Capitolo 1

INTRODUZIONE

1.1 I meta-search engines

Il World Wide Web è diventato una grande risorsa di informazioni negli ultimi anni: nel febbraio 1999, c'erano approssimativamente 800 milioni di pagine pubbliche sul web [16], oggi il solo Google ne indicizza più di 4 miliardi. Trovare informazioni è l'uso principale che si fa del web oggi, e per questa ragione sono stati creati negli ultimi anni molti motori di ricerca, con lo scopo di aiutare gli utenti a districarsi tra l'enormità di pagine web esistenti.

Ogni motore di ricerca ha un database testuale definito da un set di documenti che possono essere restituiti dal motore. Di solito le ricerche sono effettuate su un indice di questo database, costruito in precedenza per rendere più veloce l'intero processo di interogazione. Le pagine vengono indicizzare individuando al loro interno una o più *parole chiave* (oggi la maggior parte dei motori di ricerca usa il "full-text indexing", dove ogni termine del documento è incluso nell'indice) che vengono poi associate al documento, ed usate nella ricerca.

Molti dei maggiori motori di ricerca per il web, come per esempio Altavista, Google e Yahoo, cercano di *indicizzare l'intero web* e di fornire la possibilità di ricerca su tutte le pagine che lo compongono. Purtroppo, questi motori centralizzati soffrono di un gran numero di limitazioni [5]. Per esempio, la copertura del web di ognuno di essi è limitata [7, 16] dall'esclusione volontaria dei loro crawler da parte del proprietario del sito, o dalla mancanza dei link ad alcune pagine che restano quindi nascoste. Un altro comune problema di questi motori è che più i loro indici diventano grandi, più una grande percentuale del loro contenuto diventa obsoleta. Per questo motivo ci sono molti dubbi sull'effettiva possibilità di indicizzare

l'intero web utilizzando un solo motore di ricerca, una sola tecnologia, un solo indice.

Sul web però, nascosti dietro l'ombra creata dai grandi motori di ricerca, esistono migliaia di motori "specializzati" che si concentrano sull'indicizzazione di un particolare tipo di documenti. Vi è ad esempio il Cora Search Engine (cora.whizbang.com) che cerca di indicizzare tutti i documenti di carattere scientifico sull'informatica, CiteSeer (citeseer.ist.psu.edu) che si propone come libreria digitale della letteratura scientifica, ed il Medical World Search (www.mwsearch.com) che cerca di indicizzare tutti i documenti di carattere medico. Anche organizzazioni molto piccole hanno spesso i loro personali motori di ricerca, con una indicizzazione dei documenti interni di qualità sicuramente superiore a qualsiasi altro motore di ricerca esterno.
Si può facilmente intuire quindi, come l'utilizzo combinato di tutti questi piccoli motori di ricerca riuscirebbe a "coprire" il web in modo molto migliore di quanto possa invece fare un solo "grande" motore di ricerca.

Questo è l'approccio usato dai *meta-search engines*, sistemi che forniscono un accesso unificato ad altri motori di ricerca locali. Non mantengono un proprio indice delle pagine web, ma sfruttano quello degli altri. Quando un meta-motore di ricerca riceve la richiesta di un utente, la passa (con la formattazione necessaria) agli appropriati motori di ricerca locali, collezionando (e riorganizzando) i risultati restituiti. In questo modo, oltre ad una *migliore copertura del web*, è più facile mantenere l'indice dei documenti aggiornato, dato che ognuno dei motori di ricerca copre solo una piccola porzione del web. Inoltre, un meta-motore di ricerca è anche molto più economico da creare e mantenere, richiedendo molti meno investimenti in hardware (computers, memorie...) rispetto ad un grande motore di ricerca come Google, che usa migliaia di computer.

Una domanda che sorge spontanea quando si parla dei meta-motori di ricerca è "perchè un utente non può semplicemente visitare ognuno dei motori, ed effettuare la ricerca?". Generalmente è una buona domanda poichè cercando ad esempio "Madonna" su Google, AltaVista, Alltheweb o Lycos, si otterranno risultati simili. Non c'è quindi ragione di utilizzare più di un motore di ricerca per questo tipo di interrogazioni. In altri casi però, ci sono molte buone ragioni per le quali si potrebbe voler provare ad utilizzare un meta-motore di ricerca:

- *Copertura del web*: L'unione e la fusione degli indici di più motori di ricerca permetterà di accedere ad una quantità di risultati incredibilmente superiore a quella che si potrebbe ottenere utilizzando un solo motore [5].
- *Comparazione dei risultati*: Non c'è modo più veloce per comparare i tipi di risultati forniti dai vari motori di ricerca, che utilizzare un meta-search engine.

- *Velocità di risposta*: Una ricerca effettuata su un meta-search engine impiega generalmente solo una frazione del tempo che impiegherebbe se fosse l'utente a doversi preoccupare di eseguirla su più di un motore di ricerca.
- *Ricerche particolari*: Talvolta una ricerca effettuata su un solo motore di ricerca può non fornire i risultati attesi, o può addirittura non fornire alcun risultato. Se l'argomento della ricerca è "oscuro", utilizzare un meta-motore di ricerca potrebbe essere la giusta soluzione, nella speranza che almeno un motore di ricerca trovi qualcosa di interessante.
- *Interfacce consistenti*: Imparare ad usare le opzioni di ricerca avanzata messe a disposizione dai vari motori di ricerca può essere lungo e difficile. Molti meta-motori di ricerca hanno le loro funzioni di ricerca avanzata: questo significa imparare un solo set di comandi, ed utilizzarlo in ogni tipo di ricerca.

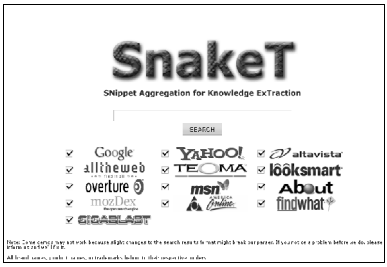
Ci sono comunque vari *problemi* da affrontare per implementare un efficente ed efficace meta-motore di ricerca. Uno dei principali, è la *selezione dei database*, nella quale si cerca di scegliere i motori di ricerca locali che più probabilmente conterranno informazioni utili per la ricerca. L'obiettivo di questa selezione è ridurre, per quanto possibile, i problemi derivanti dal traffico sulla rete ed il costo di ricerche fatte su database non utili. Per fare ciò i meta-motori mantengono localmente alcuni dei termini appartenenti ai motori di ricerca che utilizzano. Questi termini gli sono utili per caratterizzarne i contenuti, e valutarne l'utilità in rapporto ai termini della ricerca.

Un altro dei principali problemi, con cui si scontra chi intende implementare un meta-motore di ricerca, è il problema della *fusione dei risultati* che si ottengono dai vari motori di ricerca interrogati. L'obiettivo in questo caso è cercare di ordinare i risultati in modo che quelli più rilevanti si trovino davanti a quelli meno importanti. E' un compito difficile, che l'eterogeneità dei contenuti del web rende ancora più arduo da risolvere. In definitiva, un buon meta-motore di ricerca dovrebbe anche avere *performance* simili a quelle che si otterrebbero se tutti i documenti fossero indicizzati su un singolo database, minimizzando quindi il costo degli accessi.

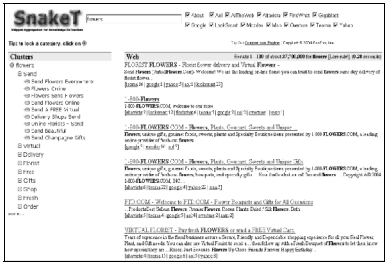
1.2 Soggetto del tirocinio

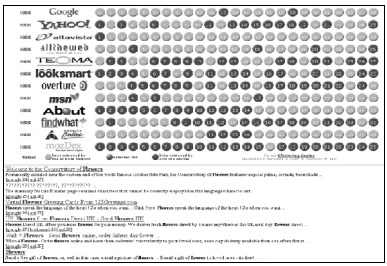
Il soggetto del mio tirocinio era proprio incentrato su quest'ultimo argomento: sviluppare un software utilizzando il linguaggio C in ambiente UNIX/Linux, in grado di interrogare simultaneamente più motori di ricerca, per minimizzare i singoli tempi di attesa e restituire i risultati ottenuti nel minor tempo possibile, con un'apposita formattazione XML.

Questo software si integra all'interno di due progetti di ricerca dell'Università di Pisa, seguiti dal professor Paolo Ferragina e da Antonio Gulli.



Il primo progetto "*SnakeT – SNippet Aggregation for knowledge ExTraction*", prevede la costruzione di un meta-motore di ricerca con clusterizzazione dei risultati.
Gli snippets recuperati dai motori di ricerca vengono raggruppati in "clusters" ed organizzati a video sottoforma di albero. Le foglie dell'albero sono associate a dei termini che ne caratterizzano il tema, ed aiutano l'utente fornendo "un'immagine" della sua ricerca a vari livelli di dettaglio. Lo "stato dell'arte" in questo campo, è rappresentato dal meta-motore Vivisimo (www.vivisimo.com).





Il secondo progetto di ricerca, in cui il mio software va ad integrarsi, è quello dal titolo "*Comparison Engine: find you own rank position on many engines*".
Il progetto prevede l'implementazione di un meta-motore di ricerca che permetta all'utente un'agevole comparazione delle risposte fornite da più motori di ricerca, attraverso una semplice interfaccia grafica. Utilizzando questo meta-motore è facile stupirsi di come il link più rilevante per Google, ad esempio, possa non

esserlo per Yahoo, dimostrando come i meccanismi di ranking utilizzati dai vari motori, siano molto diversi tra di loro.

Il titolo del tirocinio "*Interrogazione di search-engines mediante architettura multi-thread*" suggeriva l'utilizzo di una soluzione parallela basata su più thread. Nelle analisi condotte durante la progettazione del software però, alcune prove pratiche hanno dimostrato come l'utilizzo di una soluzione multi-thread generasse un grosso overhead dovuto alla creazione ed alla gestione di un elevato numero di thread. Utilizzando ad esempio "solo" 15 motori simultaneamente, per ogni ricerca sarebbe stato necessario creare, gestire e distruggere ben 16 thread! Poichè il programma deve essere integrato in due progetti con interfaccia pubblica accessibile dal web, questa caratteristica negativa avrebbe indotto una saturazione del sistema ospite con centinaia di richieste di thread, già con poche ricerche simultanee.

Si è preferito quindi utilizzare le funzionalità di *I/O Async* fornite dal sistema operativo per ottenere la parallelizzazione delle connessioni ai vari motori di ricerca. In particolare, l'uso della funzione `select()` ha permesso di gestire simultaneamente un elevato numero di socket senza sovraccaricare il sistema con centinaia di thread. La gestione dei socket è stata poi facilitata dalla possibilità (secondo lo standard POSIX) di utilizzare le funzioni di sistema `read()` e `write()` per ricevere ed inviare i dati sul canale, permettendo di scrivere un codice chiaro, elegante, e portabile. Maggiori informazioni al riguardo potranno essere trovate nel *Capitolo 5*, o direttamente tra i commenti nel codice del programma.

Le *performance* che abbiamo ottenuto con l'impiego di una soluzione parallela sono molto interessanti: *600 risultati in appena 4.5secs* (richiedendo 100 risultati ad ognuno dei 6 motori di ricerca interrogati).

Come si può vedere nel grafico (1.1) questo tempo equivale a circa il 30% di quello che avrebbe richiesto un'interrogazione seriale ed è paragonabile, e molto influenzato, da quello richiesto dal solo motore LookSmart, per fornirci i suoi 105 risultati (non utilizzandolo si ottengono *500 risultati in 1.9secs*).

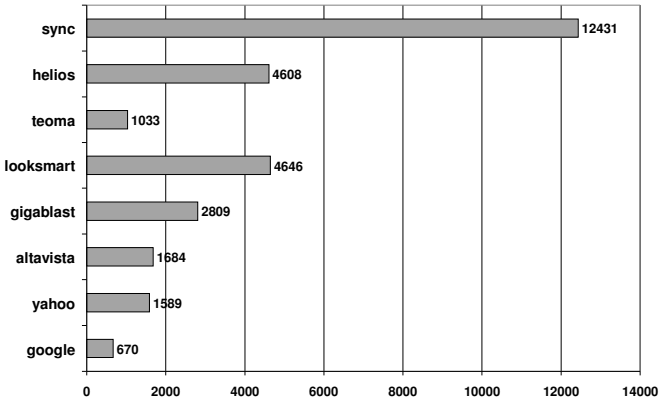


Grafico 1.1 – Tempi di completamento di una richiesta (100 ris.)

