CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 6
Reductions and NP-Completeness

Jeff Edmonds

1. What is a computational problem P? Give two examples.

   - Answer: A computational problem P is a function $P(I)$ from each possible finite input I that meets the stated preconditions to the corresponding required output that meets the stated post-conditions. (Sometimes there is more than one acceptable output.) A simple problem, given a set of numbers as input, is to output those numbers in sorted order. A much harder problem, given a video taken from outside of a car, is to determine whether the Google driver should turn the car in order to void a child in the road.

2. What is an algorithm A? When does it solve problem P?

   - Answer: An algorithm A is a finite sequence of instructions (perhaps a Turing Machine, but more likely written in a language like C or Java) that can mechanically be followed. Starting with an input I, it either halts with an answer denoted by $A(I)$ or runs forever denoted by $A(I) = \infty$. We say that algorithm A solves problem P if for every input I meeting the precondition, we have that A halts with the correct answer when given I, namely that $A(I) = P(I)$.

3. What is the time complexity of an algorithm A?

   - Answer: Informally, the time complexity $Time_A(n)$ of an algorithm A is a function from the amount of work needed for an adversary to present an input I to the amount needed for A to complete its computation on this input. Let $Time(A, I)$ denote the "time" until algorithm A halts on input I. This could be measured in seconds, computer cycles, or lines of code executed because these are the same within a multiplicative constant. The "size" of an instance I, denoted $n = |I|$ formally is the number of bits to describe it, but as well could be the number of ASCII characters or the area of the paper needed because these are all equivalent within a multiplicative constant. (If the input is an integer, do not, however, use its value as its size, because this is exponentially larger.) The time complexity of algorithm A is a defined to be the time needed for the worst case input of size n, namely $Time_A(n) = \max_{|I|=n} Time(A, I)$.

4. What is the time complexity of a computational problem P? What are upper and lower bounds on this?

   - Answer: The time complexity $Time_P(n)$ of a computational problem P is a defined to be the time complexity $Time_A(n)$ of the fastest algorithm A that solves P. An upper bound $Time_P(n) \leq Time_{upper}(n)$ is proved by providing an algorithm A and proving that it solves P in the stated time. A lower bound $Time_{lower}(n) \leq Time_P(n)$ is much harder to prove because one must prove that for every algorithm A, there is an input I on which it either runs too slowly or gives the wrong answer.

5. What is constant, linear, polynomial, and exponential time? Practically why does it matter? What are these times when $c = 2$ seconds and $n = 150$?

   - Answer: A running time is said to be constant, denoted $\Theta(1)$, if it is bounded between some two real positive numbers c_1 and c_2 for all sufficiently large values of n. It is said to be linear, denoted $\Theta(n)$, if bounded between $c_1 n$ and $c_2 n$; polynomial, denoted $n^{\Theta(1)}$, if between n^{c_1} and n^{c_2}; and exponential, denoted $2^{\Theta(n)}$, if between $2^{c_1 n}$ and $2^{c_2 n}$. It is important practically because if $c_2 = 2$ seconds and $n = 150$, then the times are 2 seconds for constant, 5 minutes for linear, 6 hours for polynomial, and 10^{82} years for the exponential. If the input is the size of DNA, then the algorithm really needs to be linear. Otherwise, we say that an algorithm is practical if it runs in polynomial time.

6. What is an *Optimization Problem P*?

   - Answer: A computational problem takes as input some instance I. Each such instance has a huge (likely exponential) set of *possible solutions* S. Each solution has well specified criteria for being *valid* and has a cost $cost(I, S)$. Given instance I, the goal is to find a valid solution of optimal value, i.e. maximum or minimum.

7. Give examples of optimization problems studied in class that have polynomial time algorithms. Give one for each key type of algorithm covered. Give a real world application.

