# 4005-800 Algorithms

## Homework 7

Christopher Wood

May 15, 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 $G_2$. In other words, the mapping $m$ is a defines a permutation from $V(G_1) \rightarrow V(G_2)$, and we can check to see if this is a valid permutation by checking that for every $v_j \in V(G_2)$ there is exactly one $v_i \in V(G_1)$ such that $m(v_i) = v_j$, and for every $v_i \in V(G_1)$ there is exactly one $v_j \in V(G_2)$ such that $m(v_i) = v_j$. We can then check to see that for every edge $(v_i, v_j) \in E(G_1)$, we also have $(m(v_i), m(v_j)) \in E(G_2)$. With these definitions in mind, we can easily devise a polynomial-time algorithm to verify that $m$ is a valid isomorphic mapping as follows:

---

Algorithm 1: GRAPH-ISOMORPHISM-VERIFIER

```
1:  function VerifyGraphIsomorphism(m)
2:      for all v_j ∈ V(G_2) do
3:          vCount = 0                          ▷ Count number of times m(v_i) appears in G_2
4:          for all v_i ∈ V(G_1) do
5:              if m(v_i) == v_j then
6:                  vCount = vCount + 1
7:              end if
8:          end for
9:          if vCount ≠ 1 then                  ▷ v_i should only appear once in G_2
10:             return False
11:         end if
12:     end for
13:
14:     for all v_i ∈ V(G_1) do
15:         vCount = 0                          ▷ Count number of times m(v_i) appears in G_2
16:         for all v_j ∈ V(G_2) do
17:             if m(v_i) == v_j then
18:                 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      ▷ Check the edge adjacency requirement for the permutation
            if (v_i, v_j) ∈ E(G_1) and (m(v_i), m(v_j)) ∉ E(G_2) then
                return False
            end if
            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**.

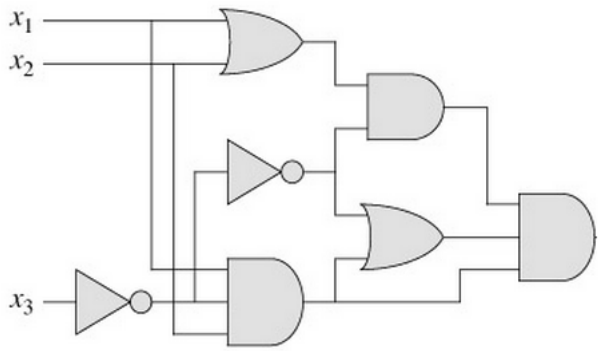
Assume, for the sake of contradiction, that if $NP \neq co\text{-}NP$, then $P = NP$. Since $P$ is closed under complementation, we know that $P = co\text{-}P$. Therefore, since $P = NP$, we also know that $NP$ is closed under complementation, and thus $NP = co\text{-}NP$. However, this contradicts our assumption that $NP \neq co\text{-}NP$, so we conclude that if $NP \neq co\text{-}NP$ then $P \neq NP$.

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

**Solution**.

| $x_1, x_2, x_3$ | $x_1 \vee x_2$ | $(x_1 \vee x_2) \wedge x_3$ | $x_1 \wedge x_2 \wedge \neg x_3$ | $x_3 \vee (x_1 \wedge x_2 \wedge \neg x_3)$ | 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.




The logical equivalent expression for this circuit is as follows:

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

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

1: **function** SOLVEDNF($\psi$)
2: **for all** Logical clauses $c_i \in \psi$ **do**
3: $satisfiable = True$

```
4:        satList = makeQueue()
5:        for all Literals l_j ∈ c_i do
6:            for all Literals l_k ∈ satList do
7:                if l_k == ¬l_j then
8:                    satisfiable = False          ▷ Found the negation of l_j in the queue
9:                end if
10:           end for
11:           PUSH(satList, l_j)                   ▷ Push this literal into the queue
12:       end for
13:       if satisfiable == True then
14:           return True                          ▷ Found one clause that can be satisfied
15:       end if
16:   end for
17:   return False
18: 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 $\psi$ and $n$ is the number of variables in the boolean expression. The reason for this is that for every clause we traverse over every literal in the clause, and for each literal 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 $\psi$ 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 $Q'$, which is also $NP$-complete. That is, we prove $Q' \leq_p Q$ as follows:

- First, we must show that $Q \in NP$. To do this, we show that a certicate of $Q$ can be verified in polynomial time. Suppse such a certificate of $Q$ contains 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 certficate of $Q$ can be verified in polynomial time ($O(n)$), and so we conclude that $Q \in NP$.

