CSCI 2400 – Models of Computation, Section 3

Solutions to Homework 8

Problem 1.

(a) We will use the abbreviation TSP for *Traveling Salesperson Problem*. We will denote any path which starts from city c_1, then goes through all the cities, and returns back to c_1 as a *TSP-path*. The *optimal TSP-path* has the shortest length.

Denote by A^* the length of the optimal TSP-path. The value A^* is not known at the beginning, and we will find it with a binary search procedure. At the same time we will find the optimal TSP-path.

For our non-deterministic algorithm, we need the following two routines. The routine `Smaller-TSP-path(`B`)` returns "yes" is there exists a TSP-path with length smaller than B, where B is a parameter, and returns "no" otherwise.

`Smaller-TSP-path(`B`):`

1. Guess a TSP-path p.
2. Check if p has length smaller than B.

   If yes then return "yes". Otherwise, return "no".

Notice that the above routine is non-deterministic. If there exists a TSP-path p with length smaller than B then we are guaranteed that this routine will return the answer "yes", since there is a possible computation in step 1 of the routine that gives path p.

In a similar way we define the routine `Exact-TSP-path(`B`)`, which returns a TSP-path p with length B if such a path exists, and otherwise it returns "no".

`Exact-TSP-path(`B`):`

1. Guess a TSP-path p.
2. Check if p has length equal to B.

   If yes then return the TSP-path p. Otherwise, return "no".

Now we are ready to give the main algorithm. Let's assume that the total sum of all the route distances $\sum_{i,j} d_{i,j}$ is smaller than 2^n. With this assumption, the length of the optimal TSP-path is smaller than 2^{n+1}.

`Find-optimal-TSP-path:`

1. Set $A = 2^n$ and $D = 2^{n-1}$.
2. Execute `Smaller-TSP-path(`A`)`.

   If the answer is "yes" then set $A = A - D$.

If the answer is "no" then execute `Exact-TSP-path(`A`)`. If the result is a path p then return the path p. Otherwise, set $A = A + D$.

3. Set $D = D/2$.

4. Go to step 2.

The above algorithm returns the optimal TSP-path. This algorithm is a binary search procedure where we try to find the value of A^*. The value of variable A holds the approximation of A^*. Every time we compute step 2, A is closer to the value of A^* by the amount of D. After n repetitions the value of variable A is equal to the value of A^*, and at that point the path computed by `Exact-TSP-path(`A`)` is the optimal TSP-path.

Let's examine now the time needed for the above algorithm. Each execution of routines `Smaller-TSP-path(`B`)` and `Exact-TSP-path(`B`)` takes time at most $O(n)$, since guessing the TSP-path p takes time $O(n)$ (there are $n-1$ cities to guess), and comparing the length of p with B takes time $O(n)$. The steps 2-4 in routine `Find-optimal-TSP-path` are repeated n times (namely, $log(2^n) = n$). At each repetition we execute the routines `Smaller-TSP-path(`A`)` and `Exact-TSP-path(`A`)` which take time $O(n)$. Therefore, the total time is $n \cdot O(n) = O(n^2)$. This time is polynomial in n.

(b) In the deterministic algorithm we search all the TSP-paths. A TSP-path path is a sequence of cities that starts and ends at city c_1. All the possible paths are all the possible permutations of the cities $c_2, \ldots, c_n$ (the intermediate cities in a TSP-path). There are $(n-1)!$ such permutations and therefore $(n-1)!$ TSP-paths.

The following routine returns the first permutation:

`First-Permutation:`

1. Return the sequence $c_2, c_3, \ldots, c_n$

The following routine returns the next permutation, given a permutation P. This routine is recursive.

`Next-Permutation(`P`):`

1. In the permutation P find the position i of city c_n. Let P' be the subsequence right from city c_n

2. If P' is the final permutation ($P' = c_{n-1}c_{n-2}\ldots c_i$) then swap c_n with the city on its left. (If c_n is leftmost then P is the final permutation and simply return).

   Otherwise, find the next permutation of P' (using `Next-Permutation`) with i the smallest city index and $n-1$ the largest city index. Store the result to P''.

3. Replace P' with P''. Return the new permutation.

Now we are ready to give the deterministic routine that finds the optimal TSP-path. We search all the possible TSP-paths. We keep in variable q the shortest TSP-path found so far. At the end, q will hold the optimal TSP-path.

`Find-optimal-TSP-path:`

1. Execute `First-Permutation` and store the result in P. From P create the respective TSP-path p (add c_1 at the beginning and end of P). Set $q = p$.
2. Execute `Next-Permutation(`P`)` and store the result to P. Create from P the respective TSP-path p.
3. Compare the length of p with the length of q. If the length of p is smaller then set $q = p$.
4. Compare P to the final permutation $c_n, c_{n-1}, \ldots, c_2$. If P is the same with the final permutation then return q. Else goto step 2.