   - Answer:
     (a) Given a weighted graph and nodes s and t, find a shortest path. Dijkstra's Algorithm solves this in $\Theta(E \log(E)$ time, where E is the number of edges. This is used by Google to find the shortest path from my current location to my destination.
     (b) Given a network find a min flow (or a max cut). This is solved using Primal Dual Hill Climbing (Ford Fulkerson or Edmonds-Karp Algorithms). This is used to route the shipments of a single product from the single factory to the single store along a network of highways. One can also consider the case when there are multiple goods, factories, and stores.
     (c) Linear programing requires optimizing a linear equation subject to a set of linear inequalities. Primal-Dual Simplex methods are exponential time in theory and poly-time in practice. The Elliptical method is poly-time in theory and poor in practice. It can be used to know what to put into a hotdog today in order to minimize its cost.
     (d) Minimum number of coins to make amount and Minimum Spanning Tree or Scheduling rooms are solved by Greedy Algorithms. The first is used to keep your pocket light. The second to minimize the cost of setting up power or internet wires between every home. The third is to know how to best schedule customers into your event room.
     (e) Shortest Edit Distance and Scheduling rooms with weights are solved using Dynamic Programming. The first might be used to know how closely related the DNA of two animals are. The second can again be used schedule your event room, but when the customers are willing to pay different amounts.

8. How do you make an optimization problem into a *decision problem*? Give an example. What is a *witness*?

   - Answer: For each instance, the answer is either *yes* or *no*. It is a *yes* instance if and only it has a valid solution (that is sufficiently good). For example, "Does network G have valid flow with value at least 17?" This solution is sometimes referred to as a *witness* or even as a *proof* because it witnesses/proves that the instance is a *yes* instance.

9. What does it mean for a decision problem P to be in *NP (Non-Deterministic Polynomial Time)*? (This was defined by Jeff's dad Jack Edmonds. (Do this without defining or referring to Non-Deterministic Turing Machines).

   - Answer: There is a *poly-time* algorithm $Valid(I, S)$ that given an instance I and a solution S, tests whether S is a valid solution for I. A key point is that the algorithm $Valid(I, S)$ runs in polynomial time in the size of the instance I.
     Formally we say that $P \in NP$ iff $\exists$ algorithm $Valid$ and constant c such that $\forall I,\ I \in P$ iff $\exists S\ Valid(I, S) = yes$ and $\forall I,\ S,\ Time(Valid(I, S)) \leq |I|^c$.

10. Your boss gives you an instance I of an NP problem that by good luck happens to be a *Yes* instance. You are blessed to have a powerful fairy god mother to help you. How do you convince your boss that the answer for his instance is *Yes* without him knowing or being affected by your fairy god mother? Similarly, how would you convince your boss that the answer for his instance is *No*?

- Answer: The instance is a yes instance iff it has a solution that is easy to check. She can simply give you such a valid solution S. You give this to your boss. Your boss then checks its validity with $Valid(I, S)$. Note your boss can do this because it runs in polynomial time in the size of his instance I.
  If it is a *no* instance, then (unless the problem is also in *co-NP*), it does not have a solution/witness to help you and/or your boss.

11. For an NP problem, is there a limit on the size of a solution for I and/or on the number of possible such solutions? Why?

    - Answer: Yes. The algorithm $Valid(I, S)$ must runs in polynomial time in the size of the instance I. For $Valid$ to even look at the solution S, its size must also be polynomial time in the size of the instance I, i.e. there is a constant c such that for all instances I, the size of the solution S is at most $|I|^c$ bits long. Hence the number of such solutions can be at most $2^{|I|^c}$.

12. What is the *brute force* algorithm for P and how long does it take?

    - Answer: The algorithm checks all possible solutions S. This takes at most $2^{|I|^c}$ time because this is the number of such solutions.

13. What is the famous problem $P = NP$?

    - Answer: The question is whether every problem in NP can be solved in polynomial time. The general belief is no, that many of them take exponential time.

14. What does it mean for a decision problem P to be *NP-Hard* or *NP-Complete*? What does this say about the time complexity of such a problem?

