Homework 6

1- Define INTEGER PROGRAMMING (IP) problem as follows:
Given m equations:
∑ j=1,n aij xj = bi , i=1, ..., m

in n variables xj with integer coefficients aij and bj , are there solutions with xj equal to 1 or 0 for each j ? Prove that IP is an NP-complete problem.

Solution:

To prove that IP is an NP-complete problem we need to prove two things:

IP is in NP:

A problem is in NP if a candidate solution can be verified in polynomial time. For IP, given a solution (a set of values for the xj), we can substitute those values into the equations and check whether all equations hold. This process clearly takes polynomial time (since there are m equations, each with n terms), so IP is in NP.

P1 is reducable to IP where P1 is any problem in NP:

We can use 3-SAT for this. We can transform a given instance of 3-SAT to IP as follows: For each variable xi in the 3-SAT instance, create a corresponding variable xi in IP. For each clause (xi OR xj OR xk) in the 3-SAT instance, create a corresponding equation xi + xj + xk >= 1 in IP. This equation ensures that at least one of xi, xj, xk is true (i.e., equals 1), which is exactly the requirement for the clause in 3-SAT. The transformation can be done in polynomial time, since it involves creating one IP variable for each 3-SAT variable and one IP equation for each 3-SAT clause.

Hence, IP is NP-Complete.

2– Define 3-COLORING (3C) problem as follows: Given an undirected graph can its vertices colored with three colors such that no two adjacent nodes have the same color. Prove that 3C is an NP-complete problem (Hint: Use a polynomial reduction from 3SAT)

Solution:

To prove that 3C is an NP-complete problem we need to prove two things:

3C is in NP:

A problem is in NP if a candidate solution can be verified in polynomial time. For any proposed 3-coloring of a graph, we can easily verify in polynomial time whether the coloring is valid (i.e., no two adjacent vertices share the same color). Thus, the 3C problem is in NP.

P1 is reducable to 3C where P1 is any problem in NP:

We can use 3-SAT for this. Given an instance of the 3-SAT problem with n variables and m clauses, we can construct a graph G such that G is 3-colorable if and only if the 3-SAT instance is satisfiable.

For each variable xi in the 3-SAT instance, we create a corresponding "variable gadget" in the graph. This gadget consists of two vertices connected by an edge, representing xi and ¬xi. We also add a "triangle" (a cycle of length 3) that includes these two vertices and an extra vertex, ensuring that xi and ¬xi must have different colors.

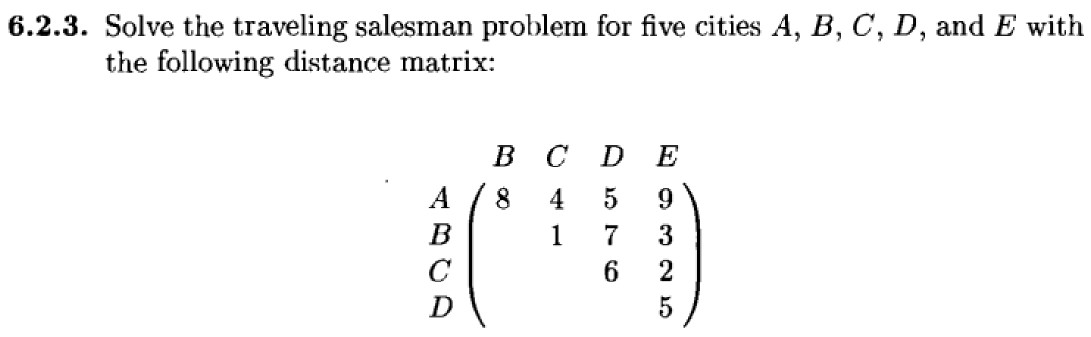
For each clause (xi OR xj OR xk) in the 3-SAT instance, we create a corresponding "clause gadget" in the graph. This gadget consists of a triangle with one vertex for each literal in the clause. We connect each literal in the clause gadget to its corresponding vertex in the variable gadget.