Utilizzando nella ricerca tutti i 16 motori attualmente disponibili nel programma, la soluzione parallela sviluppata consente di ottenere ben *1355 risultati in 9 secondi*, ovvero circa 150 risultati al secondo. Queste performance risultano paragonabili, se non addirittura superiori, a quelle ottenibili eseguendo una ricerca con il velocissimo Google, che è considerato "lo stato dell'arte" tra i motori di ricerca.

Nella valutazione delle performance, bisogna anche considerare che questo software, a differenza di quello utilizzato da altri meta-motori di ricerca, non utilizza "backdoor" o accessi privilegiati ai database dei vari motori che interroga, ma simula semplicemente il comportamento di un qualsiasi utente che svolge una ricerca sul web. Questo aspetto assume ancora più valore se si pensa che i risultati restituiti dai vari motori, formattati in HTML e con gli stili più vari, vengono anche analizzati dal software, che ne estrae titoli, URLs e snippets.

Il tirocinio inoltre prevedeva lo sviluppo di un *parser veloce ed espandibile* per l'analisi e l'estrazione delle informazioni utili dalle pagine di risposta dei vari motori. Il parser doveva anche essere configurabile tramite file esterni, in modo da non dover intervenire sul codice del programma ogni volta che uno dei motori cambiava il logo, o lo stile delle sue pagine. Molte erano le soluzioni possibili: i generatori di parser, le espressioni regolari, la costruzione dell'albero dell'HTML, la ricerca di un qualche tipo di "marker" per contraddistinguere l'inizio e la fine delle informazioni utili. Nessuna di queste però soddisfaceva, entrambi i requisiti di velocità ed espandibilità che ci eravamo prefissati. I *generatori di parser*, potenti ed efficenti, generavano codice da implementare direttamente nel programma e non erano quindi utilizzabili per il nostro scopo. L'implementazione delle funzioni necessarie al riconoscimento delle *espressioni regolari* sarebbe stato difficile e forse inefficente; ancora più difficile poi, sarebbe stato il riuscire a scrivere

espressioni regolari sempre corrette, che non generino “falsi match”. La costruzione di un *albero dell’HTML* avrebbe portato con se un’inutile overhead per la sua generazione, con liste, puntatori, ed allocazioni di nuova memoria. Infine, l’utilizzo dei *marker* avrebbe risolto il problema in maniera molto precaria, ed un semplice cambio di font, lo avrebbe messo in crisi.

In questo tirocinio è stata quindi scelta una soluzione ibrida, con lo sviluppo di un *piccolo linguaggio di programmazione* del parser. Al momento dell’analisi del buffer di ricezione, il parser decodifica l’istruzione seguente e la esegue attraverso funzioni di sistema, avanzando all’istruzione successiva in caso di esito positivo. Il linguaggio, che ha una sua precisa struttura e sintassi, supporta attualmente una buona varietà di comandi: dai più semplici, come quelli per spostare il cursore di un numero fissato di posizioni o per ricercare una stringa nel testo, ai più complessi, come *if*, e *jump* condizionali. Maggior informazioni sul parser potranno essere trovate nel *Capitolo 4*.

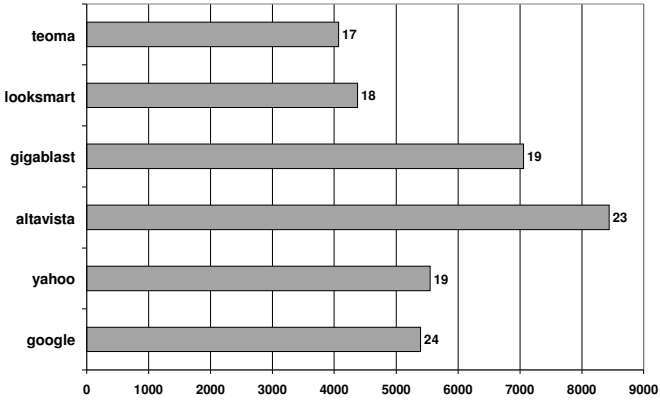


Grafico 1.2 – Rapporto tra linee di codice (a destra) e costo del parsing (in μs)

Questa soluzione ha dimostrato di essere molto buona, sia dal punto di vista dell’espandibilità, che dal punto di vista delle performance, dato che l’analisi e l’estrazione dei *1355 risultati* dell’esempio precedente richiede *circa 100 millisecondi*. Il grafico seguente (1.2) mostra il rapporto tra le linee del codice del parser (a destra) dei vari motori, ed il tempo impiegato (in microsecondi) per l’analisi e l’estrazione di 100 risultati.

1.3 Letteratura accademica

Anche se il progetto di tirocinio non riguardava la selezione dei database da interrogare ed i metodi di fusione dei risultati, per completezza è stata analizzata la letteratura scientifica corrispondente.

Nella *selezione dei database* da interrogare, la maggioranza dei meta-motori ordina preventivamente i motori di ricerca, valutandone l'utilità in rapporto alla query ricevuta, ed eseguendo la ricerca solamente su quelli che compaiono nelle prime *n* posizioni della graduatoria. Per esempio, gGlOSS ordina i motori di ricerca in base al numero di documenti rilevanti in essi contenuti [3], CORI Net calcola per ogni motore "la probabilità" che contenga documenti rilevanti per la ricerca [1], D-WISE ordina i motori calcolando per ognuno di essi "la somma delle frequenze pesate dei termini della ricerca", che appaiono nei documenti indicizzati [13], Q-Pilot fa delle statistiche sulle similitudini tra la query espansa e la descrizione del motore presente nel suo database [14]. Tutti questi metodi sono però euristici e non progettati secondo un qualche criterio di ottimalità. In [11, 12] la misura usata per ordinare i database è la "similarità". E' dimostrato che ordinare i motori di ricerca in modo decrescente rispetto alla similarità tra la query ricevuta e la descrizione statistica del loro database, è una condizione sufficente e necessaria per ordinarli in maniera ottimale. Un'altra condizione necessaria e sufficente per un ottimale ordinamento dei database è data anche in [6]. Questo lavoro però, considera solamente database e ricerche su dati strutturati, mentre [11, 12] considerano anche dati non strutturati.

Per la *fusione dei risultati*, le più recenti soluzioni al problema usano un tipo di approccio definito "allocazione pesata", ovvero, richiedono proporzionalmente più documenti ai motori di ricerca che dopo l'ordinamento hanno una posizione più alta (es. Cori Net, D-WISE, ProFusion [2] e MRDD [10]), ed usano poi le similarità tra i documenti ricevuti per fonderli insieme (D-WISE e ProFusion). Anche questi approcci sono però euristici, ed anche se generalmente forniscono ottimi risultati, non garantiscono di recuperare tutti i documenti potenzialmente utili per una ricerca. In [4, 15], per determinare quali documenti recuperare da un database locale, sono proposti metodi che cercano di determinare una soglia di "similarità locale" tra i documenti indicizzati dal motore e la confrontano con la soglia di "similarità globale". Questi approcci hanno lo scopo di recuperare tutti i risultati potenzialmente utili da un database, minimizzando il numero di dati inutili recuperati. Il problema con questo tipo di soluzione è che occorrere conoscere il tipo di algoritmo usato per il ranking e l'indicizzazione, di ogni motore di ricerca interrogato. Compito difficile, poichè gli algoritmi sono in genere proprietari. Il meta-motore di ricerca Inquirius [8], usa veramente tecniche di confronto sulla similarità globale dei documenti, per fonderli tra di loro: il vantaggio è l'alta qualità della fusione che si ottiene, mentre lo svantaggio, è che i documenti devono essere effettivamente recuperati anche dal meta-motore di ricerca, che ne ha bisogno per poter riuscire a calcolare i valori statistici sulla similarità globale. Il metodo usato in [11, 12] utilizza invece un'ordinamento ottimale (stimato) per stabilire quali documenti recuperare, ed usa poi i criteri di similarità globale (reali), per fondere tra loro i risultati.

Molti dei meta-motori di ricerca esistenti oggi interrogano solo pochi motori di ricerca locali. E' quindi molto difficile pensare che possano essere facilmente adattati ad interrogazioni parallele su migliaia di motori di ricerca, conservando una buona efficenza. Le ragioni sono varie, per prima cosa, i metodi utilizzati attualmente confrontano i termini della ricerca con tutte le "descrizioni statistiche" dei motori di ricerca disponibili, per selezionare quelli da interrogare. Questo compito è computazionalmente molto costoso se viene effettuato su un grande numero di database. In secondo luogo, basandosi sui metodi esistenti [1, 3, 9, 21, 13] per fare un'accurata selezione dei motori di ricerca da interrogare, è necessario che le descrizioni statistiche sul contenuto di ognuno di essi siano molto dettagliate. Maggior dettaglio è nell'informatica sinonimo di "maggiore quantità di informazioni", ovvero di spazio: una dettagliata descrizione del contenuto del database di un motore di ricerca, può occupare anche l'1% della dimensione del database originale. L'ovvia conseguenza è quella di far assumere al database del meta-motore di ricerca dimensioni considerevoli (anche 10 volte superiori a quelle di un database medio) obbligando l'utilizzo di grandi (e lente) memoria di massa, che introducono ritardi nella computazione, e rallentano l'intero sistema.

1.4 Bibliografia

[1] J. Callan, Z. Lu, and W. Croft. Searching Distributed Collections with Inference Networks. ACM SIGIR, 1995.

[2] Y. Fan, and S. Gauch. Adaptive Agents for Information Gathering from Multiple, Distributed Information Sources. 1999 AAAI Symposium on Intelligent Agents in Cyberspace.

[3] L. Gravano, and H. Garcia-Molina. Generalizing GlOSS to Vector-Space Databases and Broker Hierarchies. VLDB, 1995.

[4] L. Gravano, and H. Garcia-Molina. Merging Ranks from Heterogeneous Internet Sources. VLDB, 1997.

[5] D. Hawking, and P. Thistlewaite. Methods for Information Server Selection. ACM Transactions on Information Systems, 17(1), January 1999.

[6] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments. 1995.

[7] S. Lawrence, and C. Lee Giles. Searching the World Wide Web. Science, 280, April 1998.

[8] S. Lawrence, and C. Lee Giles. Inquirus, the NECi Meta Search Engine. Seventh International World Wide Web Conference, 1998.

[9] K. Liu, C. Yu, W. Meng, W. Wu, and N. Rishe. A Statistical Method for Estimating the Usefulness of Text Databases. IEEE TKDE (to appear).

[10] E. Voorhees, N. Gupta, and B. Johnson-Laird. The Collection Fusion Problem. TREC-3, 1995.

[11] C. Yu, K. Liu, W. Meng, Z. Wu, and N. Rishe. A Methodology for Retrieving Text Documents from Multiple Databases. IEEE Transactions on Knowledge and Data Engineering (to appear).

[12] C. Yu, W. Meng, K. Liu, W. Wu, and N. Rishe. Efficient and Effective Metasearch for a Large Number of Text Databases. CIKM'99, 1999.

[13] B. Yuwono, and D. Lee. Server Ranking for Distributed Text Resource Systems on the Internet. DASFAA'97, 1997.

[14] A. Sugiura, and O. Etzioni. Query Routing for Web Search Engines: Architecture and Experiments. WWW9 Conference, 2000.

[15] W. Meng, K. Liu, C. Yu, X. Wang, Y. Chang, N. Rishe. Determine Text Databases to Search in the Internet. VLDB, 1998.

[16] S. Lawrence, and C. Lee Giles. Accessibility of Information on the Web. Nature, 400, July 1999

Capitolo 2

SEARCH ENGINES

Benchè il progetto preveda un alto grado di espandibilità per quanto riguarda i motori di ricerca utilizzati è stato necessario, nella fase di analisi, sceglierne alcuni, in modo da ottenere statistiche sui tempi di risposta, sui risultati restituiti, e permettere lo sviluppo delle funzioni relative alla loro interrogazione.

Tutti i test sono stati effettuati dalla macchina *Roquefort* dell'Università di Pisa, gentilmente messa a mia disposizione dal gruppo di Algoritmisti.

2.1 I test eseguiti

Nella schede seguenti sono riportati i *dati statistici* acquisiti su ogni motore di ricerca, quando disponibili inoltre, sono riportati anche *dati storici*, come la data di fondazione del motore di ricerca, e *dati informativi*, come il numero di richieste giornaliere soddisfatte, o il numero di pagine indicizzate.

Il *tempo medio di PING*, da un'informazione generale sulla "qualità della strada" percorsa dai pacchetti per arrivare dalla macchina di test ai server del motore di ricerca. Non è una stima precisa perchè influenzata dalle momentanee condizioni della rete (sia interna che esterna), ma resta comunque un buon indicatore delle performance ottenibili.

Il *tempo medio di connessione*, è il tempo che occorre attendere dopo l'invio della richiesta di connessione, per ottenere un collegamento con il server ed essere in grado di inviargli dei dati.