    - Answer: A problem P is NP-Hard iff it is as hard as any problem in NP, i.e. $\forall P' \in NP,\ P' \leq P$. Cook did this for SAT. Hence a problem P is NP-Hard iff $SAT \leq P$. Being NP-hard means that if there was a polynomial algorithm for P then there would be one for every problem in NP and hence $P = NP$. If, as general believed, $P \neq NP$, then such problems do not have polynomial time algorithms. A problem P is NP-Complete iff it is NP-Hard and is in NP.

15. How is this useful to you in the real world.

    - Answer: If your boss gives you a brand new problem and you are able to prove that it is NP-Complete, then you know that you should not expect to find a poly time algorithm that works for every instance. You are better off trying to find some heuristic that gives an ok answer for most instances.

16. Which problem did Cook at U. of Toronto. prove was NP-Complete? Give some other examples.

    - Answer: Cook at U. of Toronto. was the first to prove that a problem was NP-Complete. His first such problem was Circuit Satisfiability which when given a circuit as an instance wants to know if there exists a satisfying assignment.

      Many important problems that industry would love to solve are also NP-complete. Some of the classic ones are knowing whether a graph has a large clique; colouring the nodes of a graph so that each edge is bi-chromatic; and scheduling courses in a way that minimizes conflicts.

17. In practice, what is done to know whether the answer for the instance is yes or no.

    - Answer: For some problems, there are poly-time algorithms that are guaranteed to give a solutions that approximately optimal. For example, dynamic programming can find a solution for the knapsack problem whose value is within a multiplicative factor of $1+\epsilon$ of the optimal value, while the time required is only $\Theta(n^2/\epsilon)$.

Alternatively there are many heuristics for the problems that sometimes gives a good enough answers fast enough for large instances of the problem. For example, David-Putnam recursive backtracking can often solve SAT problems quickly.

18. Sketch the Venn-diagram of P, NP, $NP-complete$, $Co-NP$, $NP-Complete$, Exp.

    - Answer: $P \subseteq NP \cap Co-NP$. $NP \cup Co-NP \subseteq Exp$. $NP-Complete$ is at the top of the NP circle, above P.

19. Recall in 2001 learning that problems are *computable/decidable* if there is a TM that stops on every instance I with the correct answer. What was the definition of a problem being *acceptable/recognizable*? How is this similar to the definition of NP. How is it different? How big can the solution be? Explain how the *Halting Problem* fits this definition.

    - Answer: You likely learned that a problem is *acceptable/recognizable* iff there is a TM that stops on every *yes* instance I with the correct answer and either runs forever or says no on every *no* instance. This definition is equivalent to that for NP accept $Valid$ that does not have to run in poly time. Namely, there is a *computable* algorithm $Valid(I, S)$ that given an instance I and a solution S, tests whether S is a valid solution for I. The algorithm in the first definition simply searches for such an S and finds one iff there is one. There is no bound on the size of the solution S. An instance $\langle M, I\rangle$ to the *Halting Problem* is a *yes* instance iff it M is a TM that halts on instance I. A solution for instance $\langle M, I\rangle$ is the description of a halting computation of M on I.

20. How do you prove that one computational problem is at least as hard as another? $P_1 \leq P_2$. What is an *oracle*? Note that this is used in the definition of NP-Hard.

    - Answer: It is hard to prove that P_2 is hard. It is easier to prove that P_1 is "easier" P_2. Write an algorithm for P_1 using an algorithm for P_2 as a subroutine. We sometime refer to the algorithm for P_2 as an *oracle* (Like a burning bush on top of a mountain or that at Delphi).

21. Outline the basic code for Alg_{alg} solving P_{alg} using the supposed algorithm Alg_{oracle} supposedly solving P_{oracle} as a subroutine. If Alg_{oracle} is kind and also provides a valid solution S_{oracle} for its instance I_{oracle}, then you should provide a valid solution S_{alg} for your instance I_{alg}.

