consider any $v, w \in S$, it is sufficient to show $\{v, w\}$ an edge of G. For a contradiction, suppose $\{v, w\}$ is not an edge of G, then by construction $\{v, w\}$ is an edge of G' which contradicts S being an independent set of G' as required.
   - "$\Leftarrow$" we show if S is a clique of G, then S is an independent set of G'. Consider any $v, w \in S$, it is sufficient to show $\{v, w\}$ is not an edge of G'. For a contradiction, suppose $\{v, w\}$ is an edge of G', then by construction $\{v, w\}$ is not an edge of G, which contradicts S being a clique of G as required.

   Hence, the construction is a polynomial-time reduction from **CLIQUE** to **IND-SET**.

   (b) describe a polynomial-time reduction from **CLIQUE** to **VC**.

   Solution: Given an instance (G, k) of **CLIQUE** ($G = (V, E)$ is the graph and k is the target value), we construct the instance $(G', n{-}k)$ of **VC**, where G' is the complementary graph of G and n is the number of vertices in G. Clearly this reduction can be carried out in time polynomial in the size of G.

   The correctness of the reduction follows from showing: S is a clique in G if and only if $V\backslash S$ is an vertex cover of G'.

   - "$\Rightarrow$" we show if $V\backslash S$ is an vertex cover of G', then S is a clique of G. Consider any $v, w \in S$, it is sufficient to show $\{v, w\}$ an edge of G. For a contradiction, suppose $\{v, w\}$ is not an edge of G, then by construction $\{v, w\}$ is an edge of G' and since $v, w \notin V\backslash S$ and $V\backslash S$ is a vertex cover of G' we have a contradiction as required.
   - "$\Leftarrow$" we show if S is a clique of G, then $V\backslash S$ is an vertex cover of G'. For any edge $\{v, w\}$ of G' it is sufficient to show that either v or w is in $V\backslash S$. If $v \in V\backslash S$, then we are done. On the other hand, if $v \in S$, i.e v is an element of the clique, then, since $\{v, w\}$ is an edge of G', by definition it is not an edge of G which means w cannot be in the clique, and hence $w \in V\backslash S$ as required.

   Thus, G has a clique of size greater than or equal to k if and only if G' has a vertex cover of size less than or equal to $n{-}k$. Hence we have a polynomial-time reduction from **CLIQUE** to **VC**.

Hint: in both cases, consider the complement graph $G' = (V, E')$ of graph $G = (V, E)$, i.e. the graph with the same vertex set but with edge set $E' = \{\{u, v\} \mid \{u, v\} \notin E\}$. In particular, for some small examples, look at the relationship between a clique in G and an independent set/vertex cover in the complement graph.

5. Derive a 3-CNF expression (i.e. a CNF expression with exactly 3 literals in each clause) equivalent to the following expression (in terms of satisfiability):

$$B = x_1 \wedge (\neg x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee x_3)$$

Is it satisfiable?

Solution: Convert to 3-CNF by replacing each clause as follows:

$$\begin{aligned}
x_1 &\mapsto (x_1 \vee y_1 \vee y_2) \wedge (x_1 \vee y_1 \vee \neg y_2) \wedge (x_1 \vee \neg y_1 \vee y_2) \wedge (x_1 \vee \neg y_1 \vee \neg y_2) \\
(\neg x_1 \vee x_2 \vee x_3 \vee x_4) &\mapsto (\neg x_1 \vee x_2 \vee y_3) \wedge (x_3 \vee x_4 \vee \neg y_3) \\
(\neg x_2 \vee \neg x_3) &\mapsto (\neg x_2 \vee \neg x_3 \vee y_4) \wedge (\neg x_2 \vee \neg x_3 \vee \neg y_4) \\
(\neg x_1 \vee x_3) &\mapsto (\neg x_1 \vee x_3 \vee y_5) \wedge (\neg x_1 \vee x_3 \vee \neg y_5) \\
(\neg x_2 \vee x_1) &\mapsto (\neg x_2 \vee x_1 \vee y_6) \wedge (\neg x_2 \vee x_1 \vee \neg y_6) \\
(\neg x_2 \vee x_3) &\mapsto (\neg x_2 \vee x_3 \vee y_7) \wedge (\neg x_2 \vee x_3 \vee \neg y_7)
\end{aligned}$$

where $y_1, y_2, \ldots, y_7$ are new variables.

The formula is satisfiable, for example, take x_1=`true`, x_2=`false` and x_3=`true` (the value of x_4 immaterial, i.e. can take value either `true` or `false`).

6. **Monotone SAT** is the restriction of **SAT** to expressions in which each clause contains either all negated literals or all un-negated literals. Show that **Monotone SAT** is **NP-complete**.

Hint: consider a reduction from **3-SAT** to **Monotone SAT**.

Solution: Since **SAT** belongs to **NP**, it follows immediately that **Monotone SAT** belongs to **NP**. We now give a polynomial-time reduction from **3-SAT** to **Monotone SAT**. This is sufficient since in the lectures we demonstrated that **3-SAT** was NP-complete.

