# Sviluppo di un motore di ricerca: un esempio reale, Ubi

*Marco Olivo, Matteo Coloberti*

Relatori indipendenti

# Scaletta dell'intervento

L'intervento è diviso in 4 parti:

- **introduzione** (Marco Olivo)
- **crawling ed indicizzazione** (Matteo Coloberti)
- **recupero dati a fronte di query + demo** (Marco Olivo)
- **sessione di domande e risposte**

# Introduzione

- chi sono Marco Olivo e Matteo Coloberti?
- seminario sullo sviluppo di un motore di ricerca **reale**
- si parla di algoritmi di ricerca e loro implementazioni
- si sfatano alcuni miti
- **target-audience**: persone interessate in programmazione, ricerca e sviluppo e curiosi
- **livello di difficoltà**: intermedio

# Il punto di partenza...

**Punto di partenza:**

- due tesi su indexing/search e studio grafo del web
- oltre 90 classi Java di un crawler sviluppato da IAT/CNR e DSI (Unimi): http://ubi.imc.pi.cnr.it/projects/ubicrawler/
- 8 macchine per il crawling, 3 per lo sviluppo, 1 server SUN, strumenti open-source

# ... gli obiettivi...

- **obiettivo primario:** un motore di ricerca sfruttabile a fini di ricerca accademica
- **obiettivo secondario:** fare meglio dei motori "free" già esistenti
- **obiettivo utopico:** fare, come qualità, (almeno) come Google

# ... e il punto di arrivo

**Punto di arrivo:**

- un sistema di pre-processing dei dati funzionante
- un parser HTML non “stretto”
- una interfaccia per le query
- tempi di risposta più che accettabili con qualità più che accettabile

Inoltre, i membri del team originario hanno rilasciato strumenti per l’indicizzazione: http://vigna.dsi.unimi.it/

# Problematiche generali da affrontare

**Problemi:**

- ottenere degli *store* di pagine HTML
- creare il grafo dei documenti: nodi le pagine, archi i *link*
- creare degli indici per fare ricerche dei contenuti: *query*
- sviluppare algoritmi generali di ricerca: PageRank, Proximity, Hits
- sviluppare nuovi algoritmi per migliorare i risultati: ancore, titoli, URL

# Tre fasi

Il processo di distingue in **tre fasi**:

- recupero pagine Web in grossi *store* su disco (circa 100-200GB)
- indicizzazione
- *query* online

# fase 1 - La fase di crawl

Cosa è la *crawl*?

- è il processo che colleziona documenti dal Web

**Problemi:**

- dimensioni del Web _ difficili da stimare
- quantità di dati _ (quasi) impossibile memorizzare tutto
- processo estremamente oneroso in banda e risorse hardware

**Soluzione:** affrontare il problema con metodologie distribuite e *fault-tolerant*

# fase 1 - Differenti tipi di crawl

- 3 tipi di *crawl*:
  - **extensive crawl**: recupero di tutte le pagine di un sottoinsieme del Web
  - **focused crawl**: recupero delle pagine rilevanti rispetto ad un argomento di interesse
  - **incremental crawl**: recupero delle pagine nuove o modificate dall'ultima crawl
- UbiCrawler permette di scegliere i domini di interesse

  Es.:
  - il Web italiano .it
  - il Web anglosassone .uk
  - ...

# fase 1 - UbiCrawler

UbiCrawler:

- esegue un *crawl* distribuito
- è indipendente dalla piattaforma: completamente sviluppato in Java
- è utilizzato attualmente per scopi di ricerca
  - studio del grafo del Web: complessità, proprietà e caratteristiche
  - analisi di pagine Web appartenenti a particolari domini
  - studio dei metodi di compressione del grafo del Web
- è ora parte di un progetto più ampio che comprende anche un motore di ricerca: UbiSearch

# Digressione: perché Java?

- Java è **sufficientemente** veloce (circa 10% in meno JIT compiler vs gcc **coi nostri job**)
- il debugging con Java è semplice (stack eccezioni, etc.)
- portabile ovunque
- *rapid-prototyping*: Java ha classi per RMI, *multi-threading*, strutture dati, rete, etc.