Now, any 3-coloring of this graph corresponds to a satisfying assignment of the 3-SAT instance, and vice versa.

This reduction can be done in polynomial time.

Therefore, the 3C problem is NP-Complete.

3-



6.2.3. Solve the traveling salesman problem for five cities A, B, C, D, and E with the following distance matrix:

$$\begin{array}{c|cccc} & B & C & D & E \\ \hline A & 8 & 4 & 5 & 9 \\ B & & 1 & 7 & 3 \\ C & & & 6 & 2 \\ D & & & & 5 \end{array}$$

Solution:

There is (5-1)! = 4! = 24 different routes.

A-B-C-D-E: 8+1+6+5=20

A-B-C-E-D: 8+1+2+5=16

A-B-D-C-E: 8+7+6+2=23

A-B-D-E-C: 8+7+5+2=22

A-B-E-C-D: 8+3+2+6=19

A-B-E-D-C: 8+3+5+6=22

A-C-B-D-E: 4+1+7+5=17

A-C-B-E-D: 4+1+3+5=13

A-C-D-B-E: 4+6+7+3=20

A-C-D-E-B: 4+6+5+3=18

A-C-E-B-D: 4+2+3+7=16

A-C-E-D-B: 4+2+5+5=16

A-D-B-C-E: 5+7+1+2=15

A-D-B-E-C: 5+7+3+2=17

A-D-C-B-E: 5+6+1+3=15

A-D-C-E-B: 5+6+2+3=16

A-D-E-B-C: 5+5+3+1=14

A-D-E-C-B: 5+5+2+1=13

A-E-B-C-D: 9+3+1+6=19

A-E-B-D-C: 9+3+7+6=25

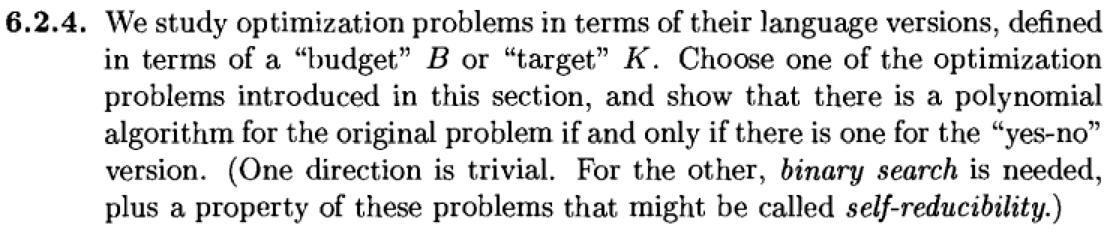
A-E-C-B-D: 9+2+1+7=19

A-E-C-D-B: 9+2+6+5=22

A-E-D-B-C: 9+5+7+1=22

A-E-D-C-B: 9+5+6+1=21

The shortest path is Route 8: A-C-B-E-D with a total distance of 13.



6.2.4. We study optimization problems in terms of their language versions, defined in terms of a "budget" B or "target" K. Choose one of the optimization problems introduced in this section, and show that there is a polynomial algorithm for the original problem if and only if there is one for the "yes-no" version. (One direction is trivial. For the other, ***binary search*** is needed, plus a property of these problems that might be called ***self-reducibility***.)

Solution:

We can choose the Knapsack problem to show there is a polynomial algorithm for the problem if and only if there is one for the "yes-no" version.

Knapsack problem is defined as: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

=> trivial direction: If there is a polynomial time algorithm for the original problem, then there is one for the "yes-no" version.

