Aniello Murano

Le Classi L e NL.

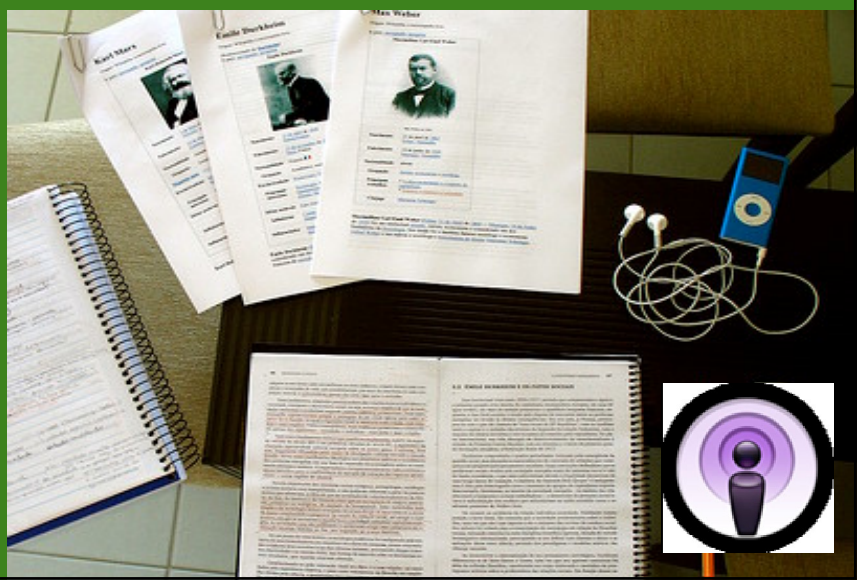
Lezione n.20
Parole chiave:
L e NL

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Overview

- Le classi di tempo sub-lineari non sono di interesse nella teoria della complessità perchè per risolvere un problema con input di taglia **n** si richiede almeno un tempo necessario a leggere l'input.
- Viceversa, è utile pensare a problemi che richiedono spazio sub-lineare.
- Per discutere di questi problemi è conveniente modificare leggermente la macchina di Turing di riferimento, ovvero distinguiamo il nastro di input da quello di lavoro. Quest'ultimo rimane di lettura e di scrittura, mentre quello di input è di sola lettura.
- Per il calcolo della complessità facciamo poi riferimento a quante celle del nastro di lavoro sono utilizzate.
- Questa separazione non è completamente artificiale se si pensa che spesso la "memoria" di input di un computer è ben più grande di quella di lavoro. In alcuni microcomputer poi, la memoria è pressoché irrisoria.

Definizione di L e NL

Definizione:

1. **L** è la classe dei linguaggi che sono decidibili in spazio logaritmo su una macchina di Turing deterministica. In altre parole,

$$L = SPACE(\log n)$$

2. **NL** è la classe dei linguaggi che sono decidibili in spazio logaritmo su una macchina di Turing non deterministica, in altre parole,

$$NL = NSPACE(\log n)$$

$\{0^k1^k \mid k \geq 0\} \in L$

- Si consideri il linguaggio $\{0^k1^k \mid k \geq 0\}$ già visto diverse volte in passato.
- Questo linguaggio può essere accettato in tempo (e spazio) polinomiale lavorando a zig-zag sul nastro per controllare carattere per carattere che la sequenza degli 0 corrisponde in numero a quella degli 1.
- Come possiamo migliorare la complessità di spazio?
- Invece di lavorare a zig-zag direttamente sull'input, con un solo passaggio prima annotiamo in binario il numero **m** di 0 e **n** di 1.
- Per annotare **m** ed **n** occorre spazio logaritmico (ma tempo lineare).
- Poi si può procedere controllando che **m** ed **n** siano uguali, lavorando a zig-zag, ma questa volta su un oggetto logaritmico nell'input di partenza.

PATH ∈ NL

- Sia PATH = { (G,s,t) | G è un grafo e contiene un percorso da s a t}
- PATH può essere deciso in tempo polinomiale da una macchina di Turing deterministica, semplicemente marcando i nodi in G partendo da t e cercando a ritroso s.
- PATH può anche essere deciso in spazio logaritmo utilizzando una macchina di Turing nondeterministica M.
- Partendo da s, la macchina M fa un guess non deterministico del prossimo nodo tra quelli in uscita e salta su quest'ultimo.
- Poi ripete l'operazione fino a che non raggiunge t, il tutto in un numero di passi minore o uguale al numero dei nodi che contiene il grafo.
- Ad ogni passo, M deve solo ricordare il nodo corrente e il numero di nodi visitati fino a quel momento.
- Queste informazioni necessitano di spazio logaritmo (con una rappresentazione binaria dei numeri).
- Dunque PATH è in NL

L è uguale a NL?

