4005-800 Algorithms

Homework 7

Christopher Wood

May 13, 2012

PROBLEM 1 - 34.2-1. *Consider the language GRAPH-ISOMORPHISM* $= \{\langle G_1, G_2\rangle : G_1$ *and* G_2 *are isomorphic graphs* $\}$. *Prove that GRAPH-ISOMORPHISM* $\in NP$ *by describing a polynomial-time algorithm to verify the language.*

Solution. By the defintion of graph isomorphism, two graphs G_1 and G_2 are isomorphic if and only if there exists a bijection $m : V(G_1) \rightarrow V(G_2)$ such that any two vertices v_i and v_j of G_1 are adjacent in G_1 if and only if $m(v_i)$ and $m(v_j)$ are adjacent in G_2. With this definition, it is enough to check the bijection m to see if it fulfills this property to verify that two graphs isomorphic. We can easily devise a polynomial-time algorithm to solve this as follows:

Algorithm 1: GRAPH-ISOMORPHISM-VERIFIER

```
function VerifyGraphIsomorphism(m)
    for all v_i ∈ V(G_1) do
        vCount = 0                              ▷ Count number of times v_i appears in G_2
        for all v_j ∈ V(G_2) do
            if m(v_i) == v_j then
                vCount = vCount + 1
            end if
        end for
        if vCount ≠ 1 then                      ▷ v_i should only appear once in G_2
            return False
        end if
    end for

    for all v_i ∈ V(G_1) do
        for all v_j ∈ V(G_1) do
            if (v_i, v_j) ∈ E(G_1) and (m(v_i), m(v_j)) ∉ E(G_2) then
                return False
            end if
        end for
    end for

    for all v_i ∈ V(G_1) do
```

```
            for all v_j ∈ V(G_1) do
                if (m(v_i), m(v_j)) ∈ E(G_2) and (v_i, v_j) ∉ E(G_1) then
                    return False
                end if
            end for
        end for

        return True
    end function
```

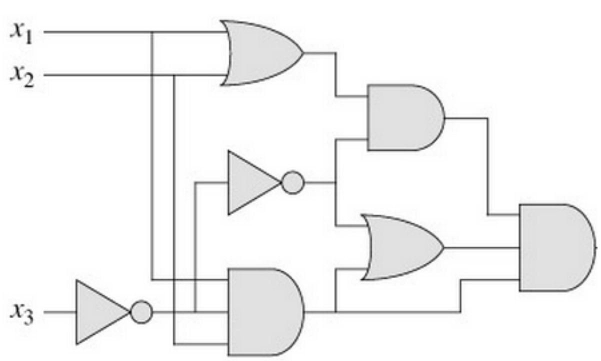
Note that m denotes the bijective mapping (the permutation) from $V(G_1)$ to $V(G_2)$. It is clear that the permutation check runs in $O(V)$ time and the edge check runs in $O(V^2)$ time. Thus, we conclude that this algorithms runs in $O(V^2)$ time and thus verifies the solution (i.e. the permutation mapping m) to the GRAPH-ISOMORPHISM problem in polynomial time.

PROBLEM 2 - 34.2-10. *Prove that if $NP \neq co\text{-}NP$, then $P \neq NP$.*

Solution. If $NP \neq co\text{-}NP$, then we know there exists a problem $Q \in NP$ such that $Q \notin co\text{-}NP$. Furthermore, by definition, we know that $P \in co\text{-}NP \cap NP$. Now, let Q be a problem in NP that is not in $co\text{-}NP$. By the definition of the set intersection, that means that $Q \notin co\text{-}NP \cap NP$, and thus we know that $Q \in NP$ and $Q \notin P$ (because P is a subset of $co\text{-}NP \cap NP$). Therefore, since there exists a problem that is in NP but not in P, we conclude that $P \neq NP$.

PROBLEM 3 - 34.3-1. *Verify that the circuit in Figure 34.8(b) is unsatisfiable.*

Solution.