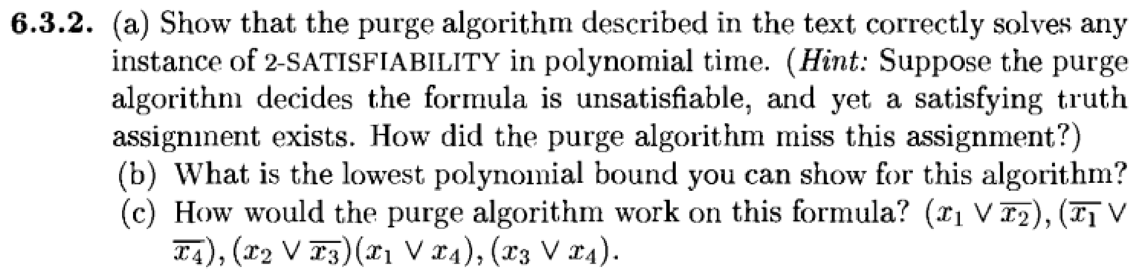
Given an algorithm for the optimization version, we can solve the "yes-no" version by running the optimization algorithm and checking if the maximum value found is at least K. This will take polynomial time if the optimization algorithm is polynomial.

<= If there is a polynomial time algorithm for the "yes-no" version, then there is one polynomial time algorithm for the original problem.

The concept of "self-reducibility" applies here. If there is a polynomial-time algorithm for the "yes-no" version, we can use it to find an optimal solution to the original problem via binary search.

The maximum possible value in the knapsack is the sum of all values, say V. Then we perform binary search on the range [0, V]. For a current candidate solution K (the midpoint of the current range), we check using the "yes-no" version whether a knapsack of total value at least K exists. If it does, we continue searching in the upper half of the range; otherwise, we search the lower half.

This binary search procedure requires logarithmically many calls to the "yes-no" algorithm, so if the "yes-no" algorithm runs in polynomial time, the entire procedure will still be polynomial in the size of the problem.



6.3.2. (a) Show that the purge algorithm described in the text correctly solves any instance of 2-SATISFIABILITY in polynomial time. (*Hint:* Suppose the purge algorithm decides the formula is unsatisfiable, and yet a satisfying truth assignment exists. How did the purge algorithm miss this assignment?)
(b) What is the lowest polynomial bound you can show for this algorithm?
(c) How would the purge algorithm work on this formula? $(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_4}), (x_2 \vee \overline{x_3})(x_1 \vee x_4), (x_3 \vee x_4)$.

Solution:

a) To show the correctness of the purge algorithm, we will make use of a contradiction. Suppose the purge algorithm decides the formula is unsatisfiable, but a satisfying truth assignment does exist.

If the purge algorithm decides the formula is unsatisfiable, it must have encountered a situation where a variable and its negation both appear as single literals in the remaining clauses, which is impossible to satisfy.

However, we supposed that a satisfying truth assignment exists, so this assignment must assign a truth value to the problematic variable such that neither the variable nor its negation would appear as a single literal. Thus, the purge algorithm could not have encountered the unsatisfiable situation. This contradicts the assumption that the purge algorithm decides the formula is unsatisfiable, which proves that if a satisfying truth assignment exists, the purge algorithm will find it.

Regarding the polynomial-time complexity, each purge operation examines every clause and makes modifications based on the literals contained. This can be done in linear time relative to the number of clauses. Furthermore, we only perform a purge operation twice for each variable in the worst case. Hence, the overall time complexity is O(nm), where n is the number of variables and m is the number of clauses, which is polynomial.

b) The purge operation examines all clauses and makes modifications accordingly. This operation is linear with respect to the number of clauses. Also, in the worst-case scenario, we perform a purge operation twice for each variable. Therefore, the overall time complexity is O(nm), where n is the number of variables and m is the number of clauses, which is polynomial.

c) Given the formula: (Xl V X2^), (Xl V X4^), (X2 V X3^)(Xl V X4), (X3 V X4).

Let's begin by arbitrarily setting X1 to true (T). The formula simplifies as follows: X2^, X4^, X3^ V X4

Here, X2^, X4^ appear as unit clauses, so we set X2 = F and X4 = F, and we obtain:

X3^ Then, we set X3 = F, making the formula empty. Therefore, the algorithm finds a satisfying assignment: X1=T, X2=F, X3=F, X4=F.