Advanced Complexity

TD n°1 : SPACE and NL

Charlie Jacomme

September 13, 2017

Exercise 1 : Graph representation and why it does not matter

Let $\Sigma = \{0, 1, ;, \bullet\}$, $n \in \mathbb{N}$ and $V = [0, n-1]$. We consider the following two representations of a directed graph $G = (V, E)$ by a word in Σ^* :

— By its adjency matrix : $m_{0,0}m_{0,1} \ \dots m_{0,n-1} \bullet \cdots \bullet \ m_{n-1,0} \dots m_{n-1,n-1}$, where for all $i, j \in [0, n-1]$, $m_{i,j}$ is equal to 1 if $(i, j) \in E$, 0 otherwise.

— By its adjency list : $k_0^0; \dots ; k_{m_1}^0 \bullet \cdots \bullet k_0^{n-1}; \dots ; k_{m_{n-1}}^{n-1}$, where for all i, $[k_0^i, \dots, k_{m_i}^i]$ is the list of neighbors of vertex i, written in binary, in increasing order.

1. Describe a logarithmic space bounded deterministic Turing machine which takes as input the graph G, represented by adjacency lists, and returns the adjacency matrix representation of G.
2. Conversely, describe a logarithmic space bounded deterministic Turing machine taking as input a graph G, represented by its adjacency matrix , and computing the adjacency list representation of G.

Therefore, the complexity of the problem `REACH` seen in class does not depend on the representation of the graph.

Solution:

We first need to be able to implement a binary counter, which can be done by using an empty tape B. To increment, we read B left to right, replacing 1s by 0s and stopping at the first 0, replacing it by a 1. If we reach the end of B, we add a 1. Then, we reset the head to the start of the tape, and the incrementation is over.

1. Using a counter, we can obtain the binary writing of N in reverse order by counting the number of $\bullet$ in the input tape. Now, M just have to read the adjency list, and write a 1 at position $N \times i + j$ for every j found in the list L_i, and O's everywhere else. M can do this by using a counter i incremented when encountering $\bullet$, and by collecting all the bits she sees between two " ;" in a tape j. To move to position $N \times i + j$, we finally need two other counters i_O and j_O, writing at every step a 0 and a 1 at the final one.. Thus, if we reuse B, we only need 4 counters of size $log_2(N)$.
2. M initiate two counters i and j and loops between $i = 0$ to $N - 1$ and $j = 0$ to $N - 1$, which is done by incrementing the counters and comparing to N (we can compute N as previously). For each (i, j) M checks the value at $N \times i + j$ (with two counters as previously), and if it is a 1 M writes j on the output tape followed by a " ;". Each times M increment i, she writes a $\bullet$ on the output tape. This uses a space of $5log_2(N)$, and the size of the entry is $n = N^2$, we do have $5log_2(N) = O(logn)$. We conclude using the speed up theorem.

Remark that the speed up theorem is required here because we expect a logarithmic space bounded TM. Also, keep in mind that the complexity of `REACH` only because we can effectively compose logspace machines, which is not a trivial result.

Exercise 2 : Inclusions of complexity classes

Definition 1. *A function $f : \mathbb{N} \to \mathbb{N}$ is said to be* space-constructible *if $\forall n \in \mathbb{N} f(n) > log(n)$ and there exists a deterministic Turing machine that computes $f(|x|)$ in $O(f(|x|))$ space given x as input.*

Show that for a space-constructible function,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{O(f(n))} + O(n))$$

Solution:

We have a Turing Machine M running in space $O(f(n))$, i.e $\beta \times f(n)$, with states Q, k tapes and an alphabet Γ, with f space constructible.

— With $n = |x|$, the number of a configuration of M on input x is inferior, with β a constant, to

$$|Q| \times (\beta f(n))^k \times |\Gamma|^{\beta k f(n)} \times n$$

The last n is required for the input tape head, which is not counted inside the k working tapes. If $f(n)$ was not space constructible, it would be impossible to consider one instance configuration of the graph. The size of the configuration graph is thus in $2^{O(f(n))}$.

— In order to decide if x is accepted by M, we just have to use a breadth first search algorithm on the graph to check whether $C_{initial}$ and C_{accept} are connected. This algorithm is polynomial in the size of the graph, so we finally do have a time complexity in $2^{O(f(n))}$.