The logical equivalent expression for this circuit is as follows:

$$((x_1 \vee x_2) \wedge x_3) \wedge (x_3 \vee (x_1 \wedge x_2 \wedge \neg x_3)) \wedge (x_1 \wedge x_2 \wedge \neg x_3)$$

x_1, x_2, x_3	$x_1 \vee x_2$	$(x_1 \vee x_2) \wedge x_3$	$x_1 \wedge x_2 \wedge \neg x_3$	$x_3 \vee (x_1 \wedge x_2 \wedge \neg x_3)$	Final AND Gate
F,F,F	F	F	F	F	F
F,F,T	F	F	F	T	F
F,T,F	T	F	F	F	F
F,T,T	T	T	F	T	F
T,F,F	T	F	F	F	F
T,F,T	T	T	F	T	F
T,T,F	T	F	T	T	F
T,T,T	T	T	F	T	F

Table 1: Truth table for problem #3, where T = True and F = False.

To show that this circuit is unsatisfiable, we simply build a truth table for the boolean expression that considers all logical values for x_1, x_2, and x_3, as shown in Table :

Therefore, since there is no possible combination of logical values for x_1, x_2, and x_3 such that the boolean expression is true, we conclude that it is unsatisfiable.

PROBLEM 4 - 34.4-5. *Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.*

Solution. We show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable by providing a polynomial-time algorithm that performs this task. This algorithm is realized below in Algorithm 3.

ALGORITHM 2: DNF-SOLVER

function SOLVEDNF(ψ)
    for all Logical clauses $c_i \in \psi$ **do**
        $satisfiable = True$
        $satList = makeQueue()$
        for all Literals $l_j \in c_i$ **do**
            for all Literals $l_k \in satList$ **do**
                if $l_j == \neg l_j$ **then**
                    $satisfiable =$ False ▷ Found the negation of l_j in the queue
                end if
            end for
            $PUSH(satList, l_j)$ ▷ Push this literal into the queue
        end for
        if $satisfiable == True$ **then**
            return $True$ ▷ Found one clause that can be satisfied
        end if

end for
return $False$
end function

Since DNF statements are composed of disjunctions (ORs) of conjunction clauses (ANDs), it is enough to check and see if only one conjunction clause can be satisfied. Therefore, this procedure simply traverses over every clause and checks to see if there is a literal and its negation in that clause, which indicates that the clause can never be true. If this is not the case, then the clause must be satisfiable, and thus the expression is satisfiable.

The time complexity of this algorithm is $O(mn^2)$, where m is the number of clauses in ψ and n is the number of literals in the boolean expression. The reason for this is that for every clause we traverse over every literal in the clause, and for each element we perform a linear search with $satList$ that can be equal to the number of literals in the clause. Therefore, since the linear search runs in $O(n)$ time, the number of literals in a clause is $O(n)$, and the number of clauses in ψ is $O(m)$, and each of these operations are nested, the resulting time complexity is $O(mn^2)$.

PROBLEM 5 - 34.5-5. *The* ***set-partition problem*** *takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $A' = S - A$ such that $\sum_{x \in A} x = \sum_{x \in A'} x$. Show that the set-partition problem is NP-complete.*

Solution. In order to show that the set-partition problem Q is NP-complete, we show that it can be reduced from the subset-sum problem Q'. That is, we prove $Q' \leq_p Q$ as follows:

- First, we must show that $Q \in NP$. To do this, we show that an instance of Q can be verified in polynomial time. Suppse for an instance of Q we have two partite subsets A and A' of an input set S of numbers. We can easily compute the sum of all elements in both A and A' in $O(n)$ time and then compare if these sums are equivalent in constant time. Thus, the instance of Q can be verified in polynomial time ($O(n)$), and so we conclude that $Q \in NP$.

- Now we need to show that $Q' \leq_p Q$. Consider an instance of Q' that consists of a set of numbers S and an integer target value t, where some the sum of all elements in a subset $S' \subseteq S$ adds to t. If we denote $s = \sum_{x \in S} x$, then we also know that $\sum_{x \in S - S'} x = s - t$. Now, we can create a new augmented set $S^* = S \cup \{s - 2t\}$, which results in $\sum_{x \in S^*} x = (s + (s - 2t)) = 2s - 2t = 2(s - t)$. Therefore, we know that there are exactly two partite sets in S that sum to exactly $s - t$, and thus S^* is an instance of Q' that also satisfies Q.