⇒ *saves headaches and time*

# fase 1 - Visitare il Web (1)

**Problema:** come si recuperano le pagine del Web?

- si parte da un **seme**: elenco iniziale di URL, redatto più o meno automaticamente
- si cercano nuove URL contenute nelle pagine appena estratte, seguendo i *tag* `<A href="…">`

⇒ questo lavoro lo fanno gli *spider*

# fase 1 – Visitare il Web (2)

**Problema:** bisogna stabilire una criterio di visita

- ***breadth first*** (in ampiezza)
- ***depth first*** (in profondità)

**Possibilità:**

- studi hanno dimostrato che una visita in ampiezza del grafo del Web (*breadth first*) porta a recuperare in fretta le pagine "rilevanti"

**Soluzione di UbiCrawler:**

- visita in ampiezza *host* diversi
- visita in profondità sul singolo *host*

# fase 1 - Architettura di UbiCrawler

- UbiCrawler usa *spider* indipendenti permettendo di:
  - tollerare gli errori in caso di malfunzionamento
  - aumentare i processi di *crawl* in esecuzione senza spreco di risorse
- è un'architettura altamente decentralizzata, scalabile ed auto-stabilizzante: struttura ad agenti
- uno o più agenti possono essere in esecuzione su uno o più computer

⇒ teoricamente potremmo avere un crawl su macchine sparse nel mondo

# fase 1 - Trappole per *spider*

**Problema:** uno *spider* può andare in *loop*

- le "trappole" (*spider-traps*) sono insiemi di pagine che ostacolano il processo di *crawl*
- esistono per due ragioni:
  - tentativi di migliorare la propria posizione nei motori di ricerca
  - errori di programmazione nelle pagine che generano loop tramite link dinamici a se stesse

Come non cadere in "trappola"?

- UbiCrawler pone dei limiti alla visita in profondità:
  - un numero massimo di livelli nella gerarchia delle directory
  - un numero massimo di pagine recuperate per host

⇒ il problema dello spam è presente anche nelle pagine HTML!

# fase 1 - L'algoritmo di *crawl*

```
URL = {seme iniziale}
While ( URL ≠  Ø ) {
        url _ select( URL )
        if ( url.isNotValid ) continue
        page _ crawl( url )
        store( page )
        S _ parse( page )    // per estrarre i link
        URL _ URL + S
}
```

Marco Oliva, Matteo Coloberti

# fase 2 - Il grafo del Web

- si può pensare all'insieme dei documenti recuperati come ad un grafo, in cui:
  - i nodi sono gli URL
  - c'è un arco tra il nodo *x* e il nodo *y* sse la pagina che corrisponde all'URL *x* contiene un *link* verso l'URL *y*
- questo grafo è chiamato **grafo del Web**; si tratta di un grafo altamente dinamico, che cambia in continuazione
- è oggetto di studi e ricerche in diversi campi, sia di carattere puramente matematico, sia sotto gli aspetti sociologici e commerciali

