Building a Search Engine to Drive Problem-Based Learning

Steven Bird
Department of Computer Science
and Software Engineering
University of Melbourne, VIC 3010, Australia
sb@csse.unimelb.edu.au

James R. Curran
School of Information Technologies
University of Sydney, NSW 2006, Australia
james@it.usyd.edu.au

ABSTRACT

Search engines pervade the digital world, mediating most access to information instantaneously. We have found that students can build search engine components, and even entire search engines, in the context of problem-based learning in introductory and intermediate computer science courses. The courses cover a broad range of topics in algorithms, data structures, and web design, with a heavy emphasis on programming. Additionally, the internet is coupled with the syllabus at many places, from web design and HTML to graph algorithms and pattern matching. This connection enlivens the discussion of otherwise dry topics like searching, sorting, indexing and hashing. Moreover, the challenge of web-scale computing motivates the continuing students in their later study of formal topics like algorithmic complexity, while non-continuing students acquire transferable analytical skills. We report on the experience in search engine projects for driving problem-based learning in computer science courses, for both high school and university students. Our experience shows that such projects are effective in both introductory and intermediate courses, and readily encompass student groups with diverse programming abilities.

Categories and Subject Descriptors

K 3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms

Algorithms, Data Structures, Digraphs

Keywords

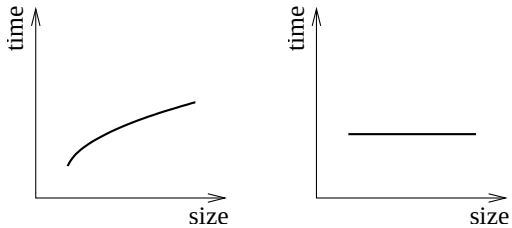
Python, Web, Google

1. INTRODUCTION

Simple demonstrations quickly convince students that searching and sorting take longer as input size grows (Figure 1(a)). Yet web-search latency has stayed constant despite the explosive growth of the web (Figure 1(b)). What is going on here?

Most computer science students take web search engines for granted. They never experienced the internet prior to search engines, and have seldom pondered how search engines work. Accordingly, the graphs in Figure 1 are a source of cognitive dissonance, grabbing students' attention and provoking them to wonder if they can apply their presumed computational know-how to explain this real-world observation. Pursuing this further, we conceive of a larger challenge, one which can drive students to learn many fundamentals of computer science, namely to implement some search engine components themselves, or even to build an entire search engine. The apparent magic of search engines disappears, but students are left in control of a "deeper magic." Applying computer science concepts and methods, they build simple software components, then marvel when many interlocking components work in concert to produce sophisticated and informative behaviour. The experience of delivering a whole system which is more than the sum of its parts turns out to deliver something else as well: a fundamental grasp of core topics in computer science.

Of course, the value of integrating the web into computer science courses is often noted (e.g. [4]). However, in most introductory courses the web serves as a *secondary* source of information about course content, rather than a *primary* source of raw data for student projects. We have found that high school students are excited by the novelty of writing programs that venture out onto the web. These programs access remote sites, and may behave unpredictably as the sites change. Tasks like extracting URLs, following



