# CS388T: Answer to Homework 1

Vishvas Vasuki

February 4, 2008

## 1 Question

(10 points) Show that every language that consists of a finite set of strings belongs to Time(n).

### 1.1 Solution

We prove the above by constructing a turing machine which accepts a certain finite language L in O(n).

#### the treeAlgorithm

First, we describe a certain algorithm, called the treeAlgorithm.

**Input**: (a set S of strings, all starting with the same letter s)

**Output**: A certain tree, with 'start symbol' as the root node, 'stop symbol' as leaf nodes, and whose every path from the root to a leaf corresponds to a certain string in S.

1. Make s the root node of a new tree T.
2. If s is the 'stop symbol', return the tree T.
3. Construct a set S' of all strings in S, with f removed from the first position.
4. Construct a set G of the first characters of the strings in S'.
5. For each element g in G, create a set Sg of strings starting with the letter G.
6. For each set Sg, invoke the treeAlgorithm with parameter Sg.
7. Take every tree T' returned by this algorithm, and attach it to the tree T by creating an edge from the root node of T to the root node of T'.
8. Return the tree T.

**Procedure for determining the states and transition functions**

The procedure for determining the states and transition functions of our turing machine is as follows:

- For each string in L, append the 'stop symbol' in the end, and prepend a 'start symbol' in the beginning. Let us call this new set of strings L'. Now, construct a tree by invoking the the treeAlgorithm with the parameter L'.
- By executing the treeAlgorithm, we are left with a tree T, which starts for which the root node is the 'start symbol', and whose leaves are all composed of 'stop symbol'.
- For each level l in T, create a state s(l). Let the set of such states be S.
- Construct an empty transition function D.
- For level l in the tree which contains a 'stop symbol', add the rule D(sl,'stop symbol')=$(yes,'stopsymbol', \_)$ to D.
- For every level l in T, and for each node n in T's lth level, add the rule $D(s(l), n) = (s(l+1), n, \rightarrow)$ to D, if n is not the 'stop symbol'.
- Let Nl be the set of nodes in T's lth level. Let X be the alphabet of L'. For every x in (X-Nl), add the rule $D(sl, x) = (no, x, \_)$ to D.

**Analysis of the above turing machine**

Now, consider starting the above turing machine with string s. If s is in L, the transition of states in the above turing machine corresponds to tracing the string s in the corresponding tree T defined above. Every path in tree T from the root (start symbol) to any leaf (stop symbol) corresponds to a certain string s in L. So, for any string s in L, the length of the corresponding path, defined as the number of levels it passes through, is $|s| + 2$. But, every level l in T corresponds to a certain state sl. Hence, for every string s in L, the transition to the accepting state involves exactly $|s| + 2$ state transitions. Similarly, for any s not in L, our turing machine exits within $|s| + 2$ state transitions.

Hence, any L with finite number of strings is in TIME(O(n)).

# 2 Question

(10 points) Problem 1.4.15 (a), (b) and (c).

Suppose that we are given an instance of the TSP with n cities and distances $d(i, j)$. For each subset S of cities excluding city 1, and for each j in S, define c[S,j] to be the shortest path that starts from city 1, visits all cities in S and ends up in city j.

a. Give an algorithm that calculates c[S,j] by dynamic programming, that is, progresing from smaller to larger sets S.

b. Show that this algorithm solves the TSP in time $O(n^2 2^n)$. What are the space requirements of the algorithm?

c. Suppose we wish to find the shortest (in the sense of sum of weights) path from 1 to n, not necessarily visiting all cities. How would you modify the above algorithm? (Is the reference to S necessary now, or can it be replaced by simply $|S|$?) Show that this problem can be solved in polynomial time.

## 2.1 Solution

We assume that d(i,j) = d(j,i). Note that the number of cities is n.

Note that c[S,j] is the shortest path that starts from city 1, visits all cities in S and ends up in city j. In the following analysis, we will rewrite c[S,j] as c[{j1,j2,...., jk,j},j]. TSP is equivalent to the problem of finding c[1,2...n,1]. We will use c[S,j] in two senses: [a] to denote the path, as described above; and [b] to describe the length of that shortest path.

Note that, for any c[{j1,j2,...., jk,j},j] = $min_{1 \leq m \leq k}$ c[{j1,j2,...., jk},jm] + d(jm,j). Also note that c[a,b,b] = d(1,a) + d(a,b).

**a.** Hence, from the above, the algorithm can calculate c(S,j) for any subset of the n cities. This is done as follows: We deliberately ignore all subsets of size 1 or 0, as they are immaterial for solving TSP. For all subsets of size 2 or more, we can calculate c(S,j) for each j in S, using the identity c[{a,b},b] = d(1,a) + d(a,b).

**cFinderAlgorithm**: In general, when we know c(S,jm) for a certain subset S of size k and for each jm in S, we can find $c(S \cup j)$ by using the fact that c({j1,j2,...., jk,j},j) = $min_{1 \leq m \leq k}$ c({j1,j2,...., jk,jm},jm) + d(jm,j).