Il *tempo medio prima risposta*, specifica quanti secondi occorrono dopo l'invio della richiesta HTTP per cominciare a ricevere i dati richiesti. Poichè il software

lavora su più motori in parallelo, questo è un dato molto importante per le nostre valutazioni.

Il *tempo medio per 10/100 risultati*, rappresenta il tempo necessario a completare una ricerca sul motore. Comprende: connessione, richiesta HTTP, ricezione dei risultati, e disconnessione.

Per ogni motore sono state eseguiti 50 rilevamenti diversi per ogni categoria di test. Le prove sono state eseguite singolarmente, e sono state ripetute in diversi momenti della giornata (mattina, sera e notte) al fine di ottenere indicatori realistici, non influenzati da picchi di afflusso massimo o minimo.

Nella scheda di ogni motore sarà riportata la *query standard* usata per le sue interrogazioni. Gli eventuali parametri accessori, come la selezione del solo materiale HTML o la disabilitazione dei filtri per certi contenuti possono essere trovati all'interno del file `engines.dat`. La sintassi usata per identificare i vari parametri della query è questa:

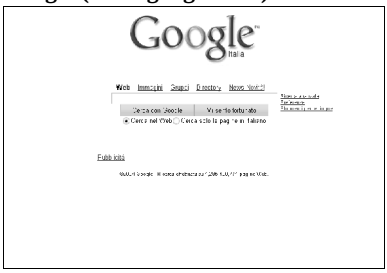
<*Q*>	stringa da ricercare
<*R*>	risultati per pagina
<*P*>	numero di pagina da recuperare

Per ogni motore saranno anche indicati: il numero di *risultati per pagina* utilizzabili, che si può generalmente trovare nella pagina riservata alla ricerca avanzata, ed il *tipo di algoritmo* usato per la selezione della pagina da recuperare. A riguardo esistono 4 algoritmi principali:

Tipo A : 0, 1, 2, 3, ...
Tipo B : 1, 2, 3, 4, ...
Tipo C : risPag*1, risPag*2, risPag*3, risPag*4, ...
Tipo D : (risPag*1)+1, (risPag*2)+1, (risPag*3)+1, ...

Informazioni più dettagliate su questi algoritmi si troveranno tra i commenti del codice.

2.2 Google (www.google.com)



Google, è l'attuale "principe" dei motori di ricerca: con i suoi 4.285 milioni di pagine indicizzate e gli ottimi meccanismi di page-ranking.

Fondato nel settembre del 1998 da Larry Page e Sergey Brin, oggi soddisfa più di 200 milioni di richieste al giorno.

```
Tempo medio di PING           :       44.791ms
Tempo medio di connessione    :      115.284ms
Tempo medio prima risposta    :      421.561ms
Tempo medio per 100 ris.      :      737.389ms

Query standard:
   /search?q=<Q>&num=<R>&start=<P>
```

Google accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 20, 50, 100), ed un sistema di scelta della pagina da visualizzare di tipo B.

2.3 Yahoo (www.yahoo.com)

Forse uno dei più vecchi motori di ricerca su internet, Yahoo, indicizza le pagine manualmente suddividendole in categorie (directory).

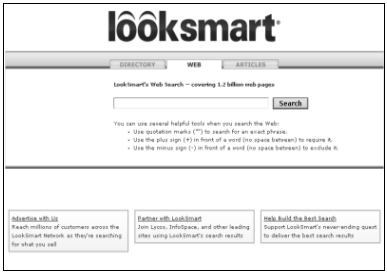
Fondato nel 1994 da David Filo e Jerry Yang, oggi ha più di 537 milioni di pagine indicizzate.

```
Tempo medio di PING           :       111.451ms
Tempo medio di connessione    :       683.930ms
Tempo medio prima risposta    :       860.771ms
Tempo medio per 100 ris.      :      1799.539ms

Query standard:
    /search?va=<Q>&n=<R>&b=<P>
```

Yahoo accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 15, 20, 30, 40, 50, 100), ed un sistema di scelta della pagina da visualizzare di tipo B.

2.4 LookSmart (www.looksmart.com)



LookSmart è uno dei motori di ricerca più “giovani” e promettenti del web.
Ha un approccio ibrido per l’indicizzazione delle pagine: che avviene in modo semi-automatico, conla loro divisione in categorie di interesse.

Fondato nel 1999, oggi indicizza più di 1.100 milioni di documenti.

```
Tempo medio di PING           :       188.979ms
Tempo medio di connessione    :       440.784ms
Tempo medio prima risposta    :       633.254ms
Tempo medio per 105 ris.      :      5742.653ms

Query standard:
    /p/search?qt=<Q>&tb=web&sn<P>&se=0,1000
```

LookSmart accetta query semplici o esatte, con un numero di risultati per pagina fissato a 15, ed un sistema di scelta della pagina da visualizzare di tipo B.

2.5 Altavista (www.altavista.com)

Altavista è stato il primo motore di ricerca per il World Wide Web, e conta attualmente 61 brevetti di tecnologie di indexing e search.

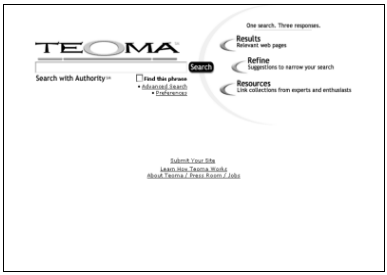
Fondato nel 1995 da alcuni componenti del Digital Equipment Corporation's Research lab, è stato recentemente acquisito da Yahoo.

```
Tempo medio di PING           :       110.561ms
Tempo medio di connessione    :       464.708ms
Tempo medio prima risposta    :       624.297ms
Tempo medio per 100 ris.      :      1707.629ms

Query standard:
    /web/results?aqa=<Q>&nbq=<R>&stq=<P>
```

Altavista accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 20, 30, 40, 50), ed un sistema di scelta della pagina da visualizzare di tipo B.

2.6 Teoma (www.teoma.com)



Teoma è un altro dei motori di ricerca più giovani del web. Si distingue per l'ottima "rilevanza" dei risultati restituiti, che i suoi autori attribuiscono alla sua "naturale capacità di comprendere il web".

Fondato nel 2000 da un gruppo di informatici della Rutgers University, oggi indicizza più di 500 milioni di pagine web.

```
Tempo medio di PING          :       120.228ms
Tempo medio di connessione   :       474.369ms
Tempo medio prima risposta   :       676.277ms
Tempo medio per 100 ris.     :      1717.539ms

Query standard:
    /search?q=<Q>&u=<R>&page=<P>
```

Teoma accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 20, 30, 50, 100), ed un sistema di scelta della pagina da visualizzare di tipo A.

2.7 Gigablast (www.gigablast.com)

Nonostante l'aspetto poco curato e l'uso di "soli" 8 server, GigaBlast stà attirando sempre più l'interesse del "popolo di internet".

Fondato nel 1995 da Matt Well, che lo ha scritto interamente in C++ e con il solo uso di librerie standard, GigaBlast conta oggi più di 300 milioni di pagine nel suo indice.

```
Tempo medio di PING           :       163.384ms
Tempo medio di connessione    :       332.096ms
Tempo medio prima risposta    :       485.226ms
Tempo medio per 100 ris.      :      3295.156ms

Query standard:
     /search?s=<P>&q=<Q>&n=<R>
```

GigaBlast accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 20, 30, 40, 50), ed un sistema di scelta della pagina da visualizzare di tipo B.

2.8 Gli altri motori di ricerca

Il software attualmente supporta altri 9 motori di ricerca:

About (www.about.com)
Fondato nel 1997 da Scott Kurnit, è un "search engine tematico" che grazie all'aiuto di esperti dei vari settori (dall'arte ai viaggi), offre circa 475 canali tematici che guidano l'utente in ricerche mirate sull'argomento di suo interesse, offrendo interessanti spunti e link.

```
Tempo medio di PING           :       213.573ms
Tempo medio per 10 ris.       :       742.681ms
Tempo medio per 100 ris.      :      7981.556ms

Query standard:
     /fullsearch.htm?terms=<Q>&pg=<P>
```

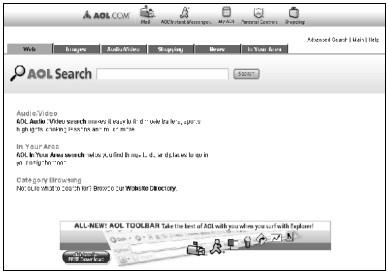
About accetta query semplici o esatte, con un numero di risultati per pagina fissato a 10, ed un sistema di scelta della pagina da visualizzare di tipo B.

Alltheweb (www.alltheweb.com)
Parte del gruppo Overture, ed indicizzato utilizzando Yahoo, questo motore ha un buon numero di "fedeli" che ne esaltano la pulizia delle pagine di risposta e la velocità con cui vengono restituite.

```
Tempo medio di PING           :       115.173ms
Tempo medio per 10 ris.       :       662.750ms
Tempo medio per 100 ris.      :      1019.181ms

Query standard:
    /search?advanced=1&q=<Q>&hits=<R>&o=<P>
```

Alltheweb accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 25, 50, 75, 100), ed un sistema di scelta della pagina da visualizzare di tipo C.



AOL Search (search.aol.com)
Il provider di riferimento per l'America ha introdotto da qualche anno un motore di ricerca nel suo portale. Anche se dalle capacità limitate, l'integrazione del motore con il software di *America On Line*, e la sua popolarità, ne hanno fatto un punto di riferimento per molti internauti.

```
Tempo medio di PING           :       251.778ms
Tempo medio per 10 ris.       :       791.230ms
Tempo medio per 100 ris.      :      9970.974ms

Query standard:
    /aolcom/search?query=<Q>&page=<P>&Stage=0
```

AOL Search accetta query semplici o esatte, con un numero di risultati per pagina fisso (10 per la prima pagina, 15 per le seguenti), ed un sistema di scelta della pagina da visualizzare di tipo B.

Findwhat (www.findwhat.com)
Fondato nel 1998, è un motore di ricerca "guidato dal commercio", come recita la pagina *About Us*. Le pagine vengono inserite nell'indice solo in seguito al pagamento di una quota, e la compagnia provvede poi a pubblicizzarne il link.

```
Tempo medio di PING           :       201.118ms
Tempo medio per 10 ris.       :       999.131ms
Tempo medio per 100 ris.      :      1935.316ms

Query standard:
    /results.asp?DC=<R>&Base=<P>&MT=<Q>
```

Findwhat accetta query semplici o esatte, con un numero di risultati per pagina variabile, ed un sistema di scelta della pagina da visualizzare di tipo C.

MozDex (www.mozdex.com)
Fondato nel 2003 dalla *Small Production*, mozDex è un motore di ricerca dal codice sorgente "open", che non utilizza algoritmi proprietari per l'indexing ed il ranking delle pagine, provando all'utente la completa "trasparenza" delle sue risposte.

```
Tempo medio di PING           :       183.009ms
Tempo medio per 10 ris.       :       847.011ms
Tempo medio per 100 ris.      :      2020.532ms

Query standard:
    /search.jsp?query=<Q>&start=<P>&hitsPerPage=<R>
```

mozDex accetta query semplici o esatte, con un numero di risultati per pagina variabile, ed un sistema di scelta della pagina da visualizzare di tipo C.

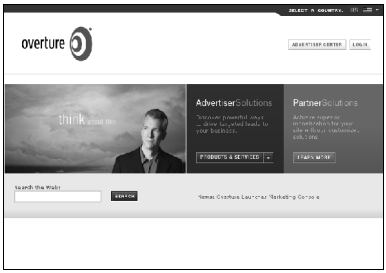
Msn (www.msn.com)

Sette anni fa il colosso *Microsoft* ha fatto il suo ingresso nel mondo dei portali internet con Msn. Utilizzato da milioni di persone, anche solo per ottenere le previsioni meteo o quelle del traffico stradale, soddisfa circa 15 milioni di richieste al giorno.

```
Tempo medio di PING            :       231.967ms
Tempo medio per 10 ris.        :       563.818ms
Tempo medio per 100 ris.       :      3591.208ms

Query standard:
    /pass/advresults.aspx?ps=ba=<P>&q=<Q>&ck_sc=1&ck_af=0
```

Msn accetta query semplici o esatte, con un numero di risultati per pagina fisso a 15, ed un sistema di scelta della pagina da visualizzare di tipo C.