- Il problema di stabilire se **L** è o meno uguale a **NL** è un altro importante "problema aperto" della teoria della complessità e difficile al pari del problema "**P=NP**?".
- I problemi **NL-completi** sono i problemi più difficili in NL.
- Come per la NP-completezza, la questione **L=NL**? è equivalente a verificare se un qualsiasi linguaggio **A NL-completo** appartiene ad **L**.
- **PATH** è uno di questi problemi.
- Noi mostreremo che **NL⊆P**. Dunque, la **NL-completezza** non può essere definita attraverso la riduzione in spazio logaritmico.
- Non useremo una riduzione in tempo (o spazio) polinomiale, perchè, come detto altre volte, una riduzione ha senso solo quando essa è più semplice del problema stesso.

Riduzione in spazio logaritmico

Definizione.

- Un ***transduttore LogSpace*** è una macchina di Turing con un nastro di input di sola lettura, un nastro di output di sola scrittura e un nastro di lavoro di lettura e scrittura. Sul nastro di lavoro, possono essere utilizzate solo **O(log n)** celle.
- Un trasduttore LogSpace calcola una funziona **f: $\Sigma^* \to \Sigma^*$** sse su ogni input **$w \in \Sigma^*$**, si ferma con **f(w)** sul suo nastro di output..
- Una funzione **f: $\Sigma^* \to \Sigma^*$** è ***LogSpace calcolabile*** sse esiste un trasduttore LogSpace che calcola **f**.
- Un linguaggio **A** è ***LogSpace riducibile*** a un linguaggio **B**, in simboli **$A \leq_L B$**, sse **A** is mapping-reducible a **B** tramite una funzione calcolabile in spazio polinomiale. Ovvero, **f** è tale che, per ogni **$w \in \Sigma^*$**, **$w \in A$** sse **$f(w) \in B$**.

Definizione di NL-completezza

Definizione Un linguaggio **B** è detto ***NL-completo*** se e solo se

1. **$B \in NL$**, e
2. Ogni **A** appartenente a **NL** è riducibile in spazio logaritmico a **B**.

Riduzione e NL-completezza

Theorem: Se $A \leq_L B$ e $B \in L$, allora $A \in L$.

Proof.

- Innanzitutto facciamo vedere che la prova usata per la riduzione polinomiale non può funzionare in questo caso.
- La prova per A potrebbe tentare di mappare ogni suo input w in f(w), usando una riduzione logspace f, e poi applicare a f(w) l'algoritmo logspace per B.
- Ma f(w) potrebbe richiedere più uno spazio non logaritmo e questo non va bene.
- Per evitare questo, usiamo una MdT M_A per A che computa singolarmente i simboli di f(w) per la macchina M_B di B (cioè uno alla volta).
- Nella simulazione, M_A mantiene solo traccia della posizione della testina di M_B su f(w).
- Ogni volta che M_B si muove, M_A fa ripartire la computazione di f su w dall'inizio, ignorando tutti gli output (dell'intera (f(w)) eccetto l'elemento singolo interessato.
- Così facendo, parte di f(w) viene ricomputata più volte e dunque perdiamo in efficienza di tempo, ma come vantaggio abbiamo che ad ogni punto della computazione abbiamo bisogno di memorizzare solo un simbolo di f(w) per volta.

Corollario. Se un linguaggio NL-completo è in **L**, allora **L=NL**.

NL-completezza di PATH(1)

Teorema. PATH è NL-completo.