(a) Phonebook (b) Google

Figure 1: Search Time as Data Size Grows

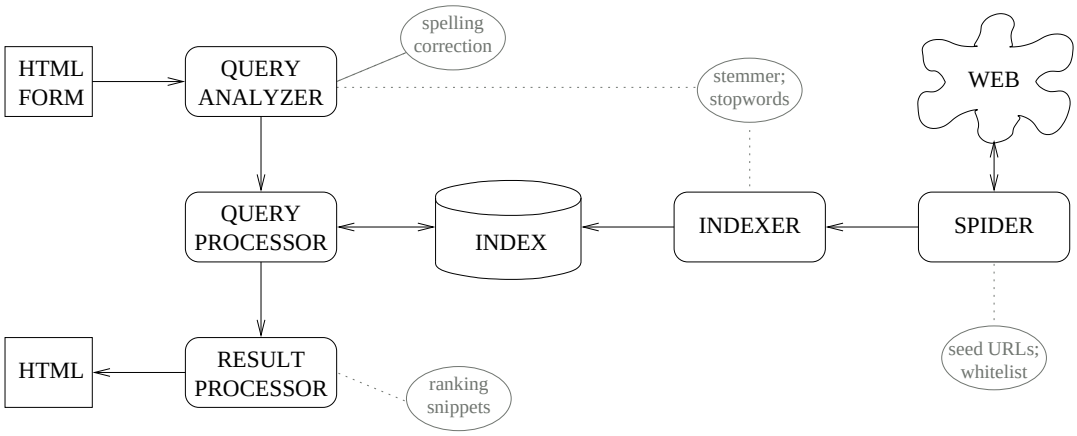


Figure 2: Search Engine Architecture

hyperlinks, indexing content, processing queries, and ranking search results all require students to grasp core concepts in computer science. Putting everything together gives students their first authentic experience of a software development team and of multi-component system-building. Similarly, we found that second-year university students were highly motivated to learn about algorithms and data structures thanks to two web-inspired projects: implementing Google PageRank [2] and spelling correction. A far cry from clicking hyperlinks, the use of the web in these problem-based learning environments was fundamental to achieving the desired educational outcomes.

In this paper we report on experiences in the use of web search engine projects to drive problem-based learning in computer science courses. This involves two groups: high-school students attending a week-long intensive summer school at the University of Sydney, and second-year students taking an introductory algorithms class at the University of Melbourne. Both courses shared three goals: (i) broad coverage of computer science with an emphasis on programming; (ii) integration of the web into the syllabus; and (iii) stimulation of both continuing and non-continuing students alike. Subsidiary goals were to attract (or retain) computer science students, and to explore problem-based learning [1] as a means of addressing diverse student populations.

This paper is organised as follows. We begin by outlining the structure of a simple search engine (§2). The following two sections report on our experience teaching introductory courses at the Universities of Sydney (§3) and Melbourne (§4), and provide objective and subjective evaluations.

2. ANATOMY OF A SEARCH ENGINE

The core of a search engine is a large inverted index, identifying all pages where a given word appears. Users access this index via a query interface, then a query processor looks up their search term(s) in the index and ranks the results before delivering them back to the user. Building the index itself is a laborious process. A program – aptly called a *spider* – periodically crawls along the hyperlinks of the web, finding documents to be indexed. See Figure 2 for a diagram showing how these components fit together. If the basic idea is simple, the possible additions are numerous and vary widely in complexity. Some popular additions are indicated in grey in Figure 2. We consider these components in turn below, along with the corresponding theoretical concepts plus an indication of the level of difficulty for implementation using asterisks (*=easy, ***=difficult).

Spider***: navigates the web to collect web pages and extract text content; Concepts: queues, breadth-first search in directed graphs, regular expressions for extracting URLs and stripping HTML markup.

Seed URLs and Whitelist*: lists of URLs defining the starting URLs and the websites to be crawled. Concepts: site topology, research on a topic area.

Indexer**: tokenizes the text of each page, normalizes the words, and stores them in an associative array which maps words to document ids; optionally removes stopwords and performs stemming; Concepts: regular expressions, associative arrays, persistent storage.

Stopwords*: a list of words optionally removed from the input documents by the indexer, obtained by downloading several large text files from Project Gutenberg,[1] and identifying the most frequent words (e.g. *and*, *the*). Concepts: tokenization, frequency counting, sorting, manual post-editing.

Stemmer**: strip suffixes to leave word stems (e.g. *walking*, *walked* and *walks* should all be indexed as *walk*); Concepts: regular expressions, knowledge of English word-formation rules.

Website*: migrate content from an existing site, re-organizing material and applying a new design, creating a hypothetical site for hosting the search service; Concepts: HTML, CSS, validation, web design, accessibility.

Query Analyzer and Processor**: split query string into search terms and optional operators (e.g. OR, NOT), optionally correct spelling errors (§4), probe index to get candidate document ids for each query term, perform unions, intersections and relative complements; Concepts: CGI processing, Boolean algebra.