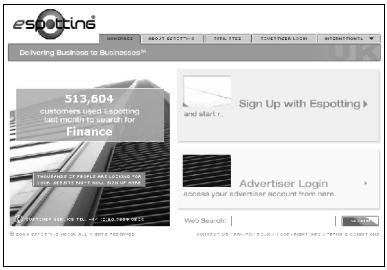
Overture (www.overture.com)

Fondato nel 1997 dal Bill Gross' Idealab è un motore di ricerca "*a pagamento*". Con più di 100.000 link sponsorizzati si propone come collegamento tra le esigenze degli utenti ed il bisogno di pubblicità dei fornitori.

```
Tempo medio di PING            :       107.061ms
Tempo medio per 40 ris.        :      1124.018ms
Tempo medio per 120 ris.       :      3506.665ms

Query standard:
    /d/search/?Keywords=<Q>
```

Overture accetta query semplici o esatte, con un numero di risultati per pagina fisso a 40, ed un sistema di scelta della pagina da visualizzare criptato.



eSpotting (www.espotting.com)
Fondato nel febbraio 2000, questo motore si autodefinisce “business to business”. I siti vengono indicizzati solo con il pagamento di una quota annuale, ma il motore risulta comunque molto usato dagli utenti: circa 1.4 milioni di richieste al mese.

```
Tempo medio di PING            :       134.611ms
Tempo medio per 10 ris.        :       321.780ms
Tempo medio per 100 ris.       :      1128.627ms

Query standard:
    /search/results.asp?keyword<Q>&hits=<R>&page=<P>
```

eSpotting accetta query semplici o esatte, con un numero di risultati per pagina variabile (10, 15, 25, 50), ed un sistema di scelta della pagina da visualizzare di tipo B.

A9 (www.a9.com)
Nato come “branchia” del famoso negozio online *Amazon.com* nell’Ottobre 2003, questo motore di ricerca (associato a Google), permette di eseguire sia le normali ricerche sul web, che quelle all’interno degli “estratti” dei libri che Amazon ha in catalogo.

```
Tempo medio di PING            :       214.728ms
Tempo medio per 10 ris.        :       411.052ms
Tempo medio per 100 ris.       :      1699.156ms

Query standard:
    /<Q>?p=<P>
```

A9 accetta query semplici o esatte, con un numero di risultati per pagina fissato a 10, ed un sistema di scelta della pagina da visualizzare di tipo B.

2.9 Altre informazioni

Molte altre informazioni sui motori di ricerca possono essere trovate direttamente sul web. La guida più autorevole in questo campo è sicuramente SearchEngineWatch.com che propone articoli, recensioni, test e comparazioni tra i maggiori motori di ricerca esistenti oggi. Tra i vari articoli spicca quello [13] scritto da Danny Sullivan nell'Aprile 2004 dal titolo "*Who Powers Whom*", che mostra l'influenza tra gli indici dei vari motori di ricerca, derivata da accordi e partnership varie. La tabella (2.1) riportata di seguito è tratta dal suo articolo, e rende la situazione molto chiara.

Search Engine	Provider: Google	Provider: Yahoo/Overture	Note
Google	Main & Paid		Open Directory an option
Yahoo		Main & Paid	
Msn		Main & Paid (12/05 & 6/05)	LookSmart, as an option on home page
AOL	Main & Paid (est. 10/05+)		AOL-owned Open Directory an option
Ask Jeeves	Paid (9/05)		Main from Ask-owned Teoma. Paid can end as early as 9/04
InfoSpace	Runs several meta search engines. Dogpile is most popular, representative of others. Google (2006), Yahoo (3/06), many small providers have distribution deals.		
Lycos	Paid	Backup	Main from LookSmart; Open Directory as an option
AltaVista		Main & Paid	Open Directory as an option; owned by Yahoo
AllTheWeb		Main & Paid	Owned by Yahoo
HotBot	Paid	Main	Backup from Google & Ask; Owned by Lycos
Netscape	Main & Paid (est. 10/05+)		Owned by AOL; Open Directory as an option
Teoma	Paid (Sept 05)		Main from Teoma Owned by Ask; Paid can end as early as 9/04

LookSmart	LookSmart provides its own Main & Paid

Tabella 2.1 – Who Powers Whom by Danny Sullivan

Un'altro articolo [14] molto interessante scritto dallo stesso autore è datato Settembre 2003, ed ha come titolo "*Search Engine Sizes*". Tra i vari grafici riportati nell'articolo vi è quello mostra la "crescita" degli indici dei motori di ricerca (figura 2.2). E' interessante notare come l'inizio della "guerra" tra i vari motori di ricerca (*GG*=Google, *ATW*=AllTheWeb, *INK*=Inktomi, *TMA*=Teoma, *AV*=AltaVista) sia datato *Gennaio 2001*: prima di tale data, tutti motori di ricerca avevano indici di dimensioni simili.

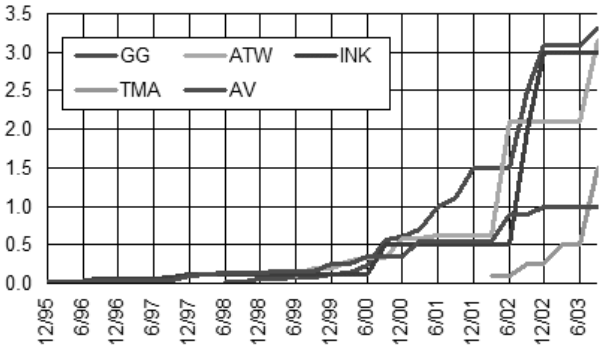


Figura 2.2 – Miliardi di documenti indicizzati (Dec'95/Sep'03)

L'ultimo grafico (2.3) che mi sembra importante riportare è preso dall'articolo "*comScore Media Metrix Search Engine Ratings*" dell'Aprile 2004, che mostra le preferenze dei navigatori americani nell'uso dei vari motori di ricerca. Come era facile immaginarsi Google e Yahoo risultano i più utilizzati.

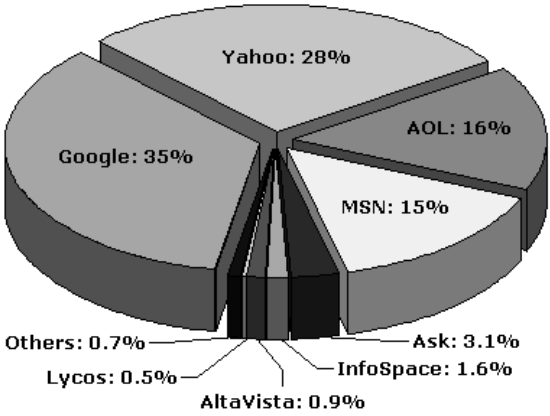


Grafico 2.3 – Utilizzo dei motori di ricerca

2.10 – Tabella riassuntiva

La tabella (2.4) seguente riassume i dati raccolti sui motori di ricerca attualmente implementati nel software.

Motore	*Ping*	*Connessione*	*Primo risultato*	*100 risultati*	*Pagine richiedibili*	*Algoritmo*
Google	44.791	115.284	421.561	737.389	10, 20, 50, 100	B
Yahoo	111.451	683.930	860.771	1799.539	10, 20, 30, 40, 50, 100	B
LookSmart	188.979	440.784	633.254	5742.653	15	B
Altavista	110.561	464.708	624.297	1707.629	10, 20, 30, 40, 50	B
Teoma	120.228	474.369	676.277	171.539	10, 20, 30, 50, 100	A
Gigablast	163.384	332.096	485.226	3295.156	10, 20, 30, 40, 50	B
About	213.573	-	-	7981.556	10	B
AllTheWeb	115.173	-	-	1019.181	10, 25, 50, 75, 100	C
AOLSearch	251.778	-	-	9970.974	10, 15	B
FindWhat	201.118	-	-	1935.316	-	C
MozDex	183.009	-	-	2020.532	-	C
Msn	231.967	-	-	3591.208	15	C
Overture	107.061	-	-	3506.665	40	-
eSpotting	134.611	-	-	1128.627	10, 15, 25, 50	B
A9	214.728	-	-	1699.156	10	B

Tabella 2.4 – I dati raccolti (in ms)

Capitolo 3

ANALISI DEI TEMPI

Completata la prima fase di raccolta dei dati, si è passati alla loro analisi.

3.1 Connessione

Con l'eccezione di Google, i motori di ricerca hanno sempre fornito valori di PING dell'ordine di 100-140ms.

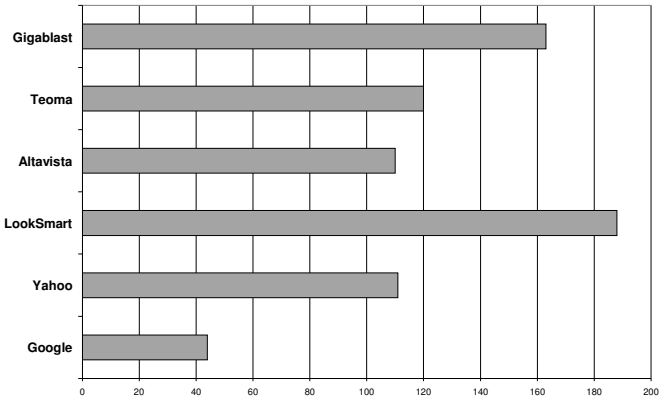


Grafico 3.1 – Tempi di PING

Nell'instaurazione delle connessioni utilizzando *socket bloccanti* [01] si sarebbe quindi perso nella migliore delle ipotesi, un tempo non inferiore alla somma dei valori di PING dei vari motori utilizzati, ovvero *736ms*.

Ciò era chiaramente inaccettabile per il nostro scopo, tantopiù che trascorso tale intervallo si dovevano ancora iniziare a ricevere i risultati, e procedere alla loro analisi. Decisamente troppo.
L'utilizzo di *socket non bloccanti* [01] era quindi una necessità, soprattutto in fase di connessione. L'implementazione dell'*I/O Async*, ha permesso di uscire dalla prima fase della connessione (quella che tecnicamente viene chiamata *handshaking*) dopo soli 681 microsecondi. Sia ben chiaro, dopo tale tempo non si sono già stabilite le connessioni, si sono solamente fatte le richieste. In questo modo però il software è già pronto per la trasmissione delle richieste HTTP, che vengono inviate non appena la connessione è instaurata ed è possibile trasmettere sul canale.

3.2 Trasmissione

Stabilita la connessione ad un server web, sia esso un motore di ricerca, un servizio di meteo, o un server privato, per ottenere una risposta bisogna trasmettere la "richiesta HTTP".

In questo standard le richieste sono fatte utilizzando semplici stringhe di testo. La stringa inizia con il comando "GET", che identifica la richiesta di una pagina. Il comando è seguito dall'indirizzo locale della pagina a cui si vuole accedere, e dall'indicazione, tramite il suffisso "HTTP/1.X" della versione del protocollo da utilizzare nella risposta. Ogni stringa va terminata con i caratteri speciali "\r\n", che permettono al server di comprendere che la linea è terminata e dare così inizio all'acquisizione del comando.

Possono esserci altri parametri nella richiesta HTTP, talvolta opzionali, in alcuni casi obbligatori: molti motori di ricerca ad esempio, si rifiutano di iniziare la ricerca se non viene specificato il parametro "HOST" previsto dallo standard HTTP/1.1. Altri motori di ricerca restituiscono un errore se non viene specificata, tramite il parametro "User-Agent", la versione del browser in uso, o il tipo di tabella codici utilizzato.

Una tipica richiesta HTTP è quindi

```
GET /index.htm HTTP/1.0
HOST: www.foo.com
User-Agent: Browser/2.0  [en]
```

I dati necessari per l'interrogazione di un motore di ricerca sono in genere molto pochi e vanno, nel nostro caso, da un minimo di 50bytes ad un massimo di 200 bytes.

La trasmissione di un così breve messaggio, perdipiù di solo testo, è quindi limitata solo dalla larghezza di banda. Su *Roquefort* vengono impiegati mediamente 500 microsecondi per trasmettere i 200bytes necessari ad una richiesta HTTP complessa. Tale tempo è quindi trascurabile.

3.3 Ricezione

Una volta instaurate le connessioni ed inviate le richieste, si è pronti per la fase di ricezione delle risposte. In questa fase, i tempi sono influenzati da più parametri:

- La *distanza del motore di ricerca*, ben riassunta dal tempo di PING, è influenzata dal livello di congestione della rete, ed in particolare, da quello di tutti i server/router/switch che faranno transitare i nostri pacchetti da e verso il server.
- Il *tempo di analisi della richiesta*, è composto dal tempo che la richiesta passerà nella coda di attesa (del protocollo TCP) del server, più il tempo necessario al motore per analizzarne i parametri.
- Il *tempo di ricerca*, è completamente dipendente dal server. E' influenzato dal numero di richieste simultanee che stà servendo, dalla dimensione del suo database, e dall'algoritmo utilizzato per il recupero delle informazioni.
- Il *tempo di trasmissione* di risultati dipende sia dalla larghezza di banda disponibile che dalla "pesantezza" delle pagine web di risposta, spesso piene di "informazioni" inutili e pubblicità.