  Now, consider an instance of Q that consists of a set of numbers S^* and two partite sets A and A', where $\sum_{x \in A} x = \sum_{x \in A'} x = (s - t)$. By our construction technique, we know that one of these partite sets contains the number $m = s - 2t$, and if we remove that number from the corresponding set we are left with $s - t - (s - 2t) = t$. Therefore, since $S^* - \{m\} = S$ and there is a subset in S that now sums to t, we conclude that this instance of Q also satisfies Q'.

- Finally, we show that the construction of S^* can be done in polynomial time from an instance of Q' by computing the sum $s - 2t$ and including it in the set S.

Thus, since we defined a polynomial time construction f that builds instances of Q from Q', and such instances satisfy both Q and Q', we conclude that $Q' \leq_p Q$. Now, since Q' is NP-complete, we can also conclude that Q is NP-complete.

PROBLEM 6-a. *Write pseudo-code for a recursive solution to the variation on the 0-1 knapsack problem that computes the maximum value that can be placed in the knapsack.*

Solution.

ALGORITHM 3: RECURSIVE 0-1 KNAPSACK

function RECURSIVEKNAPSACK(n, v, w, W)
    if $n == 0$ **then** ▷ There are no items to contribute weight or value
        return 0
    else if $W - w[n] < 0$ **then** ▷ This item is too heavy, so don't include it
        return $RecursiveKnapsack(n-1, v, w, W)$
    else
        return $\max(v[n] + RecursiveKnapsack(n-1, v, w, W - w[n]),$
$RecursiveKnapsack(n-1, v, w, W))$
    end if
end function

PROBLEM 6-b. *Give a dynamic programming solution to the 0-1 knapsack problem that is based on the previous problem; this algorithm should return the items to be taken. Implement this algorithm and call it knapsack.*

Solution. TODO: insert source code once finished.

PROBLEM 6-c. *What is the time complexity of your dynamic programming based algorithm?*

Solution. The time complexity of this dynamic programming based algorithm depends on the computation of the *value* table and identifying the items that were added to the knapsack. Since these procedures are run back-to-back, we consider their time complexity separately in order to determine the time complexity of the entire algorithm.

The time complexity of the value computation depends on the initialization procedure in which the table is constructed and then the nested loops that perform the bottom-up computation. The initialization procedure generates a table that has dimensions $n \times W$, so it runs in $O(nW)$ time. Similarly, the table computation procedure performs a constant time table lookup (or returns a 0 in the base case) when traversing across every possible knapsack capacity for every item, so we can conclude that this procedure runs in $O(nW)$ time as well.

Analyzing the time complexity of the item identification procedure indicates that it runs in $O(n)$ time, because at every iteration through the main loop the item counter is decreased by 1 until we consider all items in the knapsack. Hence, the linear time complexity of $O(n)$.

Now, putting these two results together, the dynamic programming based algorithm that solves the 0-1 knapsack problem has a time complexity of $O(nW) + O(nW) + O(n)$, which can be reduced to $O(nW)$.

PROBLEM 6-d. *The knapsack decision problem is NP-complete. Does your analysis above prove that $P = NP$? Explain.*

Solution. No, this analysis does not prove that $P = NP$, because the $O(nW)$ time complexity comes from the number of bits that are used to represent W, not the value of W. As the weight capacity of the knapsack increases the number of individual items that can be stored in W increases by individual items as well, and if we treat W as a binary number then we also say that the number of bits used to store W increases by 1. However, by increasing the number of bits in W in a linear manner, we increase the value of W exponentially. Thus, we can conclude that this dynamic programming problem runs in psuedo-polynomial time, which means that it ultimately runs in exponential time complexity. Therefore, this solution is still in NP, so we cannot conclude that $P = NP$.