Result Processing**: probe index using document ids to retrieve full URLs and document titles, and output in HTML for rendering in a browser, optionally generating snippets from indexed documents, and optionally ranking using term frequency or Google PageRank

[1] `http://www.gutenberg.org/`

(§4); Concepts: programmatic generation of HTML, term weighting, directed graphs, breadth-first search.

Snippets***: display text fragments showing the best document context covering the search term(s), using an augmented version of the index incorporating word offsets. Concepts: positional indexing, heuristics.

Although this is a considerable list of tasks, several can be omitted, and others can be made easier through the provision of pseudocode. If necessary, difficult tasks such as the spider can be provided as off-the-shelf components. Key features of the above set are the broad coverage of computer science topics, and the wide range of tasks geared to different skills and different levels of programming ability.

3. SUMMER SCHOOL UNIVERSITY OF SYDNEY

The National Computer Science School, now in its eleventh year, is an intensive one-week residential program for some 65 final-year secondary students together with a dozen secondary teachers. It is largely staffed by volunteers – past students donating a week of vacation – under the leadership of the second author.

Students from across the country are selected through an application process that includes a letter explaining their interest in the summer school, grades, teacher reference, plus a sample of work, such as program code or a website. Students are highly motivated, though few have any experience of building a system or working in a development team.

Students work in groups of fifteen, each group developing a complete search engine in the space of a week. All programming is done in Python, which is ideal in a context serving newcomers and experienced programmers, as also noted by [6]. Individual Python modules can be developed independently, and data structures (such as the web index) shared using Python's *shelve* methods. Students use a specially prepared Python text that emphasises features required for search engines [3]. Programming classes are highly interactive, and 75% of the time is spent experimenting with the Python interpreter projected onto a large screen. Other lectures cover HTML, CSS and web design (topics which had once been the sole focus of the summer school when it was established in 1995 by Judy Kay and Bob Kummerfeld).

On the first two days, laboratory sessions focus on learning Python. From the third day onwards, groups split into two sub-groups, one focussing on web design (HTML and CSS) and one on building the back-end. This leaves about eight programmers in each group, consisting of the technically strongest students. The systems are presented and demonstrated on the final day.

At the outset, the students are given a high-level description of the components of a search engine and their interconnections, along with the skills and effort required to build them. The members of each group volunteer to work on different components, resulting in teams of 1–4 students. The tutor guides the process using knowledge of the programming aptitude of each individual, gained from observing them working on the initial Python tutorial materials.

The level of detail of the instructions is adjusted according to the strengths of each team. The description of the tasks is usually sufficient for the students to work out most of the architecture of Figure 2 for themselves. Teams focus on individual components and successively refine their understanding of the required inputs and outputs. Teams negotiate periodically to hammer out the details of the interfaces (i.e. each arrow in Figure 2). Frequent intensive discussions within and between the teams clarify concepts and get clear separation of functionality. A whiteboard is updated with the current architecture, tasks, interfaces, and schedule. Through all of this, the tutors play a largely non-directive role, being continually on call to advise students on conceptual and practical matters.

Evaluation

During the summer school, the first author served as a tutor for one of the groups, and periodically visited the other groups, interviewing many students and tutors. The tasks were pursued energetically, and the excitement of the students was palpable. The first goal, namely broad coverage of computer science with an emphasis on programming, was amply met for those group members who did programming tasks. However, this judgement is subjective as no formal assessment was undertaken. It was notable that students of widely differing programming ability could pursue interesting tasks at an appropriate level of difficulty. The web authoring tasks provided useful occupation for the students who discovered that they were not interested in programming. The second goal, integration of the web into the syllabus, was also met. Finally, it was clear that both continuining and non-continuing students were challenged at their level of ability.

The greatest challenge for all groups was collaborating in a team to build a multi-component system. Even the most capable programmers had no prior experience of software development teams or multi-component system-building. Many of them struggled with this, and the tutors' role was as much social as technical. For instance, one student produced a second spider for his team *fait accompli* and proceeded to argue that it was better than the one that had already been developed by another student. After heated debate it was agreed that the original spider had better regular expressions, while the new spider was cleaner program code, and they were able to combine the two. Such teamwork experiences provided students with a valuable insight into the information technology industry, and were better equipped to choose a suitable university course.