Let's examine now the time needed for the algorithm. Each execution of routines `First-Permutation` and `Next-Permutation`, takes time $O(n)$. Steps 2,3, and 4 of routine `Find-optimal-TSP-path` are executed $(n-1)!$ times. At each iteration the routine `Next-Permutation` is executed once. Therefore, the total number of steps is $O(n! \cdot n) = O(n!)$. This time is exponential (From Sterling's formula).

There is no known polynomial time algorithm that solves the TSP problem.

Problem 2.
(a) The proper order of strings of the form $a^n b^n$ is:

$$ab, aabb, aaabbb, aaaabbbb, \ldots$$

(in proper order we first print strings with smaller length). Below is the algorithm of the Turing machine that prints the strings in proper order. By "print" we mean: write on the tape.

`Procedure Enumerate:`

1. Print ab.
2. Add an a to the left and a b to the right of the previously printed string. Print the new string.
3. Go to step 2.

(b) No, for this language the proper order is different than the lexicographic order. In particular, the lexicographic order is the opposite of the proper order. For example, in the lexicographic order the string *aabb* appears before string *ab*.

Problem 3.
(a) Let's assume that we have two countable sets S_1 and S_2 (these sets may be infinite). We want to prove that the set $S_1 \cup S_2$ is also countable. In other words, we want to find an enumeration procedure that prints the elements of $S_1 \cup S_2$ in some order.

Since S_1 is a countable set there is a Turing machine M_1 that enumerates the elements of S_1. Similarly, there is a Turing machine M_2 that enumerates the elements of S_2.

Using M_1 and M_2 we can construct a Turing machine M that enumerates the elements of $S_1 \cup S_2$ as follows. Machine M interleaves the computations of M_1 and M_2. Machine M first allows machine M_1 to print one element of S_1, then allows machine M_2 to print one element of S_2, and then the process is repeated. Therefore, machine M prints the sequence

$$x_1, y_1, x_2, y_2, x_3, y_3, \ldots$$

where x_i are the elements of S_1, and y_i are the elements of S_2.

(b) We want to prove that the set of non recursively enumerable languages is uncountable.

Let S be the set of all languages over an alphabet. We know that the set S of all languages is uncountable (see Theorem 11.2, page 289). The set S is the union of two sets S_1 and S_2:

- Set S_1 consists from all the recursively enumerable languages (the languages accepted by Turing machines).
- Set S_2 consists from all the non recursively enumerable languages (notice that $S_2 = \overline{S_1}$).

We want to prove that S_2 is uncountable. We have $S = S_1 \cup S_2$. We know that the set S_1 is countable, since the set of Turing machines is countable (see Theorem 10.3, page 280, and Theorem 11.2, page 289).

If S_2 was countable then from part (a) of this problem we would have that the set S is also countable (the union of two countable sets is countable). But this is impossible, since set S is uncountable. Therefore, it must be that set S_2 is uncountable.

Problem 4.
We want to prove that if a language L is not recursively enumerable then its complement $\overline{L}$ cannot be recursive.

Assume for contradiction that the language $\overline{L}$ is recursive. Then according to the definition of recursive languages (Definition 11.2, page 286) there is a

Turing machine $\overline{M}$ that accepts $\overline{L}$ and this machine halts for every input string w.

We will modify $\overline{M}$, and we will construct a Turing machine M that accepts L as follows. Let q be a halting state of $\overline{M}$ (we can easily identify the halting states because they have no outgoing transitions). If q is a final state then we make it a non-final state, and if q is a non-final state then make it a final state. We do the same for all the halting states of $\overline{M}$.

It is easy to see that if $\overline{M}$ halts in a final state then M halts in a non-final state, and vice-versa. Therefore, if string w is accepted by $\overline{M}$, then w is not accepted by M, and vice-versa. Subsequently, machine M accepts language L. Therefore, language L is recursively enumerable (more specifically, language L is recursive). This is a contradiction, since language L is not recursively enumerable. Thus, our original assumption that the language $\overline{L}$ is recursive must be wrong, and therefore, language $\overline{L}$ is not recursive.

Problem 5.

Suppose that language L is such that there is a Turing machine M' that enumerates the elements of L in proper order. We want to prove that L is recursive. In other words, we want to construct a Turing machine M that accepts L and halts for every input string w.

The construction of M uses the machine M'. For input string w, machine M does the following:

- M computes the length of w. Let n be the length of w.
- M uses M' to enumerate all strings with length up to n, and compares each enumerated string with w:
  - If M finds a string which is is the same with w then M halts in a final state.
  - If M doesn't find any string the same with w then it halts in a non-final state.

Machine M always halts for an input string w, since machine M' enumerates the strings of L in proper order (M' prints first the strings of smaller length). Furthermore, if $w \in L$ then machine M will halt in a final state. Therefore, L is recursive.