Il grafico (3.2) mostra la dimensione delle pagine ricevute dai motori di ricerca usati per i test, per le due query "Bush" e "Madonna", richiedendo 100 risultati per motore.

Come si può notare LookSmart, 148.799bytes per 105 risultati, risulta essere il motore di ricerca con meno "fronzoli" nelle pagine di risposta, seguito a breve distanza da GigaBlast e Google, con un rapporto bytes/risultato pari a 1.500bytes.

Altavista al contrario, è il motore di ricerca con il peggior rapporto tra dati utili e dati inutili, con ben 3.597bytes inviati per ogni risultato.

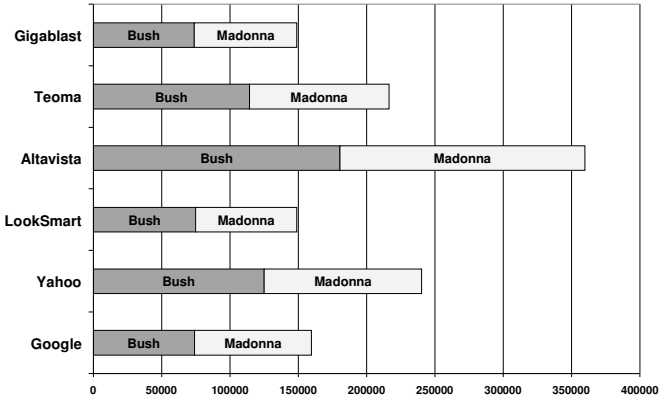


Grafico 3.2 – Dimensione pagine di risposta

3.4 Tempi di attesa

Il grafico (3.3) confronta i tempi totali di attesa dei dati, tra i motori di ricerca utilizzati nelle prove. La prima area, in grigio chiaro, corrisponde al tempo di PING; la seconda, colorata di grigio scuro, è il tempo di attesa che il server remoto impiega per assegnarci una connessione "pronta" per lo scambio dei dati; la terza area, colorata in bianco, equivale al tempo di attesa della risposta da parte del server.

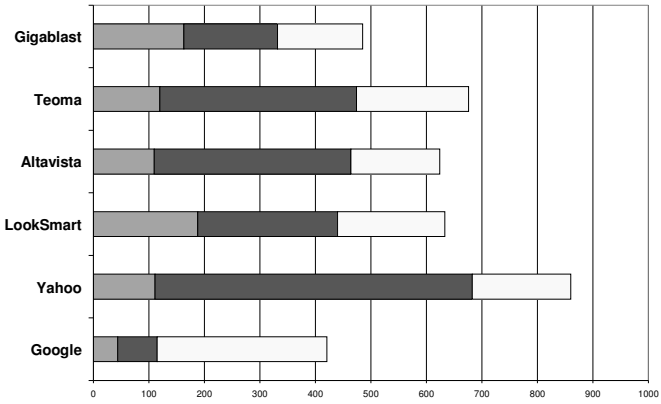


Grafico 3.3 – Tempi totali di attesa

Google, pur avendo un tempo di "attesa della risposta" più alto degli altri motori, ha i migliori tempi di PING e di connessione, risultando in questo modo il motore

più veloce nel rispondere alle richieste degli utenti. Il suo basso rapporto dati utili/inutili, rende inoltre molto veloce il download delle pagine di risposta facendolo sembrare ancora più veloce.

3.5 – Download delle risposte

Il grafico (3.4) mostra i tempi di download delle pagine di risposta a partire dall'inizio della ricezione dei dati, fino al loro completamento.

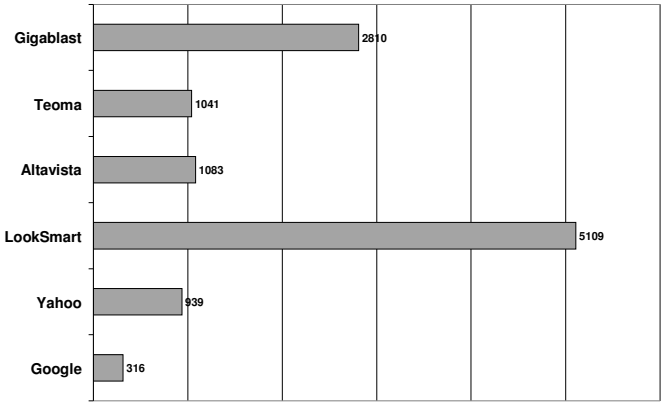


Grafico 3.4 – Tempi di download delle risposte (ms)

Google è il più veloce, con tempi 3 volte inferiori agli altri motori di ricerca. E' interessante notare come Altavista, che prima abbiamo visto essere il motore con le pagine di risposta più "pesanti", sia tra i più veloci nel completarne la trasmissione.

Le pessime performance di LookSmart sono da imputare all'irrimovibile limitazione dei 15 risultati per pagina. Il nostro software infatti, nel caso non riesca ad ottenere tutti i risultati desiderati in una stessa pagina, suddivide la ricerca in più richieste, e le sottopone al motore di ricerca in maniera seriale. Se per esempio si è interessati ai primi 100 risultati di LookSmart, si è costretti ad eseguire ben 7 interrogazioni consecutive. L'overhead generato dalla gestione di una nuova connessione, necessaria per ogni richiesta di pagina (secondo lo standard HTTP/1.0) è molto alto, e come dimostra il grafico, pesa notevolmente sulle performance del software. Il software prevede però la possibilità di parallelizzare anche le richieste multiple indirizzate ad uno stesso motore: sfruttando questa caratteristica per richiedere le stesse 7 pagine a LookSmart, riusciamo a completarne il recupero in appena *985ms*, equivalenti a circa il 19% del tempo precedente. Per maggiori informazioni al riguardo, vedere il *Capitolo 6*, dedicato ai test.

Riducendo i nostri test ad una sola pagina, che contenga 10 o 15 risultati poco importa, scopriamo che LookSmart risulta il 3° motore più veloce del web, ed anche che Gigablast, con solo 9245 bytes per pagina, guadagna parecchie posizioni nella classifica dei motori di ricerca più veloci. Google, con i suoi 380ms di tempo totale per soddisfare la richiesta, rimane l'indiscusso vincitore di questa gara all'ultimo bit.

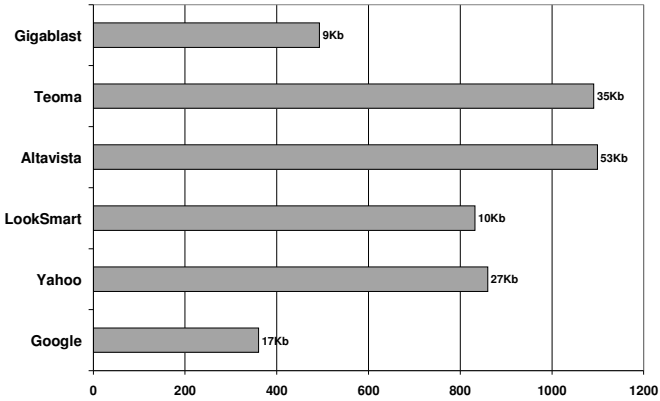


Grafico 3.5 – Tempi di completamento della richiesta (10 risultati)

3.6 – Parallelizzazione

Alla luce di questi dati, l'introduzione di un sistema di richieste parallele [02] nel sistema progettato, è stato una necessità.

Nel Capitolo 6 del libro di W.R.Stevens [01] sono illustrati molti modelli di client che sfruttano l'*I/O Async*. Dopo alcune valutazioni, si è deciso di utilizzare per questo progetto l'*I/O Multiplexing*, ed in particolare la funzione di sistema `select`. Ciò ha permesso di parallelizzare l'invio delle richieste HTTP ai vari motori, così come parallelizzarne la ricezione ed il parsing delle risposte, senza dover ricorrere all'utilizzo dei thread.

Tale funzione lavora su due *set di bit* (uno per la ricezione, ed uno per la trasmissione), inizializzati grazie a delle macro e ai i socket delle varie connessioni con i motori di ricerca. La chiamata alla funzione ritorna quando almeno uno dei vari socket è pronto in lettura o scrittura, oppure, quando si verifica un timeout (se impostato). Nel software, dopo aver richiesto una connessione TCP ad ogni motore, si attende l'uscita della `select`. A questo punto si individuano i socket pronti in scrittura e si inviano le relative richieste HTTP o, nel caso sia segnalato pronto in lettura, si procede allo svuotamento del relativo buffer ed al parsing dei

dati ricevuti. Maggiori informazioni sulle caratteristiche delle varie funzioni di connessione, ricezione e trasmissione, possono essere trovate direttamente tra i commenti del codice.

In questo modo non siamo costretti ad attendere il completamento di ognuna delle richieste per poter richiedere/ricevere i dati dagli altri engine, ed il tempo di completamento totale sarà dell'ordine di quello più lento.

A titolo di esempio, una richiesta parallela per la query "bush" ha restituito 605 risultati in 4608ms. La stessa query, eseguita sul solo motore LookSmart, ha restituito 105 risultati in 4646ms.

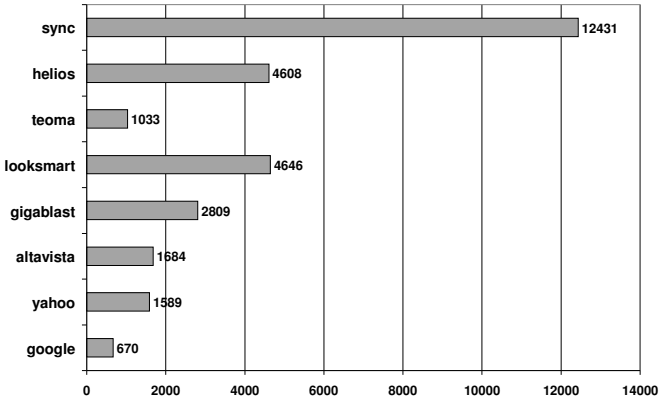


Grafico 1.1 – Tempi di completamento di una richiesta (100 ris.)

Capitolo 4

IL PARSER

Scrivere un parser efficente ed espandibile era l'obiettivo secondario di questo mio tirocinio. Il parser non solo doveva riuscire ad estrapolare dallo "sporco" HTML delle pagine di risposta dei vari motori di ricerca le giuste informazioni, ma doveva anche consentire una grande manutenibilità: non è raro infatti che un motore di ricerca cambi, anche se impercettibilmente, la grafica della sua pagina di risposta o i tag usati per la definizione dei font, rendendo quindi necessari piccoli ritocchi al suo codice. Inoltre, in accordo al requisito di espandibilità massima, doveva essere possibile aggiungere nuovi motori di ricerca, senza toccare il codice del software.

4.1 Un esempio di risposta

Per permettere a chi legge, di capire meglio l'ambiente in cui si muove il parser, riporterò qui sotto un estratto dell'HTML di una pagina di risposta di Google:

```
[…]
<a class=fl href=/search?hl=en&lr=&ie=UTF-
8&q=related:www.madonnashots.com/>Similar pages</a><
/font></td></tr></table> <p class=g><a
href=http://www.madonnainn.com/><b>Madonna</b> Inn - San
Luis Obispo, California</a><table cellpadding=0
cellspacing=0 border=0><tr><td class=j><font size=-1>
<b>...</b> The world-renowned <b>Madonna</b> Inn offers
108 rooms, uniquely decorated
with a special theme and color scheme, no two alike!
Enjoy your <b>...</b>
[…]
```

Come si può notare il codice è molto confuso. Anche con la sottolineatura si fatica a comprendere le informazioni che vi sono racchiuse all'interno: il tag `<a href>`

indica l'inizio di un link, specificandone anche l'indirizzo di destinazione "http://www.madonnainn.com/"; segue il titolo della pagina "Madonna Inn – San Luis Obispo, California" che sarà poi *sensibile* al click del mouse; il tutto è terminato dallo *snippet*, ovvero un estratto della pagina puntata, che cerca di aiutare l'utente nella scelta di quale link seguire.

Una volta identificata (e compresa) la struttura in cui sono organizzati i dati, a chi è dotato di vista, non appare poi così difficile estrarre le informazioni. Per un computer però, la cosa è molto più difficile.

4.2 Studio del problema

Le soluzioni percorribili [11], con un occhio sempre puntato verso l'efficenza, erano comunque molte:

I *generatori di parser*, come Bison [05], Flex [06], o Gold Parser Builder [12] sono in grado, una volta specificata la struttura delle informazioni, di scrivere autonomamente il codice C necessario al loro riconoscimento. Nel nostro progetto però non era possibile utilizzarli, se non come parametro di riferimento per le prestazioni; il nostro meccanismo di parsing doveva essere espandibile senza dover intervenire sul codice, e questo rendeva inutile il disporre di un codice pronto per essere compilato (seppur preciso ed efficente).