The choice of Python was critical to the success of the summer school. Being interpreted facilitated experimentation with the language in the early part of the week, and rapid prototyping later on. Python's transparent syntax made it easy for students to share code, and its lightweight object-orientation facilitated component integration. Notably, Python's *shelve* methods were ideal for saving associative arrays to disk, avoiding the need for explicit file I/O. Thus the spider and indexer could be written independently, and later integrated via a shared file on disk.

On the last day of the summer school, students completed a survey which included questions geared to the topic of this study: 87% said their programming ability is greatly improved; 97% agreed that building a search engine is an interesting way to learn about computer science; and 70% said that their interest in doing tertiary study in computer science has increased significantly.

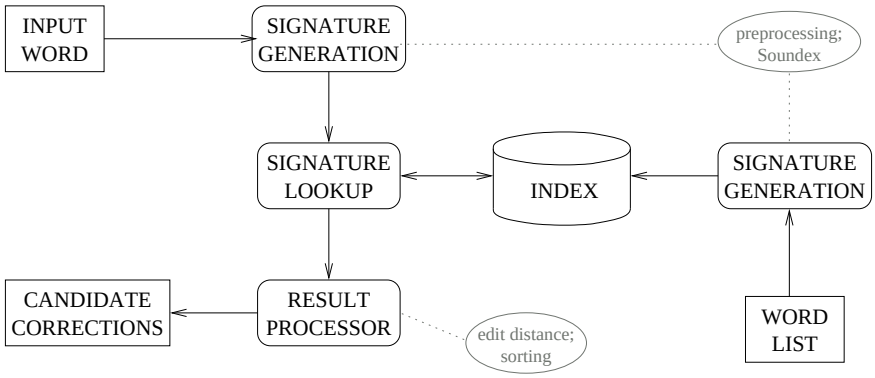


Figure 3: Spelling Correction, including Approximate String Matching

4. ALGORITHMS AND DATA STRUCTURES UNIVERSITY OF MELBOURNE

Algorithms and Data Structures is a second-year course in the Department of Computer Science and Software Engineering at the University of Melbourne. The course covers 80% of the material in Levitin's text [5], and Python is used in lectures to demonstrate algorithms (often using code downloaded from Wikisource). After identifying the most basic operation performed by the program, such as addition or key comparison, the code is instrumented to report the number of times this operation is performed. This facilitates experimentation and discussion in the class. In 2005 the course included two C programming projects motivated by tasks performed by web search engines: Google PageRank and spelling correction. Both tasks involved large quantities of realistic data, and both were incompletely specified.

PageRank: This topic was selected thanks to its potential for students to learn several graph algorithms. PageRank is Google's method for scoring the importance of web pages, used for ranking search results.[2] A hyperlink from some page p to another page q is considered as a *vote* by p in favour of q. Those pages with many incoming links are ranked more highly, and (on the next iteration) cast more influential votes on the importance of other pages. The algorithm is simple enough, but requires students to code an incidence matrix and perform computations on a directed graph.

```
d = 0.85
score = [1.0, 1.0, ..., 1.0]
repeat 50:
    next_score = [1-d, 1-d, ..., 1-d]
    foreach page:
        out_degree = count outgoing links (or target_pages)
        contribution = score[page]/out_degree
        foreach target_page
            next_score[target_page] += d * contribution
    score = next_score
```

In preparation for their work on PageRank, students implemented brute-force algorithms to identify the *roots* of the webgraph (pages from which all other pages are accessible), and the *hubs* (pages having a large number of outgoing links). Roots and hubs are easy to identify but, as students discovered, not a very useful source of information. The next task was to identify the *authorities* using PageRank.

[2] `http://www.webworkshop.net/pagerank.html`

The project used webgraphs of three sizes, an artificial site with 5 pages, the department's course advice collection with 60 pages, and a local news and entertainment website with over 3,000 pages. We used the `w3mir` utility to save the incidence matrix of a website to a file as part of the process of mirroring the site. Student projects were assessed individually, and the available marks were equally divided between the following: (i) operational program; (ii) correct implementation; (iii) written report containing a complexity analysis of the three algorithms.