Exercise 3 : Restrictions in the definition of $\mathsf{SPACE}(f(n))$, and why they do not matter

In the course, we restricted our attention to Turing machines that always halt, and whose computations are space-bounded on every input. In particular, remember that $\mathsf{SPACE}(f(n))$ is defined as the class of languages L for which there exists some deterministic Turing machine M that always halts (i.e. on every input), whose computations are $f(n)$ space-bounded (on every input), such that M decides L.

Now, consider the following two classes of languages :

— $\mathsf{SPACE}'(f(n))$ is the class of languages L such that there exists a deterministic Turing machine M, running in space bounded by $f(n)$, such that M accepts x iff $x \in L$. Note that if $x \notin L$, M may not terminate.

— $\mathsf{SPACE}''(f(n))$ is the class of languages L such that there exists a deterministic Turing machine M such that M accepts x using space bounded by $f(n)$ iff $x \in L$ (M may use more space and not even halt when $x \notin L$).

1. Show that for a space-constructible function $f = \Omega(logn)$, $\mathsf{SPACE}'(f(n)) = \mathsf{SPACE}(f(n))$
2. Show that for a space-constructible function $f = \Omega(logn)$, $\mathsf{SPACE}''(f(n)) = \mathsf{SPACE}(f(n))$

Solution:

1. First, $\mathsf{SPACE}(f(n)) \subseteq \mathsf{SPACE}'(f(n))$. Conversely, let there be M as specified, with $a = |\Gamma|$, M terminates in times $O(a^{f(n)})$ or does not terminate (she cannot run more than the number of possible configurations). We construct M_0 from M by adding a counter B (basically, a timeout) incremented by 1 at every steps of M. B is written in base a, and as soon as B takes more bit than necessary to write the number of possible configurations, we reject (the maximum size of B can be set in space f(n) because f(n) is space constructible). This uses $O((fn))$ space, we conclude with the speed-up theorem.
2. First, $\mathsf{SPACE}(f(n)) \subseteq \mathsf{SPACE}''(f(n))$. Conversely, we want to force M to only use space $f(n)$ on every input. We construct M_0, which starts by writting $f(n)$ 0 on

a tape B ($f(n)$ space constructible). M_0 also contains the tapes of M, and copy B those tapes, adding a special symbol #at the end of each of them. M_0 then simulates M, but rejecting if she go over of #. If x is in L, M runs in space $f(n)$ and M_0 accepts. If $x \notin L$ M_0 cannot accept x. So M_0 accepts x iff $x \in L$, and M_0 runs in $O(f(n))$ space, so $M_0 \in \mathsf{SPACE}'(O(f(n))) \subseteq \mathsf{SPACE}(f(n))$.

Exercise 4 : Dyck's language

Let A be the language of balanced parentheses – that is the language generated by the grammar $S \to (S)|SS|\epsilon$. Show that $A \in L$.

— What about the language B of balanced parentheses of two types ? that is the language generated by the grammar $S \to (S)|[S]|SS|\epsilon$

Solution:

— Read the input from left to right. Maintain a counter on the worktape with initial value zero, and increment or decrement it when reading '(' or ')' respectively. Reject if the counter ever becomes negative, and accept if the counter is zero at the end of the input. Since the counter can never exceed the input length n, it is a $log_2(n)$-bit number. More generally, any language which can be recognized by a one-counter machine or a machine with any constant number of counters is in L.

— Let us call say that each symbol has a type, either round or square, and say that each symbol is a left or right bracket regardless of type. Each left bracket has a right bracket which is its partner, and our goal is to check that every left bracket's partner is of the same type. To find its partner we use a counter as in question 1 above. First, we check that the word is in the bracket language of question 1 if we ignore round vs. square, so that every left bracket has a right partner. Then, to check that partners match, we use the following pseudocode :

```
i = 1
do until i exceeds the length of the input {
    move i-1 steps from the left end of the input
    read the input symbol a    // a = w(i)
    if a is a left bracket {
    c = 1
    do until c = 0 {    // find w(i) partner
        move right and read the next input symbol b
        if b is a left bracket, increment c
        if b is a right bracket, decrement c
        }
    if a and b are of different types, reject
    }
}
accept
```