L'utilizzo delle *espressioni regolari*, con la loro smisurata potenza, poteva essere una buona soluzione al nostro problema. Implementare [10] un "automa" in grado di riconoscere un'espressione regolare non è un compito difficile, ma lo scrivere buone espressioni regolari che non generino falsi match, lo è.

La *costruzione di un albero*, o di una lista, tramite l'analisi [09] dei tag HTML era un'altra delle soluzioni percorribili. L'implementazione si basava sulla ricerca sequenziale dei caratteri di inizio "<" e fine ">" di un tag, con conseguente analisi di ciò che vi era racchiuso per comprenderne il contenuto. Il nostro scopo però era l'estrazione veloce delle informazioni e non la comprensione della struttura della pagina; questo metodo, seppur potenzialmente il più efficace e facile da istruire (è sufficente specificare una entry-tipo) è anche il più lento e difficile da codificare a causa delle innumerevoli eccezioni.

Un'ultima soluzione dall'implementazione molto semplice, ma allo stesso tempo dalle ridotte capacità, era l'utilizzo dei *marker*. Questo metodo prevede la costruzione di una tabella in cui si contraddistingue ciò che precede l'inizio e ciò che segue la fine di una informazione utile. In questo modo, con una semplice ricerca di sottostringhe all'interno del buffer di ricezione, si possono estrarre parti di testo "sperando" che contengano l'informazione desiderata. E' però raro che un motore di ricerca "marchi" sempre l'inizio e la fine di un link, di un titolo, o di uno

snippet, allo stesso modo. Spesso i tag usati, ed i vari tipi di formattazione dell'HTML, variano di giorno in giorno, o dipendono addirittura dai parametri della ricerca. Questo invalida l'utilizzo del sistema basato sui marker.

4.3 La soluzione adottata

Lo studio delle possibili alternative, e due chiacchere con un amico, mi hanno fatto pensare ad un'altra possibile soluzione al problema.

L'idea si basa sulla costruzione di un piccolo *linguaggio di programmazione per parser*, con cui vengono costruiti gli automi per il riconoscimento dell'HTML. Il linguaggio, dipendente dal motore di ricerca, viene interpretato *on-fly* al momento dell'esecuzione del parser, e tradotto in funzioni implementate all'interno del programma.

Inizialmente si è partiti con istruzioni semplici come *LOOKUP* e *GRAB*, per poi estendere il linguaggio con costrutti più complessi come *JUMP* condizionali, *IF* e *RETURN*.

Questa soluzione sembra permettere la necessaria espandibilità e manutenibilità del software, con prestazioni di tutto riguardo: nei test più recenti il parser ha dimostrato di necessitare di *soli 39ms per estrarre 715 risultati* dalle varie pagine di risposta dei motori di ricerca.

Al generico passo di iterazione il parser decodifica la prossima istruzione da eseguire tra quelle relative al motore di ricerca, e la esegue su quanto presente nel buffer. Dopo ogni istruzione eseguita con successo, l'*istruction counter* viene incrementato, andando a posizionarsi sul comando seguente.

Si troveranno le specifiche delle funzioni supportate dal parser nel *Paragrafo 4.6*, mentre per avere maggiori informazioni sul funzionamento del parser riferirsi ai commenti presenti nel codice del file `parser.c`.

4.4 Sintassi degli script

In ogni script i comandi vengono scritti su linee diverse, ed hanno una struttura standard:

```
<c> <param>
```

Il primo carattere è riservato all'identificazione del comando. Si è scelto di utilizzare un solo carattere per rendere più facile (e più veloce) sia la codifica che la decodifica degli script, ma volendo è possibile implementare i comandi anche per esteso. La maggior parte dei comandi, hanno un solo parametro, che viene specificato subito dopo il carattere che lo contraddistingue, separato da uno spazio. Ci sono comandi speciali, come *CLEAN* o *PRINT*, che non richiedono parametri, ed altri, come *IF*, che ne richiedono invece due. Il generico script termina con un comando *PRINT*, ed un *JUMP* alla linea di codice che deve essere la prossima ad essere eseguita.

Il codice dei parser per i motori attualmente funzionanti con il software può essere trovato nella *Appendice A*.

4.5 Esempio di script

Riporto qui sotto, a titolo di esempio, l'attuale script del parser di Google:

```
 0:  l <font size=-1 color=#000000>Results
 1:  + 35
 2:  l </b> of
 3:  l <b>
 4:  + 3
 5:  g FOUND
 6:  u </b>
 7:  l <p class=g><a href=
 8:  + 19
 9:  g URL
10:  u >
11:  g TITLE
12:  u </a>
13:  c
14:  i 100 <font size=-1> - [
15:  l </font>
16:  l <font size=-1>
17:  g TEXT
18:  u <font color=#008000>|<a class=fl
19:  c
20:  p
21:  j 7
```

Alla riga 0, il parser cercherà (L) nel buffer la sottostringa `<font size=-1 color=#000000>Results`, spostandosi poi al suo inizio. Alla riga 1, il parser sposterà il puntatore (+) di 35 caratteri in avanti, arrivando alla fine della sottostringa trovata in precedenza. La riga 2 farà cercare (L) al parser la sottostringa `</b> of`, la riga 3 gli farà cercare una il successivo tag `<b>`, e la riga 4 lo farà spostare (+) davanti ad esso.

Alla riga 5, il parser comincerà a memorizzare (G) ciò che troverà nella variabile `FOUND`, prima di incontrare il tag `</b>` come specificato alla riga 6 dal comando *until* (U).

Verrà quindi eseguita una nuova ricerca (riga 7), ed il *grab* (G) dell'url, fino al carattere `>`. La riga 11 da inizio al *grab* (G) del titolo della pagina fino ad incontrare il tag `</a>`, come specificato alla riga 12. Il comando *clean* (C) alla riga 13, eseguirà un algoritmo che "ripulirà" il titolo estratto da eventuali tag interni (è comodo per rimuovere ad esempio il grassetto).

Alla riga 14 incontriamo un *if* (I), che cercherà la presenza della sottostringa `<font size=-1> - [` nei successivi `100` caratteri. Nel caso la trovi, sarà eseguita la ricerca del comando successivo, ovvero la ricerca del tag `</font>`, altrimenti, l'istruzione sarà saltata e verrà subito eseguita la *lookfor* (L) del tag `<font size=-1>` specificata alla riga 16.

Le righe 17, 18 e 19, si occupano dell'estrazione dello snippet, e della sua pulizia da eventuali tag di formattazione del testo.

Alla fine dello script, come annunciato nell'introduzione di questo paragrafo, troviamo l'istruzione *print* (P) che fà stampare a video quanto estratto, seguita dall'istruzione *jump* (J) che fa ripartire l'algoritmo dalla riga 7, saltando quindi l'estrazione del numero di pagine trovate che, chiaramente, va eseguita solo una volta all'inizio del parsing della pagina.

4.6 Comandi supportati

LOOKFOR: `l <text>`
Cerca la stringa specificata nel parametro `<text>` in avanti a partire dalla posizione corrente del puntatore. Nel caso venga trovata, il puntatore viene spostato sul suo primo carattere.

MOVE FW: `+ <n>`
Sposta il puntatore in avanti di un numero `<n>` specificato di caratteri.

MOVE BW: `- <n>`
Sposta il puntatore indietro di un numero `<n>` specificato di caratteri.

GRAB: `g <var>`
Inizia l'estrazione dei dati, a partire dalla posizione corrente, fino a quella specificata dal comando *until*. Quanto estratto viene memorizzato nella variabile specificata da `<var>`. Le variabili utilizzabili sono: FOUND, URL, TITLE, TEXT.

WRITE: `w <text>`
Accoda quanto specificato in `<text>` al contenuto della variabile puntata dal comando *grab* precedente.

UNTIL: `u <text>`
Termina l'estrazione dei dati, iniziata dal comando *grab*, appena viene incontrato la sottostringa specificata da `<text>`. Il puntatore viene spostato sul carattere successivo alla sottostringa trovata.

IF: `i <n> <text>`
Cerca la sottostringa `<text>` a partire dalla posizione corrente, nei successivi `<n>` caratteri. Se viene trovata un'occorrenza, si passa ad eseguire l'istruzione successiva, altrimenti, l'istruzione viene saltata e si passa subito a quella seguente. Il puntatore non viene comunque spostato.

PRINT: `p`
Stampa a video l'attuale contenuto delle 4 variabili, secondo una struttura XML concordata.

JUMP: `j <line>`
Salta alla linea specificata dal parametro `<line>` e ricomincia l'esecuzione dello script.

Capitolo 5

IL SOFTWARE

5.1 - Binaries

Il *pacchetto binario* è composto da 3 classi di file: il file eseguibile `helios`, che consente l'esecuzione della ricerca ed la specifica dei suoi parametri; il file `engines.dat`, che contiene i dati relativi ai motori di ricerca che verranno utilizzati, come l'indirizzo IP e la richiesta HTTP; i file con estensione `.PRS`, che contengono il codice del parser dei vari motori di ricerca.

5.2 – Organizzazione dei sorgenti

I sorgenti del software sono organizzati in 8 file `.c`, ognuno dedicato ad una diversa sezione del codice:

bench.c, bench.h
Qui sono racchiuse le funzioni per il calcolo delle performance del software, con precisione al microsecondo (1/1000 di millisecondo) grazie all'utilizzo della funzione di sistema `gettimeofday()`.

cmdline.c, cmdline.h
Contengono l'implementazione delle funzioni dedicate all'estrazione delle informazioni dalla linea di comando come: la query (semplice o esatta), i vari flag (disabilitazione del parser, verbose mode...), i motori di ricerca da utilizzare, il numero di risultati per pagina, il numero di pagine da recuperare, quale è la pagina iniziale...

core.c, core.h

Questi due file contengono il cuore del software. La procedura di ricerca, la misurazione dei tempi, le funzioni di trasmissione e ricezione dei dati, sono tutte contenute in questi due file.

helios.c
Contiene la funzione `main()` del software.

macro.h
Questo file contiene le macro definite per il programma ed i "magic number" usati, come la dimensione del buffer di invio/ricezione o il numero di motori utilizzabili simultaneamente.

parser.c, parser.h
Come si può intuire dal loro nome, le funzioni legate al parser sono racchiuse in questi due file, così come l'interprete del suo piccolo linguaggio di programmazione.

searchengines.c, searchengines.h
Questi due file contengono le routine di gestione dei motori di ricerca. Qui troviamo le funzioni di lettura del file `engines.dat` e dei vari file `.prs` necessari alla costruzione dell'array contenente i motori di ricerca utilizzabili. Si trovano in questi file anche le routine per la semplificazione delle connessioni, per la risoluzione di nomi di dominio, e per la costruzione e l'invio delle richieste HTTP.

socketutils.c, socketutils.h
Questi file racchiudono le funzioni relative alla manipolazione dei socket. La loro creazione, l'impostazione dei vari parametri (compreso il famoso NON_BLOCK), e la chiusura delle connessioni, vengono tutte effettuate attraverso chiamate a funzioni presenti in questi due file.

utils.c, utils.h
Le funzioni di utilizzo generale, come la `itoa()` che serve per convertire i numeri in stringhe, oppure la `strrpl()` che viene usata per sostituire i tag dei modelli delle query con i dati passati come parametro, trovano posto in questi due file.

5.3 – Installazione e compilazione

L'installazione del software è estremamente semplice. Una volta posizionati nella cartella `src`, non si deve far altro che digitare il comando

```
make
```

per ottenere nella directory padre il file eseguibile `helios`. Nel caso si volesse utilizzare strumenti di profiling, o di debug del software, il comando

```
make debug
```

consentirà di ottenere una versione del software adatta allo scopo. Inoltre, il comando

```
make clean
```

consente di fare pulizia tra i file generati dal programma, soprattutto nella versione di debug.

5.4 – Uso

La sintassi con cui richiamare il programma è questa

```
helios <flags> <query> <engines>
```

Le <*flags*> servono per attivare (o disattivare) alcune funzionalità del programma che risultano utili solo in certe occasioni, come al momento della scrittura del codice del parser di un nuovo motore, oppure durante l'analisi delle performance del programma. Di seguito riporto le più importanti:

`-p` disabilita il parsing dei dati ricevuti dai vari motori, che saranno comunque ricevuti, ma non riportati a video.

`-d` crea un file su disco, con il nome del motore di ricerca, e ci appende quanto ricevuto in risposta.

`-v` attiva il verbose mode, riportando a video una notevole quantità di informazioni supplementari, come le richieste HTTP inviate, o le READ eseguite.

`-s` riporta a video informazioni sui socket, come connessioni e disconnessioni.

`-n` disattiva la stampa a video dei risultati che sono stati correttamente processati dal parser. Può essere utile nell'analisi delle prestazioni del software, dove la stampa a video influenzerebbe comunque il calcolo dei tempi di completamento.