Prova

- Abbiamo già mostrato che **PATH**$\in$**NL**. Resta da dimostrare che **PATH** è NL-hard. Usiamo una riduzione LOGSPACE da un linguaggio arbitrario **A**$\in$**NL** a **PATH**.
- Sia **M** la MdT nondeterminisitca che decide **A** in spazio logaritmico.
- Dato un input **w**, costruiamo **<G,s,t>** in spazio logaritmico, dove **G** è un grafo diretto che contiene un percorso da **s** a **t** sse **M** accetta **w** (e quindi **w**$\in$**A**)
- I nodi di **G** sono le configurazioni di **M** su **w**. Ogni configurazione consiste del contenuto del nastro di lavoro (dunque, non del nastro in input), la posizione delle due testine sui due nastri, e uno stato di M.
- Per due configurazioni c_1 e c_2 di **M** su **w**, ci sarà un arco da c_1 a c_2 se e solo se **M** può andare da c_1 a c_2. Si noti che questo è pienamente determinato da **w** e dalla funzione di transizione **M**.
- Sia **s** la configurazione iniziale di **M**. Senza perdita di generalità assumiamo che **M** ha una sola configurazione accettante, che indichiamo come nodo **t**.
- Questa costruzione permette di ridurre **A** a **PATH** perchè ogniqualvolta **M** accetta il suo input, esisterà un ramo della sua computazione che accetterà.
- Questo corrisponde ad avere un percorso da **s** a **t** e viceversa.

NL-completezza di PATH(2)

- Rimane da mostrare che la riduzione sia effettivamente in ***LogSpace***.
- Per far questo, mostriamo un trasduttore che in spazio logaritmo, su input **w**, restituisce una descrizione di **G**.
- La riduzione costruisce due liste: I nodi e gli archi di G.
- Si ricordi che ogni nodo è una configurazione di **M** su **w** e può essere rappresentata in spazio **c log n** per qualche costante **c**, dove **n** è la dimensione di **w**.
- Il trasduttore sequenzialmente prende in esame tutte le possibili stringhe di lunghezza **c log n**, controlla se ognuna è una configurazione legale di **M** su **w**, e restituisce quelle corrette
- In modo simile, il trasduttore può elenca gli archi.
- Uno spazio logaritmico è ovviamente sufficiente per verificare che una configurazione c_1 di **M** su **w** può portare ad una configurazione c_2.
- Il trasduttore prende in esame sequenzialmente tutte le coppie (c_1,c_2) per trovare quelle che effettivamente formano un arco in **G**, per poi aggiungerle al nastro di output.

NL⊆P

Corollary 8.26 **NL⊆P** (and hence also **L⊆P**).

Proof. Theorem 8.25 showed that every language in NL is log space reducible to **PATH**. And we know that **PATH∈P**. So, all we need to see is that log space reducibility implies polynomial time reducibility.

In other words, we need to see that every TM **M** that operates in **log** space operates in polynomial time.

But indeed, on an input **w** of length **n**, the work tape of such a machine **M** utilizes only **O(log n)** cells. So, there are only $2^{O(\log n)} = n^{O(1)}$ different possible contents of the work tape. Plus, there are **O(n)** different possibilities for the locations of the two heads of the machine.

Thus, altogether there are $n^{O(1)} \times O(n) = n^{O(1)}$ different possible configurations of **M** on **w**. This means that **M** can only make $n^{O(1)}$ (polynomially many!) steps on input **w**, for otherwise a configuration would repeat and an infinite loop occur.

NL=coNL

coNL is defined as {A | the complement A' of A is in NL}.

Theorem 8.27 NL = coNL.

Proof Idea. It is sufficient to prove the following:

PATH∈coNL, i.e. PATH'∈NL (*)

Indeed, assuming that (*) holds, consider any language **A** in NL. Since PATH is NL-complete, there is a log space reduction **f** of **A** to PATH. The same **f**, of course, is also a log space reduction of **A'** to PATH'. By (*), however, PATH'∈NL. Hence **A'**∈NL and thus **A**∈coNL.

For the opposite direction, consider any language **A** in coNL, so that **A'**∈NL. Then there is a log space reduction **f** of **A'** to PATH. Hence **f** is also a reduction of **A** to PATH'. By (*), however, PATH'∈NL. So, **A**∈NL.

To summarize, as long as (*) is true, we have **A∈NL** iff **A∈coNL** for all **A**, meaning that NL=coNL. It remains to understand why (*) holds. This will be done through constructing an NL algorithm **M** for PATH'.

Relazioni conosciute su L, NL, coNL, P e PSPACE

This is what we know:

1. L ⊆ NL = coNL ⊆ P ⊆ PSPACE

2. L, NL, coNL ≠ PSPACE

Open problems:

1. L = NL, coNL?

2. NL, coNL = P?

3. P = PSPACE?

At least one of these two equalities does not hold; but which one we do not know (probably both)