- Now we need to show that $Q' \leq_p Q$. To do so, we define a construction technique that translates instances of $Q'$ to instances of $Q$ as follows. Let $S$ be a set of numbers that contains a subset $S' \subseteq S$ that sum to the integer target $t$. We denote $S_{sum} = \sum_{x \in S} x$, and with this we construct a new set $S^*$ as $S \cup \{S_{sum} - 2t\}$. The reasoning behind this construction is that the set $S'' = S - S'$ has a sum of $S_{sum} - t$, and so if we add the number $S_{sum} - 2t$ to $S$ we are guaranteed that by including this number in $S'$ we are left with another total sum of $t + S_{sum} - 2t = S_{sum} - t$. Thus, there will be two partite sets that sum to exactly $S_{sum} - t$.

- Now, suppose we have a satisfiable instance of $Q'$ that consists of a set of numbers $S$ and a subset of $S' \subseteq S$ that sums to an integer target $t$. Applying the construction technique described above, we are left with a new set $S^*$, where we have the following:

$$\begin{aligned}
\sum_{x \in S^*} x &= (S_{sum} + (S_{sum} - 2t)) \\
&= (t + (S_{sum} - t) + (S_{sum} - 2t) \text{ (Split up the subsets in } S \text{ that sum to } S_{sum} \text{ and } S_{sum} - t \\
&= (S_{sum} - t) + (S_{sum} - 2t + t) \text{ (Include } S_{sum} - 2t \text{ in } S') \\
&= 2(S_{sum} - t)
\end{aligned}$$

  Therefore, we know that there are exactly two partite sets in $S$ that sum to exactly $S_{sum} - t$, and thus $S$ is an instance of $Q'$ that can be used to construct $S^*$, which also satisfies $Q$.

- Now, suppose we have a satisfiable instance of $Q$ that contains two partitite sets $A$ and $A'$, where $\sum_{x \in A} x = \sum_{x \in A'} x = (S_{sum} - t)$. By our construction technique, we know that one of these partite sets contains the number $m = S_{sum} - 2t$, and if we remove that number from the corresponding set we are left with $S_{sum} - t - (S_{sum} - 2t) = t$. 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 $Q'$ simply by computing the sum $S_{sum} - 2t$, which is done in $O(n)$ time by traversing over all of the elements in $S$, and then including it in the set $S$ in constant time.

Thus, since we defined a polynomial time construction that transforms satisfiable instances of $Q'$ to $Q$, 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 $Q$ (the set-partition problem) 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

1: **function** RECURSIVEKNAPSACK($n, v, w, W$)
2: **if** $n == 0$ **then** ▷ There are no items to contribute weight or value
3: **return** 0
4: **else if** $W < w[n]$ **then** ▷ This item is too heavy, so don't include it
5: **return** $RecursiveKnapsack(n-1, v, w, W)$
6: **else**
7: **return** $\max(v[n] + RecursiveKnapsack(n-1, v, w, W - w[n]),$
8: $RecursiveKnapsack(n-1, v, w, W))$
9: **end if**
10: **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**. The source code for the dynamic programming solution to the 0-1 knapsack problem is given below.

```
1  def knapsack(n, values, weights, capacity):
2          table = list()
3          for i in range(0, n):
4                  rowList = list()
5                  for j in range(0, capacity + 1):
6                          rowList.append(0)
7                  table.append(rowList)
8          for i in range(0, n):
9                  for w in range(0, capacity + 1):
10                         if (weights[i] > w):
11                                 table[i][w] = table[i - 1][w]
12                         else:
13                                 vIn = values[i] + table[i - 1][w - weights[i]]
14                                 vOut = table[i - 1][w]
15                                 table[i][w] = max(vIn, vOut)
16         return getItemIndices(table, n, weights, capacity)
17
18 def getItemIndices(values, n, w, capacity):
19         i = len(values) - 1
20         weight = capacity
21         indices = []
22         for i in range(0, n):
23                 indices.append(0)
```

```
24          while i >= 0 and weight >= 0:
25                  if ((i == 0 and values[i][weight] > 0) or
26                          (values[i][weight] != values[i - 1][weight])):
27                          indices[i] = 1
28                          weight = weight - w[i]
29                  i = i - 1
30          return indices
```

**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 depends on the value of $W$, not the size of $W$. Time complexity measurements consider the size of the input of an algorithm (i.e. the number of bits if the input is a numeric value), and since $W = 2^{\lg W}$, where $\lg W$ is the number of bits in $W$, we can see that the resulting time complexity is thus $O(n2^{\lg W})$ in terms of the size of $W$. This means that this dynamic programming solution is a pseudo-polynomial time algorithm in that it is polynomial with respect to the value of the input, but exponential with respect to the size of the input, and thus is not polynomial in the traditional sense. Therefore, we can see that this does *not* prove that $P = NP$.