La <*query*> è certamente la parte più importante della ricerca. Sono possibili *ricerche su stringhe esatte*, racchiudendo la stringa di cercare tra doppi apici

```
"White House"
```

o *ricerche semplici*. In quest'ultimo caso i termini della ricerca vanno uniti tra di loro con il segno "+": il software eseguira l'interrogazione dei vari motori locali mettendoli in AND.

```
car+pegeout+206
```

Nelle ricerche semplici formate da una sola parola il segno "+" può anche essere omesso

```
flowers
```

Le ricerche semplici composte da più parole vengono effettuate mettendo in AND i vari termini. La ricerca dell'esempio `car+pegeout+206` restituirà quindi tutte le pagine contenenti le parole *car*, *pegeout* e *206*.

La linea di comando è conclusa dalla sezione <*engines*>, ovvero dalla specifica dei motori di ricerca da utilizzare. Ogni motore di ricerca deve essere specificato con la sintassi

```
engineName=rpp,pag,first
```

dove ad `engineName` va sostituito il nome di uno dei motori di ricerca specificati dentro il file `engines.dat`; il parametro `rpp`, indica il numero di risultati per pagina richiesti; il parametro `pag`, specifica il numero di pagine richieste; `first`, indica quale deve essere la prima pagina. Ad esempio,

```
google=100,3,2
```

indica al software di eseguire la ricerca anche su Google, richiedendo 3 pagine (la 2°, la 3°, e la 4°) da 100 risultati ciascuna. Allo stesso modo, il blocco

```
looksmart=15,1,7
```

indica al software di eseguire la ricerca anche sul motore LookSmart, richiedendo 1 pagina (la 7°) con 15 risultati.

I parametri di un motore possono essere anche omessi. In tal caso saranno utilizzati quelli presenti nel file `engines.dat`. Specificando solo

```
google
```

ad esempio, è come se si scrivesse `google=100,1,1`, poichè nel file `engines.dat` sono presenti tali dati. Per ogni motore il file di configurazione specifica i parametri che consentono di ottenere 100 risultati nel più basso tempo

possibile. E' possibile omettere anche solo qualche parametro (preservando però l'ordine), specificando ad esempio `google=50`, è come se si scrivesse `google=50,1,1`.

La *ricerca parallela* su più motori di ricerca viene eseguita specificando più di un motore nella sezione <*engines*>. I blocchi dei parametri di un motore devono essere separati gli uni dagli altri, attraverso uno spazio:

```
google=10,3,1 looksmart=15,2,1 msn=15,2,1
```

equivalente alle forme più compatta, ottenuta attraverso l'omissione dei parametri

```
google=10,3 looksmart=15,2 msn=15,2
```

Come ultimo esempio, ricordo che se si vogliono utilizzare direttamente i parametri specificati nel file `engines.dat`, il blocco <*query*> può anche essere specificato come semplice elenco dei nomi dei motori di ricerca, separandoli con uno spazio

```
google looksmart yahoo altavista msn
```

5.5 – Esempi di ricerche

Ricerca della stringa esatta "*Alessio Signorini*" tra le prime 100 pagine di alcuni motori di ricerca

```
helios "Alessio Signorini" google looksmart yahoo msn
```

Ricerca delle parole *cane* e *gatto* tra i risultati 10-20 di alcuni motori di ricerca

```
helios cane+gatto google=10,1,2 yahoo=10,1,2 msn=10,1,2
```

Dump su disco della risposta (una pagina, composta da 10 risultati) alla query *bush* di AbsoluteSearch, un nuovo motore di ricerca immaginario di cui si deve ancora scrivere il parser

```
helios –d –p bush absolutesearch=10,1
```

Test del tempo di completamento della ricerca flowers, per la quale vogliamo i primi 100 risultati di alcuni motori di ricerca, escludendo la stampa a video dei risultati in modo da evitare possibili influenze sui tempi

```
helios –n flowers yahoo mozdex gigablast
```

Controllo dell'utilizzo dei socket da parte del software, in una ricerca formata da più motori e più pagine

```
helios –s –v –p car yahoo=10,6 msn=15,9 aol=10,7
```

5.6 – Il file ENGINES.DAT

Come detto in precedenza questo file contiene le impostazioni di tutti i motori di ricerca utilizzabili dal software. E' organizzato in blocchi, uno per ogni motore di ricerca, con questa struttura:

```
[foofoo]
#host                  =       www.foofoo.com
host                           =       211.145.59.104
port                           =       80
desidered_pages                =       1
result_per_page                =       100
next_page_calc         =       2
{
        GET /s?q=%QUERYSTR&n=%RESULTS&p=%PAGE HTTP/1.0
        HOST: www.foofoo.com
        User-Agent: Browser/2.0  [en]

}
```

La linea `[foofoo]` identifica l'inizio del blocco di un nuovo motore, il cui nome è quello racchiuso tra le parentesi quadre. Le righe che iniziano con `#` sono ignorate perchè considerate commenti.

La riga `host`, specifica l'indirizzo del motore di ricerca. Se viene indicato come dominio, il software si occuperà di tradurlo in un indirizzo IP ad ogni ricerca. La traduzione introdurrà però un considerevole (circa 200ms) ritardo, dovuto all'accesso al DNS locale. Se l'indirizzo viene invece indicato nella "dotted-form" (es. 157.110.152.124) la traduzione sarà molto più veloce. Gli indirizzi vengono tradotti attraverso le funzioni di sistema `gethostbyname` e `gethostbyaddr`, di

cui si può trovare un'ottima trattazione nel Capitolo 9 del libro di W.R. Stevens [01].

La riga `port`, specifica la porta del server a cui collegarsi. Le 3 righe seguenti `desidered_pages`, `result_per_page`, `next_page_calc`, hanno nomi autoesplicativi che indicano rispettivamente: il numero di pagine desiderate, il numero di risultati per pagina, ed il sistema di calcolo della pagina successiva (vedere il *Capitolo 2*, dedicato ai Search Engines, per maggiori informazioni al riguado).

Le *parentesi graffe* racchiudono la richiesta HTTP che sarà inviata al server. E' importante lasciare una riga vuota alla fine di tale richiesta, poichè prevista dallo standard HTTP. All'interno della richiesta è possibile scrivere qualsiasi cosa, specificando ogni parametro che possa essere utile (permettendo così di utilizzare anche il metodo POST dei form, anzichè solo il GET), compresa la versione del browser da simulare o l'indicazione di lingue, tipi di documento accettati e simili.

5.7 – Aggiungere un motore di ricerca

Nel caso si voglia aggiungere un nuovo motore di ricerca al software, occorre innanzitutto eseguire qualche query di prova attraverso il proprio browser, in modo da individuare la sintassi corretta per la richiesta HTTP. A titolo di esempio riportiamo di seguito l'url di una risposta del motore di ricerca A9, che ci consentirà di individuarne la parte interessante

```
http://a9.com/bush?p=2
```

l'`host` è quindi A9.com, che possiamo inserire nel file ENGINES.DAT direttamente in questa forma, oppure nella già citata "dotted-form". Se si ha poca dimestichezza con DNS e simili, un semplice PING verso il dominio ci rivelerà il suo indirizzo IP.

La richiesta HTTP dovrà quindi iniziare con il comando GET, proseguire con quanto segue l'host nell'indirizzo della risposta, e terminare con l'indicazione del protocollo da utilizzare, HTTP/1.0.

```
GET /bush?p=2 HTTP/1.0
```

Adesso si sostituiscono i parametri della ricerca con i tag speciali che il software ci mette a disposizione. Si sostituisce quindi la parola bush con il tag %QUERYSTR, per indicare che li andrà copiata la stringa da ricercare, e il numero 2, con il tag % PAGE, per indicare il numero della pagina desiderata. In questo particolare caso non è possibile specificare il numero di risultati per pagina, e quindi non utilizzeremo il tag %RESULTS. Il risultato finale è quindi

```
GET /%QUERYSTR?p=%PAGE HTTP/1.0
```

Nell'URL, la seconda pagina era indicata con il numero 2. Il sistema di calcolo della pagina da visualizzare è quindi di *tipo B* (vedere il *Capitolo 2* dedicato ai motori di ricerca, per maggiori informazioni).

E' bene inserire nella richiesta HTTP anche parametri accessori come il parametro `HOST`, che aiuta i web-server che gestiscono più di un sito ad individuare a quale sito web passare la richiesta, ed il parametro `User-Agent`, che specifica la versione del browser in uso (che noi simuleremo) in quanto certi web-server restituiscono pagine diverse a seconda delle capacità del browser in uso. La richiesta completa diventa quindi

```
GET /%QUERYSTR?p=%PAGE HTTP/1.0
HOST: a9.com
User-Agent: Opera/7.50  [en]
```

Creata la nuova entry nel file `ENGINES.DAT` occorre creare un file fittizio per il parser che andremo a scrivere. Se il file non viene creato, il software ne segnalerà la mancanza interrompendo la sua esecuzione. Creiamo quindi il file vuoto `A9.PRS`.

Fatto ciò, lanciamo il programma specificando la flag `-d`, in modo da eseguire il dump della risposta su disco, e la flag `-p` in modo da escludere il parser (che altrimenti genererebbe un errore in quanto il file contenente il codice operativo è vuoto).

Sul disco verrà creato un file con il nome dell'`host` specificato in `ENGINES.DAT` e che conterrà tutti i dati ricevuti in risposta dal motore. Analizzando tale file non sarà difficile scrivere il parser per il nuovo motore di ricerca (si veda il *Capitolo 4* relativo al parser per la sintassi e la specifica dei comandi utilizzabili).

Capitolo 6

TEST

6.1 – Caricamento

Nella fase di preparazione all'esecuzione di una ricerca il software apre il file `ENGINES.DAT` e crea l'array contenente i motori di ricerca, per ognuno dei quali viene aperto e copiato in memoria lo script del parser presente nel relativo file `.PRS`.

I test fatti hanno dimostrato che il tempo necessario alle operazioni di "caricamento dei dati" e "preparazione dei socket", è di circa 1500 microsecondi (su *Roquefort*). La tabella seguente (6.1) mostra i tempi impiegati da ognuno dalle funzioni del software, quando eseguito su una "ricerca tipo" che coinvolge 6 motori di ricerca.

Load	*READ*	*WRITE*	*SELECT*	*Parser*
1.4ms	1418ms	3.5ms	4608ms	38ms
0.02%	23.38%	0.05%	75.98%	0.62%

Tabella 6.1 – Influenza delle varie funzioni sulle performance

Inizialmente si era pensato di creare questo software come "server" interrogabile da remoto, in modo da evitare che il tempo di caricamente e di preparazione delle connessioni si addizionasse a quello di ricerca, ritardando la restituzione dei dati. Addirittura, si era pensato di creare per tale scopo un modulo Apache. Un tempo così basso però, dimostra di essere praticamente ininfluente sulle performance globali del programma.

6.2 – Seriale vs Parallelo

Come era facile intuire l'introduzione di un modello di ricerca che sfrutti l'*I/O Async* ha aumentato le performance del software. L'iniziale modello seriale,

costruito per i primi test, impiegava circa 12 secondi per completare una ricerca su 6 motori.

L'attuale modello parallelo, come si può vedere dal grafico (6.2), impiega solo 4608ms per completare una richiesta di 100 risultati su 6 motori. Il tempo è praticamente equivalente a quello massimo, introdotto da LookSmart, e dimostra come la funzione sia ben implementata poichè non aggiunge ritardi al download dei dati.

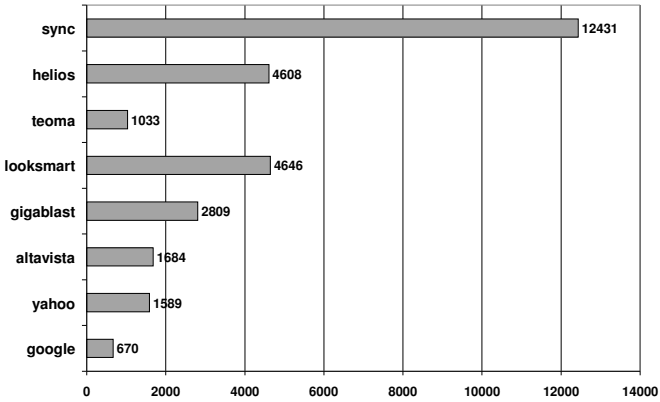


Grafico 6.2 – Tempi di completamento di una richiesta (100 ris.)

Come dimostra anche il grafico seguente (6.3), l'introduzione del modello parallelo ha ottimizzato i tempi "morti" dei vari motori di ricerca, fornendo risultati di tutto rispetto.