    - Answer:

      algorithm $Alg_{alg}(I_{alg})$
      $\langle$***pre−cond***$\rangle$**:** I_{alg} is an instance of P_{alg}.
      $\langle$***post−cond***$\rangle$**:** Determine whether I_{alg} has a solution S_{alg} and if so returns it.

      begin
      $I_{oracle} = InstanceMap(I_{alg})$
      $\langle ans_{oracle}, S_{oracle}\rangle = Alg_{oracle}(I_{oracle})$
      if($ans_{oracle} = Yes$) then
      $ans_{alg} = Yes$
      $S_{alg} = SolutionMap(S_{oracle})$
      else
      $ans_{alg} = No$
      $S_{alg} = nil$
      end if
      return($\langle ans_{alg}, S_{alg}\rangle$)
      end algorithm

22. What steps do you have to take to prove that this reduction is correct?

    - Answer: We have to prove that Alg_{alg} works if Alg_{oracle} works. For this we prove

(a) Given an instance I_{alg}, $InstanceMap(I_{alg})$ maps this to a valid instance I_{oracle}.

(b) If S_{oracle} is a solution for I_{oracle} than $S_{alg} = SolutionMap(S_{oracle})$ is a solution for I_{alg} whose cost is just as good.

(c) If S_{alg} is a solution for I_{alg} than $S_{oracle} = ReverseSolutionMap(S_{alg})$ is a solution for I_{oracle} whose cost is just as good.

23. Give two purposes of reductions.

- Answer:
  (a) Designing new algorithms
  (b) Arguing that a problem is hard or easy.
  (c) Identifying equivalence classes of problems.

24. Name two reductions done this term (using these two purposes).

- Answer: We got an algorithm for the Boy & Girls Marriage given an algorithm for Network Flows. We got an algorithm for Circuit Satisfiability using one for 3-Colouring. But there in likely not one for Circuit Satisfiability and hence not likely one for Colouring.

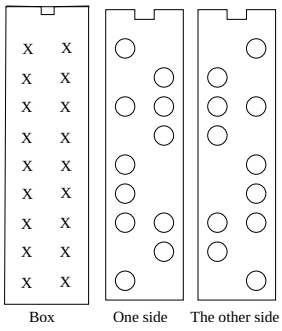
25. NP-Completeness:

(a) The problem Clause-SAT is given a set of clauses where each clause is the OR of a set of literals and each literal is either a variable or its negation. The goal is know whether you can set each variable to true or false so that each clause is satisfied.

The course notes (along with an exercise on 3-SAT), prove that the problems (circuit) SAT, 3-Colouring, Course Scheduling, Independent Set, and 3-SAT are NP-Complete. In a few sentences, explain how you know that Clause-SAT is NP-Complete.

- Answer: Clause-SAT is more general that 3-SAT. Hence, if you can solve any instance of Clause-SAT then you can definitely solve any of 3-SAT. Also being in NP, it follows that Clause-SAT is NP-Complete.

(b) The problem *Card* is defined by the following puzzle.
You are given a box and a collection of cards as indicated in the figure. The box and each card has r rows and two columns of positions in the same places. The box contains Xs in these positions. In each card at each of these positions, there is either a hole punched out or there is not. Each card must be placed in the box either face up or flipped over left to right. It has a notch at its top which must be at the top of the box. Consider some row in some card. Suppose its left position has a hole but its right position does not. Putting the card in face up covers the right X in the box of this row but not the left. Putting the card in flipped over covers the left but not the right. The goal is to cover each and every X in the box. A solution specifies for each card whether to flip the card or not. A solution is valid if it covers all the Xs.



Prove that *Card* is NP-Complete by reducing it to Clause-SAT. Be sure to think about each of the 12 steps. As a huge hint, I do step 5 for you. You are to write up steps 0, 6, and 7. For each of these have a clear paragraph with pictures.