Given an instance B of **3-SAT**, we construct an instance B' of **Monotone SAT** by consider each clause in turn. We have three cases to consider.

1. If the clause contains 3 negated or 3 unnegated literals, then we add the clause to B'.

2. If the clause is of the form $(x_1 \vee x_2 \vee \neg x_3)$, then we add the two clauses $(x_1 \vee x_2 \vee y)$ and $(\neg x_3 \vee \neg y)$ to B', where y is a new variable that appears nowhere else in B'. If there is a satisfying truth assignment for B, then the values of x_1, x_2 and $\neg x_3$ will make at least one of the new clauses `true`, and the value of y can be chosen to make the other clause `true`. On the other hand, if there is a satisfying truth assignment for B', then it is impossible that both of these new clauses can be `true` because of y, hence at least one of x_1, x_2 or $\neg x_3$ is `true`, and the original clause is therefore also `true`.

3. If the clause is of the form $(x_1 \vee \neg x_2 \vee \neg x_3)$, then we add the two clauses $(x_1 \vee y)$ and $(\neg x_2 \vee \neg x_3 \vee \neg y)$ to B', where again y is a new variable that appears nowhere else in B'. The argument is similar to the previous case.

Hence, there is a satisfying truth assignment for B if and only if there is a satisfying truth assignment for B'. Since the clauses in B' contain only negated or un-negated literals, the proof is complete.

7. **(hard)** The following yields a polynomial-time algorithm for 2-SAT, hence showing that it is a member of the class **P**.

   (a) Show that, for a directed graph $G = (V, E)$ and pair of vertices $v, w \in V$, finding if there is a path from v to w in G is in **P**.

   Solution: Checking the existence of a path in a directed graph can be achieved in polynomial time using, for example, depth or breadth first search.

   (b) For a 2-CNF expression, consider the directed graph G where:

   - there is a vertex for each variable and a vertex for the negation of each variable appearing in the expression;
   - there is an edge (l_1, l_2) in G if and only if the clause $(\neg l_1 \vee l_2)$, or equivalently $(l_2 \vee \neg l_1)$, appears in the expression.

   Show that if G contains a path from a literal l_1 to a literal l_2, then G also contains a path from $\neg l_2$ to $\neg l_1$.

   Recall: that $\neg(\neg x)$ equals x and a literal is either a variable or its negation.

   Solution: The result follows by induction, using in the base case the fact that by definition, if G contains an edge (l_1, l_2), then by definition it also contains the edge $(\neg l_2, \neg l_1)$.

   (c) Using this result prove that the 2-CNF expression is unsatisfiable if and only if there exists a variable x, such that there is a path from x to $\neg x$ in G and from $\neg x$ to x in G.

   Solution: We first show that, if there is a variable x, such that in the resulting graph there is a directed path from x to $\neg x$ and from $\neg x$ to x, then the expression cannot be satisfied. The proof is by contradiction.

   We use the fact that, by construction, if there is an edge (l_1, l_2) in the graph, then, in any satisfying assignment for which l_1 is `true`, l_2 must also be `true`. First, suppose, x is `true` in a satisfying assignment, then since there is a path from x to $\neg x$ it follows that $\neg x$ must also be `true` which is a contradiction. On the other hand, if $\neg x$ is `true` in a satisfying assignment, then since there is a path from $\neg x$ to x it follows that x must also be `true` which again is a contradiction.

   For the reverse direction, we need to show that if no such variable x exists then there exists a satisfying assignment. To proceed, we construct a satisfying assignment as follows:

   - find an unassigned variable x and literal l such that l equals x or $\neg x$ and there is no path from l to $\neg l$;
   - if l equals x, then that assign x to be `true`, else l equals $\neg x$ and we assign x to be `false`, repeat this assignment with all literals reachable by directed paths from l;
   - repeat until there is no unassigned variables.

For the algorithm to be well defined we need to show that there does not exist variables x and y such that there are paths from x to both y and $\neg y$. Using the second part of the question, if such variables existed than there would also be a path from $\neg y$ to $\neg x$, which would mean a path from $\neg x$ to x would exist yielding a contradiction, and hence the algorithm is well defined.

Next, we show that we do indeed construct a satisfying assignment. Suppose, for a contradiction, that the constructed assignment is not a satisfying assignement, then there exists a clause $(l_1 \vee l_2)$ where both l_1 and l_2 are `false` under the assignment. If l_1 equals `false`, then, during the construction, $\neg l_1$ must be set to be `true`, and hence l_2 must also be assigned `true` (since there is a directed edge from $\neg l_1$ to l_2). Similarly, we can show if l_2 is `false` under the assignment, then l_1 must be set to `true`. Therefore, both l_1 and l_2 cannot be assigned `false` and we have our contradiction.

(d) Finally, use these results give a polynomial time algorithm for **2-SAT**.

Solution: Combining the above, it follows that to check if a **2-SAT** expression is satisfiable it is sufficient to identify the connected components of the graph and checking whether there exists a variable x such that both x and $\neg x$ are in the same connected component. The final result follows from the fact that the size of the graph is polynomial in the size of the expression and the above graph search can be performed in polynomial time, for example, using depth first search.