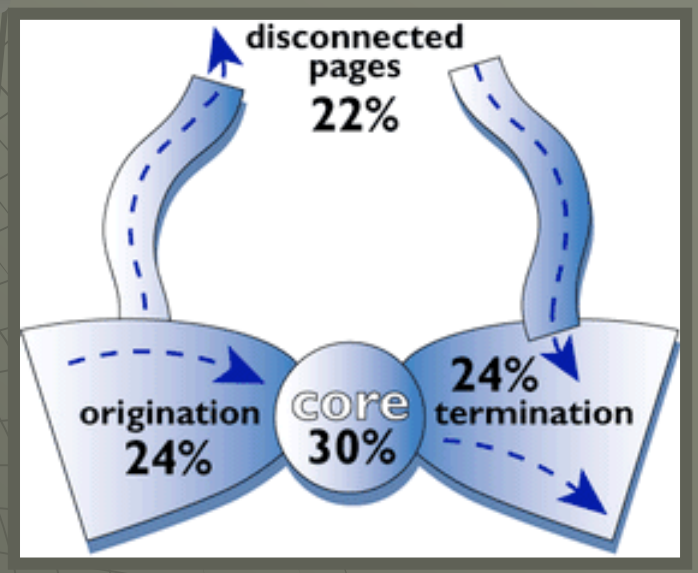
⇒ capire come e perché si formano le comunità può fornire numerosi spunti di interesse

# fase 2 - Caratteristiche del grafo

- il grafo si presenta come un grafo fortemente connesso con due caratteristiche fondamentali:
  - cammino caratteristico breve (distanza media tra due nodi)
  - coefficiente di clustering alto (localmente denso)
- questo modello prende il nome di *small-world*
- nel grafo del Web possiamo identificare una **componente gigante** dove da ogni nodo posso raggiungere ogni altro nodo
- al suo interno presi due nodi a caso, la distanza tra di essi e un numero molto piccolo compreso tra 15 e 20

# fase 2 - La struttura a “cravattino”

- ricerca del 2000 di AltaVista, IBM, Compaq: il Web ha la forma di un cravattino
- ci sono pagine raggiungibili dalle altre (componente gigante)
- alcune hanno solo *link* verso la componente gigante
- altre pagine sono puntate ma prive di *link* esterni
- altre, infine, sono del tutto esterne al “cravattino” (pagine non linkate e senza *link* esterni, etc.)



# fase 2 – Il grafo: a noi è utile?

**A cosa serve il grafo del Web?**

- dato un documento ogni *link* verso un'altra URL può essere visto come un *link* entrante in un altro documento
- il grafo così computato viene utilizzato per assegnare un punteggio alle pagine tramite algoritmi specifici (es. PageRank ma non solo)

# fase 2 - Indicizzare un *corpus*

**Definizione:** *corpus* è un insieme di documenti

**Problema:** dobbiamo indicizzare un *corpus*

**Soluzioni possibili:**

- usare un **indice**: dato un documento, sapere quali parole contiene
- usare un **indice inverso**: data una parola, sapere in quali documenti è contenuta

# fase 2 – Indicizzare un *corpus*

- Es.: interrogazione "Milano" su un *corpus* di 1.000.000 di documenti
  - con l'indice: devo scandire l'**intera lista** dei documenti **(!)**
  - con l'indice inverso: data una parola so già quali documenti la contengono **(!)**

⇒ un motore di ricerca **deve** usare un indice inverso

# fase 2 - Parole, parole, parole...

- Il *corpus* documentale del Web è formato da centinaia di milioni di parole differenti:
  - *hapax legomena*: parole che ricorrono una sola volta, probabili errori di sintassi
  - lunghe sequenze alfanumeriche: codici di prodotti, elenchi telefonici, …
- Cos'è per noi una parola?
  - sequenza alfanumerica con *lunghezza* < *n*, *n* scelto
  - tutte le sequenze più lunghe vengono divise

# fase 2 - Il parsing

**Per indicizzare:**

- il primo passo è riconoscere ed estrarre tutte le parole

**Problema:** le pagine HTML contengono frequentemente errori di sintassi

- da analisi da noi fatte più del 90% delle pagine contiene almeno un errore, alcuni fatali per l'analisi del documento
- alcuni esempi di utilizzo improprio dei *tag* sono:
  - mancata chiusura
  - errato utilizzo dei commenti o *script*
  - errori di battitura

⇒ non si può usare un *parser* di grammatiche di secondo livello

**Soluzione:** serve un *parser* robusto, scritto *ad hoc*

# fase 2 - Cosa indicizzare

Un indice inverso può contenere:

- **frequenza**: quante volte un termine compare nel *corpus*
- **puntatori** (a documenti): in quali documenti compare
- **conteggi**: quante volte compare in ogni documento
- **posizioni**: in che posizioni compare in ogni documento
- **dati globali** come il numero complessivo di occorrenze, la lunghezza media dei documenti, ecc.
- la **lista dei termini**, altrimenti non si lavora!
- può essere privo di *stopword* e *Hapax legomena*

⇒ quali di questi elementi salvare è funzione (anche) degli algoritmi in uso

# fase 2 - Hashing

**Problemi:**

- nonostante gli sforzi fatti per diminuire il numero di parole, queste sono ancora **troppe**
- per gestire efficientemente un insieme così grande di termini, tecniche tradizionali come le tabelle *hash* o gli alberi bilanciati sono inutilizzabili anche su macchine studiate per operazioni *I/O bound* e con diversi GB di RAM

⇒ dobbiamo utilizzare altre tecniche di *hashing*: ***Minimal perfect hash***: dati i termini

$$t_0, t_1, \ldots, t_{n-1}$$

questa funzione mappa ogni termine $t_i$ in $i$

# fase 3 – Recuperare i dati “al volo”

**Ora che abbiamo:**

- uno snapshot del Web
- il grafo dei documenti recuperati
- gli indici inversi delle parole contenute in essi

**… come procediamo?**

# fase 3 - Recupero dati: linee di sviluppo

**Per il recupero dati a fronte di *query*, due linee direttrici di ricerca:**

- ideazione e realizzazione di nuovi algoritmi di *ranking* per il recupero più mirato di informazione
- tecniche per l'aggregazione di risultati e per la valutazione efficiente dei *match*

# fase 3 - Algoritmi di ranking già esistenti

Implementazione di algoritmi già esistenti:
PageRank, Proximity

- PageRank funziona *sul grafo*: più una pagina è puntata, più è rilevante (misura *esogena* della popolarità)
- Proximity funziona *sul testo*: più nella pagina le parole richieste sono vicine, più la pagina è rilevante (misura *endogena* dell'importanza, relativamente alla richiesta)

# fase 3 – Gli algoritmi noti non bastano

**Problema:** PageRank + Proximity non bastano: i risultati sono piuttosto scarsi e deludenti

**Soluzione:** servono (anche) altre tecniche, quali punteggio ai titoli, punteggio alle URL, punteggio al testo con cui le pagine sono riferite (*ancore*)

# fase 3 - Nuovi algoritmi: TitleRank

- si assegna un punteggio ai titoli delle pagine: i titoli sono spesso un "riassunto" del contenuto delle pagine
- il punteggio viene assegnato in maniera dipendente dalla prossimità: più le parole richieste sono vicine nel titolo, più il punteggio della pagina è elevato

# fase 3 - Nuovi algoritmi: URLRank

Cercando il nome di un sito si desidera di solito vedere comparire il dominio associato: va dato un punteggio anche agli indirizzi

"comune di milano" ⇒ www.comunedimilano.it

- si ricercano le parole contenute nelle URL tramite un TST (*ternary search tree*)
- si assegna un punteggio basato sulla prossimità

# fase 3 - Nuovi algoritmi: AnchorRank

Le pagine a volte sono note per qualcosa che non dicono esplicitamente di trattare

"agenzia stampa ansa" ⇒ www.ansa.it

⇒ sono le pagine che vi si riferiscono ad usare queste parole nelle ancore

⇒ bisogna estrarre il testo dalle *ancore* per trovare le pagine corrette

# Un esempio: “miserable failure”

- L’esempio più recente e più noto di “applicazione” di AnchorRank è “miserable failure” (e “miserabile fallimento”)
- Molte pagine puntano ad una unica pagina (che non contiene quelle parole) usando quelle specifiche parole nel testo del *link*