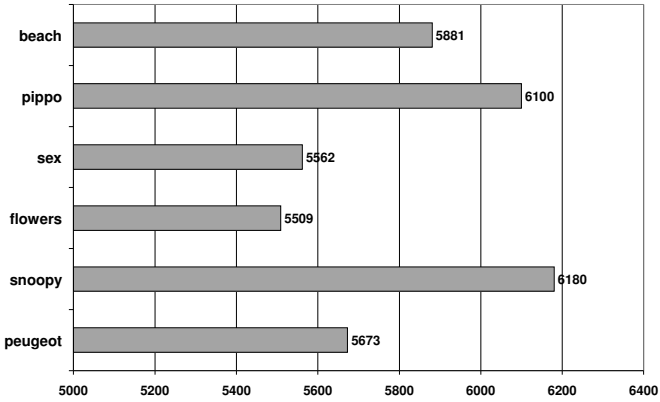


Grafico 6.3 – Ricerca parallela (6 motori) di alcune parole

Allo stesso modo è possibile parallelizzare le ricerche anche su uno stesso motore. LookSmart ad esempio, non permette di superare il limite dei 15 risultati per pagina e costringe il software ad effettuare ben 7 ricerche seriali. In questi casi, utilizzando l'*I/O Async* anche sul singolo motore, i tempi di completamento vengono drasticamente diminuiti, come dimostrato dal grafico (6.4).

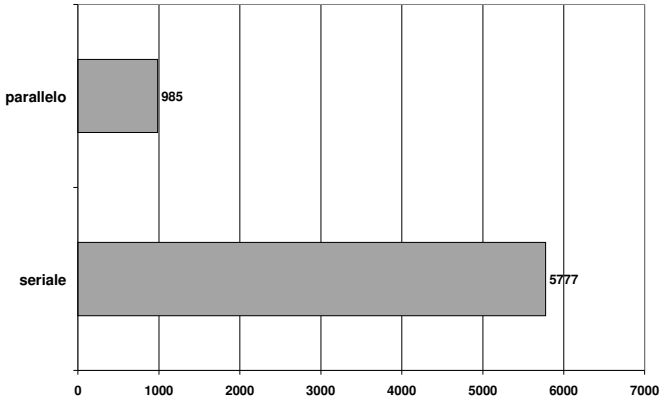


Grafico 6.4 – Ricerca su LookSmart con i due modelli

Il tempo si è ridotto a circa *985ms*, equivalenti al *17%* del tempo necessario ad eseguire la stessa ricerca in modo seriale. Considerando le performance generali di LookSmart questo è un risultato molto buono.

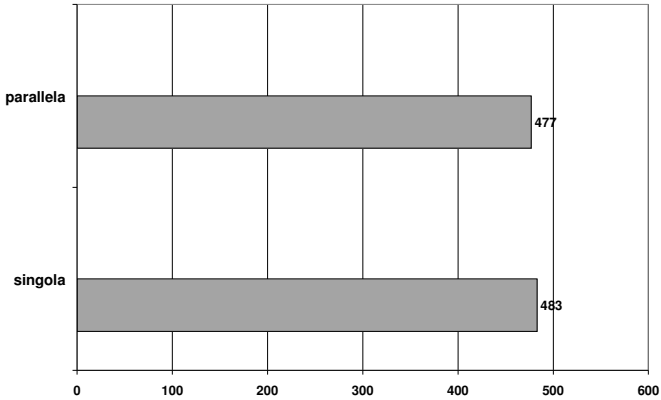


Grafico 6.5 – Richiesta di 100 risultati a Google

Non sempre però è così: suddividendo una richiesta di 100 risultati in 10 richieste parallele, e sottoponendola ad un motore veloce come Google, non si ottiene praticamente nessun beneficio, come si può vedere dal grafico (6.5).

6.3 – Il parser

Le performance del parser sono in genere molto buone, e si possono grossolanamente approssimare con una proporzione sulle linee di codice dello script.

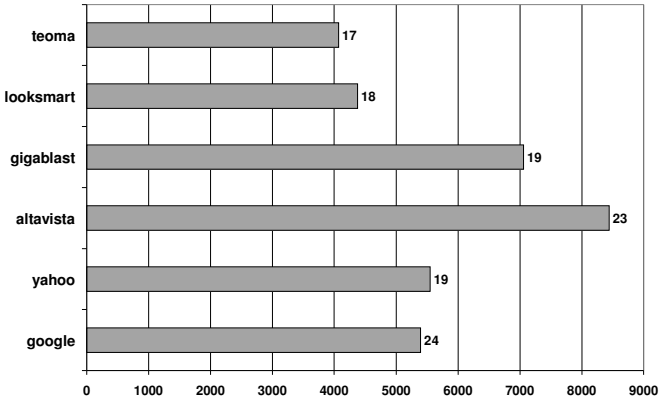


Grafico 6.6 – Rapporti tra linee di codice (a destra) e costo del parsing (in m-sec)

E' possibile stimare il costo di esecuzione in maniera più precisa.

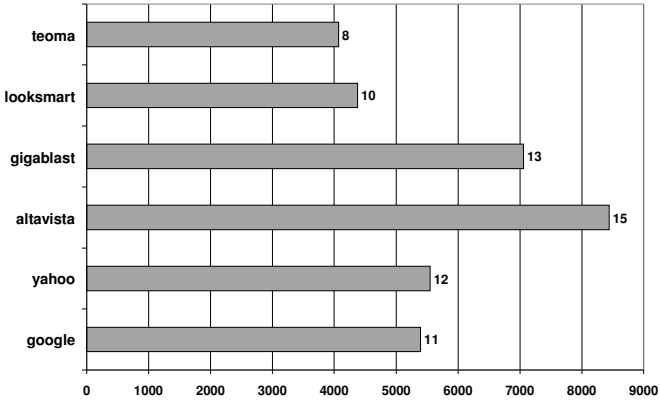


Grafico 6.7 – Rapporto tra i comandi "pesanti" ed il costo del parser

Calcolando il numero di `lookfor`, `until`, ed `if` utilizzati nello script, poichè sono i comandi più “dispendiosi” in termini computazionali e quindi i più pesanti nell’incidere sulle sue performance, si ottiene una buona approssimazione dei tempi di parsing.

Bibliografia

[01] UNIX NETWORK PROGRAMMING, VOL.1: NETWORKING API & SOCKET
by W. Richard Stevens, Bill Fenner, Andrew M. Rudoff
http://www.kohala.com/start/unpv12e.html

[02] INTERNETWORKING WITH TCP/IP, VOL.3: CLIENT-SERVER PROGRAMMING AND APPLICATIONS
by Douglas E. Comer, David L. Stevens, Michael Evangelista
http://www.cs.purdue.edu/homes/dec/netbooks.html

[03] THE PRACTICE OF PROGRAMMING
by Brian W. Kernighan, Rob Pike
http://cm.bell-labs.com/cm/cs/who/bwk/

[04] LIBCURL
URL transfer library
http://curl.haxx.se/libcurl/

[05] BISON
general-purpose parser generator
http://www.gnu.org/software/bison/

[06] FLEX
fast lexical analyser generator
http://www.gnu.org/software/flex/

[07] BUILDING EFFICIENT AND EFFECTIVE METASEARCH ENGINES
by Weiyi Meng, Clement Yu, King-Lup Liu
http://doi.acm.org/10.1145/505282.505284

[08] EFFICIENT AND EFFECTIVE METASEARCH FOR TEXT DATABASES INCORPORATING LINKAGES AMONG DOCUMENTS
by Clement T. Yu, Weiyi Meng, Wensheng Wu, King-Lup Liu
http://doi.acm.org/10.1145/375663.375684

[09] NoDoSE - semi-automatically extracting structured and semistructured data from text documents
by Brad Adelberg
http://doi.acm.org/10.1145/276304.276330

[10] IEPAD: information extraction based on pattern discovery
by Chia-Hui Chang, Shao-Chen Lu
http://doi.acm.org/10.1145/371920.372182

[11] A brief survey of web data extraction tools
Alberto H. F. Laender, Berthier A., Altigran S., Juliana S. Teixeira
http://doi.acm.org/10.1145/565117.565137

[12] GOLD Parser Builder
free pseudo-open-source parsing system
http://www.devincook.com/goldparser/

[13] Who Powers Whom
by Danny Sullivan for SearchEngineWatch.com
http://searchenginewatch.com/reports/article.php/2156401

[14] Search Engine Sizes
by Danny Sullivan for SearchEngineWatch.com
http://searchenginewatch.com/reports/article.php/2156481

Appendice A

CODICE DEI PARSER

About (www.about.com)

```
l <font color=#CC0000>Displaying
+ 30
l  
+ 6
g FOUND
u </font>
l <a href="
+ 9
g URL
u ">
g TITLE
u </A><BR>
c
g TEXT
u <br><span
c
p
l <br><br>
+ 8
j 6
```

AllTheWeb (www.alltheweb.com)

```
l <span class="resTitle">
+ 23
l **http%3a//
+ 11
g URL
w http://
u " >
g TITLE
u </a>
c
g TEXT
u </span><br><a
c
```

```
p
j 0
```

Altavista (www.altavista.com)

```
l <br class='lb'><a class='res' href='
+ 36
g URL
u //
i 200 =http%3a
j 10
l **http%3a
+ 9
u '>
j 15
l =http%3a
+ 8
u %26
l '>
+ 2
g TITLE
u </a>
c
g TEXT
u <span class=ngrn>
c
p
j 0
```

Aol (search.aol.com)

```
l <span class="small_light">Page
+ 30
l of
+ 3
g FOUND
u </span>
l <li class="resultItem">
l onmouseover="self.status='
+ 26
g URL
u '
l >
g TITLE
u </a>
c
l </a>
+ 9
g TEXT
u <span class="small url">
c
p
j 6
```

Espotting (www.espotting.com)

```
l </script><form name="frmResults"
l <td colspan="2" class="title"><a class="title"
+ 46
l >
+ 1
g TITLE
u </a>
l <td colspan="2" class="description">
+ 37
g TEXT
u  </td>
l <a class="url"
l >
+ 1
g URL
u </a>
p
j 1
```

FindWhat (www.findwhat.com)

```
l <table width="100%" border=0 cellspacing=0
cellpadding=0 bgcolor="#ffffff" align="center">
+ 90
l <font face=arial size=3><b>
+ 27
g TITLE
u </b>
c
l <font face=arial size=2>
+ 24
g TEXT
u <br><i>
c
g URL
u </i>
p
j 2
```

Gigablast (www.gigablast.com)

```
l </form><b>Results
+ 17
l about
+ 6
g FOUND
u </b>
l <font size=+0>
+ 14
g TITLE
u </a>
c
g TEXT
u <font color=#008000>
c
g URL
w http://
u </font>
p
j 6
```

Google (www.google.com)

```
l <font size=-1 color=#000000>Results
+ 35
l </b> of
l <b>
+ 3
g FOUND
u </b>
l <p class=g><a href=
+ 19
g URL
u >
g TITLE
u </a>
c
i 100 <font size=-1> - [
l </font>
l <font size=-1>
g TEXT
u <font color=#008000>|<a class=fl
c
p
j 7
```

LookSmart (www.looksmart.com)

```
l <strong>Web results</strong>
l of
+ 3
g FOUND
u )
l <li><a href="
+ 13
g URL
u ">
g TITLE
u </a>
c
g TEXT
u <span class="dU">
c
p
j 5
```

Mozdex (www.mozdex.com)

```
l (out of
+ 8
g FOUND
u  total matching documents)
l <b>Internet Links: </b>
l <div>
l <a href="
+ 9
g URL
u ">
g TITLE
u </a>
c
l <br>
+ 4
g TEXT
u <font color=#996600>
c
p
j 5
```

Msn (www.msn.com)

```
l <span class="i">Results
+ 23
l of about
+ 9
g FOUND
u
l <li><a href="
+ 13
g URL
u "
l >
+ 1
g TITLE
u </a>
c
g TEXT
u <span class="u">
c
p
j 6
```

Overture (www.overture.com)

```
l <li style="margin-left:20px;"><b><a href=http://
+ 48
l yargs=
+ 6
g URL
w http://
u  target=_top>
w /
g TITLE
u </a>
c
g TEXT
u <br><em>
c
p
j 0
```

Teoma (www.teoma.com)

```
l <span class="bold">Showing
+ 26
l about
+ 6
g FOUND
u :
l <div id="result"
l <a href="http://
+ 33
l http://
g URL
u "
l >
+ 1
g TITLE
u </a>
c
g TEXT
u </div> <span class="baseURI">
c
p
j 6
```

Yahoo (www.yahoo.com)

```
l <font style="color:#c00;font-size:1.05em">
+ 42
g FOUND
u </font>
l <li><a class=yschlp href="
+ 30
l http
g URL
u ">
g TITLE
u </a>
c
l <br>
+ 4
g TEXT
u <br>|</em>|<div
c
p
j 4
```