0) $P_{card} \in$ NP:

5) InstanceMap: Given an instance $I_{Clause-SAT}$ to Clause-SAT consisting of a set of clauses, we construct as follows an instance $I_{card} = InstanceMap(I_{Clause-SAT})$ for the card game consisting of a set of cards. Our box and cards will have one row for each clause. We will have a card for each variable x. For each clause c, this card for x will have its left hole position for row c not punched out if x appears positively in c and will have its right hole position for

c not punched out if x appears negatively in c. It will have all other hole positions punched out. We will have one additional card that has all of its left hole positions punched out and none of its right hole positions punched out.

6) SolutionMap:

7) Valid to Valid:

- Answer:

  0) $\boldsymbol{P_{card} \in}$ **NP:** Given an instance consisting of a set of cards and a solution consisting of an orientation of each card, it is easy to determine if every hole position is blocked.

  6) SolutionMap: Given a solution S_{card} for the instance $I_{card} = InstanceMap(I_{Clause-SAT})$ consisting of a notflipped/flipped orientation of each card, we construct as follows a solution $S_{Clause-SAT}$ for instance $I_{Clause-SAT}$ consisting of a true/false setting of each variable. First, if in the solution S_{card}, the additional card is flipped then we can flip the entire deck of cards over without changing whether all the hole positions are covered. Hence, without loss of generality assume that in S_{card}, the additional card is not flipped. Then for each variable x, set it to true iff its associated card is notflipped in S_{card}.

  7) Valid to Valid: Assume that S_{card} is a valid solution, i.e. every hole position is covered. Our goal is to prove that $S_{Clause-SAT}$ is a valid solution, i.e. every clause is satisfied. Consider some clause c. Because the additional card is not flipped in S_{card} and every hole position is covered, we know that there is some variable card x covering the left hole position for clause c. If this card is not flipped then its left hole position for c must not be punched. In this case, x is set to true in $S_{Clause-SAT}$ and x appears positively in c. Hence, this clause is satisfied. Alternatively, if this card is flipped then its right hole position for c must not be punched. In this case, x is set to false in $S_{Clause-SAT}$ and x appears negatively in c. Again, this clause is satisfied. Hence, each clause in $S_{Clause-SAT}$ is satisfied.

(c) Suppose you work for an airplane manufacturing company. Given any detailed specification of a plane, right down to every curve and nut and bolt, your boss has a way of determining whether it is a great plane or not. However, he has tried thousands of different specifications and none of them turn out to be great. He is completely frustrated.

Meanwhile back at home, your son didn't finish high school, is socially awkward, and is living in your basement at 31. You blame it on the fact that he is completely addicted to playing video games. You feel this is only one step better to being a crack addict. One game he plays all the time is the card flipping puzzle described above. He has a magical ability to instantly put any set of cards into the box so that all the Xs are covered.

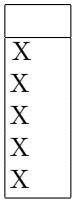
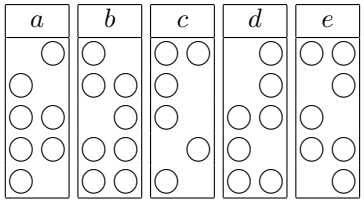
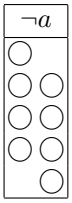
You are are only able to do things described in the course notes (or hinted at in exercises). How do you get your son a job at work so that he quickly becomes a billionaire?

- Answer: Following your boss' method, you write a JAVA program that takes as input the description of plane and outputs whether or not it is great. (Given a fixed number of bits n to describe the input, it is not too hard to automatically compile this program into an AND-OR-NOT circuit that does the same thing. The notes describe a way to convert this circuit into a graph to be 3-Colored. The exercise in the notes hints at how to convert this into an instance of 3-SAT, which is itself an instance of Clause-SAT. Above we describe how to convert this into an instance of the Card game. You get your son to magically solve it. You convert this solution into a solution for the 3-Sat instance, which you convert into a solution of the graph coloring instance, which you covert into a solution of the circuit problem and a valid input of the JAVA program. This you translate into a description of a great plane, which you give your boss. You and your son go on to solving all of the world's optimization problems.