Thus, we can show that, for any subset S of cities {1,2...n} and for any city j in S, we can find c(S,j). Also, note that we can use the cFinderAlgorithm to find c(1,2...n,1), that is, the solution to TSP.

**Input**: A set of n cities, and the distance function d(a,b) for those cities.

**Output**: The shortest circuit.

**The Algorithm**:

1. For all x from 2 to n:

   For all subsets S of the n cities of size x

   For all cities j in S

   Find c(S,j) using c({j1,j2,...., jk,j},j) = $min_{1 \leq m \leq k}$ c({j1,.., jk,jm} ,jm) + d(jm,j), as described in the dynamic programming formulation above.

2. return c(set of all cities,1)

**b.** We calculate the time and space complexity, assuming that the algorithm runs on the RAM computational model. Note that for cFinderAlgorithm to work, it must find c(S,j) for all subsets S whose size is greater than 1 and for all j in S. Note that there are $\sum_{k=2}^{n} k\binom{n}{k}$ ordered pairs c(S,j) which need to be evaluated. Note that each evaluation of the identity c({j1,j2,...., jk,j},j) = $min_{1 \leq m \leq k}$

c({j1,j2,...., jk},jm) + d(jm,j) requires (k-1) load, store and addition operations. So, the time complexity of our algorithm becomes $T(n) = l\sum_{k=2}^{n}(k-1)k\binom{n}{k}$ (where l is a constant). But, we know that $k\binom{n}{k} = n\binom{n-1}{k-1}$. Hence, T(n)= $l\sum_{k=2}^{n}(k-1)n\binom{n-1}{k-1} = l\sum_{k=2}^{n} n^2\binom{n-2}{k-2} = ln^2 2^{n-2}$. Hence, cFinderAlgorithm solves TSP in $O(n^2 2^n)$.

The total number ordered pairs (S,j) which are evaluated is $O(n2^n)$. If we store only the cost corresponding to each ordered pair (S,j), the space required by the algorithm in such a case is $O(n2^n)$. However if we store the path corresponding to each c(S,j), we will be required to store up to n cities. Hence, the **space** required by the algorithm in this case is $O(n^2 2^n)$. (Just to cross check: we know that a computational machine cannot require a greater order of memory than the running-time.)

**c**. Suppose we wish to find the shortest (in the sense of sum of weights) path, s(dest), from 1 to dest, not necessarily visiting all cities. Instead of c(S,j), we use and define c(x,j) to be the shortest path from 1 to j through at most x cities (j included). We will also use c(x,j) to represent the length of that shortest path. Our objective, using this terminology, is to find c(n-1,j). Note that $c(1,j) = d(j,1)$. For convenience, we define d(j,j)=0. Also note that $c(x,j) = min_{1\leq k\leq x-1} c(x-1,k) + d(j,k)$.

We would modify the algorithm as follows: Now, consider the set S, the set of all given cities except 1. First we find c(1,j) for all cities in S. Then we find c(2,j) for all cities in S. We keep increasing x, until finally, we find c(n-1,j) for j.

**Analysis**: We analyse the algorithm with a RAM computational model. For any value of $x > 0$, there exist exactly n-1 functions c(x,city): one for each city except the city 1. So, as our algorithm calculates c(x,city) for (n-1) values of x, and n-1 values of city, there are $(n-1)^2$ computations of c(x,city). Note that each computation of $c(x,j) = min_{1\leq k\leq x-1} c(x-1,k) + d(j,k)$ requires merely the use of the values for c(x-1,city) and the n-2 distances d(j,city) for all values of 'city'. Hence, each computation of c(x,city) requires $\Theta(n)$ operations. Hence, the time complexity of our algorithm is $(n-1)^2\Theta(n) = \Theta(n^3)$. Hence, our algorithm is polynomial.

# 3 Question

(10 points) Prove that atleast $\Omega(nlogn)$ comparisons are necessary to produce a sorted list of n elements. (Assume that your input is a list of n integers, and that you have to produce a list of them in increasing order. We are only interested in the number of comparisons you need to make.)

## 3.1 Solution

We assume that the n integers are distinct. The task of sorting is equivalent to the following task:

1. Choose a certain permutation of the list (1,2,....n), p. In this permutation p, i occupies the position p(i).

2. Apply that permutation to the input list L by moving the number in the ith position to the position p(i).

There exist n! distinct permutations for a set of n distinct integers. Thus, sorting involves the selection of 1 permutation out of these n! distinct permutations. Also, every sorting algorithm must be capable of choosing every one of these n! distinct permutations, depending on the input list it is given.