⇒ la pagina puntata “acquisisce” le parole con cui viene puntata

# fase 3 - Aggregazione

**Problema:** come aggregare i punteggi dei vari algoritmi?

**Idea:** generare una combinazione lineare di risultati

**Pregi:**

- è facile effettuare esperimenti variando i coefficienti
- pulizia di progettazione

# fase 3 - Valutazione veloce (1)

**Problema:** cercare **tutte** le pagine che contengono una parola può essere costoso

**Due motivi:**

- la parola è presente in molti documenti (es. “milano”)
- la parola è presente più volte nei documenti (es. “la”)

# fase 3 - Valutazione veloce (2)

**Soluzione:** la valutazione dei *match* (Proximity) deve essere "tagliata" oltre una certa soglia (è meglio se le pagine sono ordinate in maniera decrescente secondo un punteggio statico, ad es. PageRank)

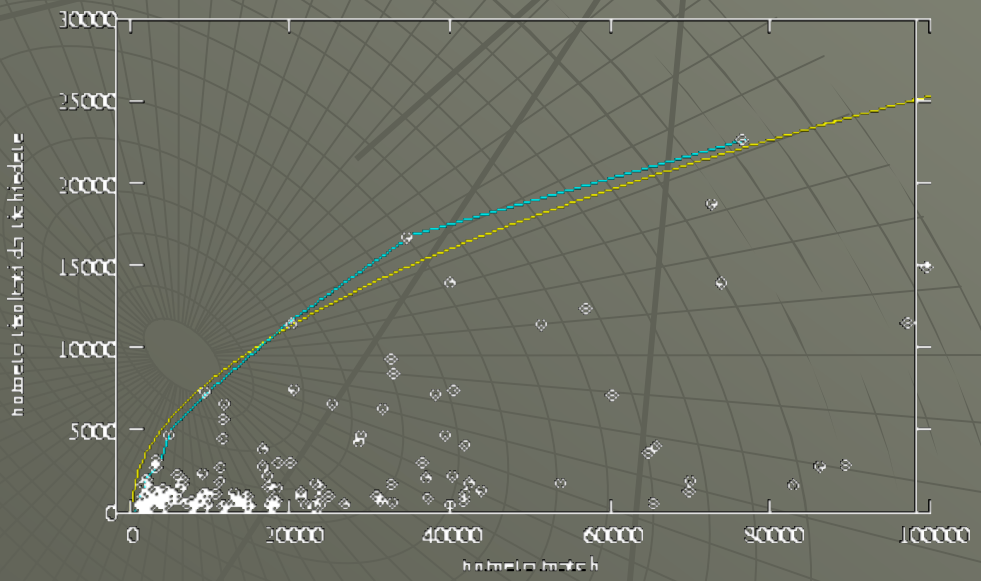
⇒ si usano operatori *lazy* per trovare i match

Ci interessano i primi N risultati con una precisione data: quando tagliare?

⇒ simulazione con *query* fittizie

# fase 3 - Valutazione veloce (3)

Per esempio, se ci interessano i primi 400 risultati di PageRank + Proximity con precisione 95%:




— inviluppo convesso

— approssimazione

# fase 3 - Demo

- 50M+ di pagine web tratte da .it
- 10 giorni per recuperarle
- 4 giorni macchina per indicizzarle

proviamo qualche interrogazione…

# Conclusioni

- partiti da *crawler* sviluppato da DSI/CNR in Java
- sviluppati strumenti per l'indicizzazione
- sviluppati algoritmi per migliorare la ricerca
- sviluppata tecnica per aggregare i risultati restituiti da questi algoritmi
- sviluppato parser HTML non stretto
- sviluppate tecniche di valutazione veloce dei *match*
- implementazione **completa** delle tecniche suddette in un motore di ricerca sperimentale

# Riferimenti

**Marco Olivo** marco@olivo.net

**Matteo Coloberti** matteo@coloberti.it

Grazie per l'attenzione