26. Consider the following two computational problems and following correspondence between them.

	Card Puzzle	Clause-SAT
Input:	A box and cards n cards: Each labeled with var. and its negation on back Each card has m rows - card v, j^{th} row, & 1^{st} col: non-hole - card v, j^{th} row, & 2^{nd} col: non-hole	The *and* of clauses: Each the *or* of vars n variables m clauses - var v appears positively in j^{th} clauses - var v appears negatively in j^{th} clauses
Solution:	Each card flipped or not and put in box	Each variable negated or not
Satisfied:	Each X has a nonhole of card covering it - eg: Flipped card a covers last X	Each clause has a variable satisfying it - eg: Negated a satisfies last clause
Determine:	Ans Yes iff $\exists$ a satisfying solution	Ans Yes iff $\exists$ a satisfying solution
See more:	4111/ass/06-ass-NP_Complete.pdf Q3	2001-60-NP_Complete.pptx Quick Reduce

Example:

Box	Cards	Flipped	Clauses
			(a or $\neg b$ or d) and ($\neg a$ or $\neg c$ or d or e) and (b or $\neg c$ or $\neg e$) and (c or $\neg d$) and ($\neg a$ or $\neg c$ or e)

(a) I spelled out a correspondence between
- an instance of the card puzzle and
- an instance of clause-SAT.

Spell out a similar correspondence between
- a solution of the card puzzle and
- a solution of clause-SAT.

(1 sentences)

- Answer:
  A card in a solution of the card puzzle is flipped
  iff the corresponding variable in the corresponding solution of clause-SAT is negated.

(b) Is there a correspondence between
- an instance of the card puzzle requiring a Yes and
- the corresponding instance of clause-SAT requiring a Yes?

Explain.

(5 sentences)

- Answer:
  For each j and v,
  - card v covers the j^{th} X
  - iff variable v satisfies the j^{th} clause.

  Hence, for each j,
  - the j^{th} X is covered
  - iff the j^{th} clause is satisfied.

  Hence,
  - the solution of the card puzzle satisfies the requirement
  - iff the corresponding solution of clause-SAT does.

  Hence,
  - $\exists$ a satisfying solution for the instance of card puzzle
  - iff $\exists$ a satisfying solution for corresponding instance of clause-SAT.

Hence,
- the instance of card puzzle requires a Yes
- iff the corresponding instance of clause-SAT requires a Yes.

(c) A problem P is in *NP (Non-Deterministic Polynomial Time)* iff there is a *poly-time* algorithm $Valid(I, S)$ that given an instance I and a solution S, tests whether S is a valid solution for I. A key point is that the algorithm $Valid(I, S)$ runs in polynomial time in the size of the instance I. Formally we say that $P \in NP$ iff $\exists$ algorithm $Valid$ and constant c such that $\forall I,\ I \in P$ iff $\exists S\ Valid(I, S) = yes$ and $\forall I,\ S,\ Time(Valid(I, S)) \leq |I|^c$.
Prove that the Card Puzzle is in NP.

(1 sentences)

- Answer: Given cards I and whether they are flipped S, it is easy to tell if all the X in the box are covered.

(d) A problem P is NP-Hard iff $Clause\text{-}SAT \leq P$.
Prove that the Card Puzzle is NP-Hard by giving a few sentences for steps 4 and 5 of the "A Quick Reduction" slides.

(4 sentences)

- Answer: Step 4: Given an oracle for the Card Puzzle, we give an algorithm for Card-SAT. Given a set of clauses construct the corresponding cards as described above, give these to the oracle, and answer the same.
  Step 5: Clauses satisfiable iff Card Puzzle satisfiable iff Oracle says yes iff Alg says yes. See question (b).