Now, we consider a sorting algorithm based on comparisons. By 'sorting algorithm based on comparisons', we mean a sorting algorithm whose decisions are based on the decisions it takes after each comparison. Every comparison is a binary operation, as it operates on two integers, a and b. Given that all elements in our input set are distinct, every comparison can have exactly the following outcomes: $a > b$ or $a < b$ or $a = b$. But, for the sorting algorithm, there can be exactly **two decisions** : 1. place integer a to the left of integer b, or 2. place integer b to the left of integer a.

In other words, based on the outcome of each comparison, one can distinguish between at most two sets of permutations. So, with m comparisons, the maximum number of permutations an algorithm can distinguish is $2^m$. So, the minimum number of such comparisons required for an algorithm to distinguish n! permutations is log n!.

Note that $n! \geq (n/2)^{(n/2)}$. So, $log(n!) \geq (n/2)log(n/2) = O(nlog(n))$ Hence, atleast $\Omega(nlog(n))$ comparisons are necessary for any algorithm to produce a sorted list of n elements.

# 4 Question

(10 points) Prove that any language decided by a k-string nondeterministic Turing machine within time f(n) can be decided by a 2-string nondeterministic turing machine within time O(f(n)).

HINT: Note that in general we cannot assume that the simulating machine 'knows' the function f. This is part of the challenge in this problem.

## 4.1 Solution

**Acknowledgement**: In solving this problem, I benefited from discussion with Daniel Lessin and Jing-Tang Jang.

Let the k-tape non deterministic turing machine be called KNDTM. Let the 2-tape non deterministic turing machine be called 2NDTM. Let the first tape of 2NDTM be called T1 and, let the second tape of 2NDTM be called T2. Let x be the input KNDTM starts with. First we describe a way to **simulate the operation of KNDTM on 2NDTM**. We assume that, initially, x is written after the start symbol in T1.

We define a **snapshot** to be a tuple which includes the following: a KNDTM state, the symbols under the head of each of the k tapes of KNDTM, one of the 3 possible directions of movement for each of the k tapes. The size of this snapshot is $\Theta(k)$.

**Step 1**: 2NDTM is a non deterministic turing machine: So it can enter multiple computational paths simultaneously. 2NDTM **guesses** a sequence of m snapshots, and writes this sequence to T2. By this, we mean that, for any value of m in the set of natural numbers, 2NDTM follows computational path which directly produces all possible sequences of m snapshots. Every computational path which 2NDTM follows is completely defined by the value of m and the snapshot sequence it produces. We say a snapshot sequence is **valid** for the input x, if it corresponds to a sequence of valid transitions of KNDTM on the input x.

Now, consider one computational path. Suppose that there is a sequence of m snapshots which 2NDTM has produced. As the size of each snapshot is $\Theta(k)$, the length of this snapshot sequence is $\Theta(km)$. We define 2NDTM's transition function in such a way that it produces this snapshot sequence in time $\Theta(km)$. (This can be done by defining a state of 2NDTM, during which it produces a snapshot, and by creating a non deterministic transition function which, when in this state, either stops, or creates another snapshot and appends it to the sequence produced earlier.)

**Step 2**: In each computational path, 2NDTM then verifies the validity of the snapshot sequence produced. Suppose that the length of the snapshot sequence produced is m. For each successive pair of snapshots, s1 and s2, 2NDTM verifies that there is a transition function in KNDTM which will allow the transition from the state corresponding to s1 to the state corresponding to s2, considering the k head symbols and movement directions recorded in s1 and s2. It is important to note that 2NDTM does it by verifying transitions on one tape at a time. So, this verification step costs 2NDTM O(km) time. If it finds, using the above test, that the snapshot sequence is invalid, the computation path ends.

**Step 3**: On the other hand, in any 'valid' computational path, if it finds that the snapshot sequence is valid, 2NDTM next checks to see whether KNDTM would have halted computation at the end of the corresponding sequence of state transitions. If KNDTM would have halted in the state "yes", 2NDTM accepts the string x. If KNDTM would have halted in the state "yes", 2NDTM accepts the string x. If KNDTM would have halted in the state "no", the computation path stops, but 2NDTM will not yet reject the string. If KNDTM would not have halted at the end of the corresponding state transitions, the computation path continues, but in this solution, we don't care how that computational path operates after this point, as useful computation is being carried out in other computational paths: the ones for which m is greater.

Now, we consider the a string w in L. For any such string, we are told that KNDTM would have accepted that string. It follows, from the construction of 2NDTM described above, that w would be accepted by 2NDTM too, as there will exist a certain computational path with the right snapshot sequence. We

are told that KNDTM accepts any string w in L in time f(n). So, this snapshot sequence is O(kf(n)) long. Considering our construction of the snapshot sequence, the time spent by 2NDTM in generating this snapshot sequence is O(kf(n)). The time taken by 2NDTM in verifying a f(n)-long snapshot sequence is also O(kf(n)).

Hence, we have proved that any string w in L, which is accepted by KNDTM in f(n) time is accepted by 2NDTM in O(kf(n)) + O(kf(n)) = O(f(n)).