Spelling Correction: Students had already experienced spelling correction in the context of word processors, and also when searching using Google. This project taught students about string processing (Soundex), associative arrays (the soundex index) and dynamic programming (Levenshtein's algorithm), and how a linear algorithm can be used to save work for a quadratic algorithm. The systems developed by the students were based on the architecture given in Figure 3, described below:

- Signature generation: supplied Soundex code converts a word to a signature (e.g. *giant* is mapped to `G530`); multiple words having similar pronunciation are mapped to the same signature; Concepts: integrating third-party components, string processing.
- Signature lookup: probe the index using the signature of a mis-spelled word to find candidate corrections; Concepts: associative arrays.
- Result processor: score candidates for similarity to the input word using Levenshtein edit distance, then sort and return the 10 most similarly spelled words; Concepts: dynamic programming.
- Preprocessing: collapse spelling distinctions to make up for Soundex shortcomings (details below); Concepts: improving performance of black-box component by preprocessing its input; research methods for identifying common mis-spellings.
- Sorting: sort candidates by similarity score; Concepts: sorting based on non-key value.

Systems were coded in C, and a major part of the task was to implement dynamic hashing with linked-list values, sorted using insertion sort. The Levenshtein algorithm was provided in pseudocode, and needed to be translated into C. The large data size (45,000 correctly spelled words) ensured

the hashing code was thoroughly tested. Students learned how to adjust program parameters to improve performance.

The coding task was somewhat open-ended, in the sense that students had to address shortcomings of Soundex, e.g. pronunciations of words involving silent letters (*thumb* and *thum* have different signatures); and transpositions of consonants (*algorithm* and *aglorithm* have different signatures). Students investigated these shortcomings and described three problems of their own. They fixed these problems by preprocessing the words. For example, if *jiant* was considered to be a possible mis-spelling of *giant*, the spelling distinction would be collapsed by preprocessing all strings, replacing *j* with *g* (or perhaps just *ji* with *gi*), before submitting them to soundex. This way, both spellings would be mapped to soundex code `G530`. Many students agreed to demonstrate this aspect of their system to the whole class (90 students).

As for the first project, a third of the assessment was based on a written report in which students discussed the complexity of the algorithms. Students were also asked to speculate on how they might extend the program further to cope with three additional shortcomings:

keyboarding errors: soundex was intended for pronunciation errors, not typing mistakes;

word frequency: words were ranked by edit-distance, but common words are more likely corrections than rare words; e.g. the input gentel would be more likely to be a mis-spelling of the high-frequency word gentle than the low-frequency word genteel, but edit-distance would order genteel first;

affixes: the English lexicon is an open set, not closed as the existence of a fixed words file might suggest; we can easily form new words using existing prefixes and suffixes.[3]

Evaluation

The most significant change in the course from previous offerings was the use of project work inspired by search engines. The tasks involved large data sets (e.g. a directed graph with 3,000 nodes) enabling the students to experience the difference between an n^3 algorithm (identifying roots) and an n^2 algorithm (identifying hubs), and therefore appreciate the significance of algorithmic complexity. Projects were not tightly defined and somewhat open-ended. Students seemed to enjoy the unpredictability of the programs, whose performance and results depend on the topology of an external website, or on the unpredictable input of misspelled words.

During the project periods, students frequently complained about the requirement to use the C programming language, a choice imposed by the department. The most common pitfalls in the projects were not conceptual, but due to memory management errors. The C language is evidently too low-level and detracts from the learning of high-level concepts in algorithm design.

On the university administered quality-of-teaching survey, students rated several facets of the subject on a scale of 1–5. For the statement *"I found this subject intellectually stimulating"* the mean score was 4.2. This compares favourably with scores over the past four years (mean=3.4, standard deviation=0.2). Free-text comments were positive about the project, e.g. *"project topics were well chosen/interesting"*; *"projects were heaven"*; *"projects were challenging but not impossible"*; *"projects ended up being the main motivation"*. An exception was the small number of predictable complaints about the projects being insufficiently well-defined.

[3] `http://en.wikipedia.org/wiki/List_of_English_prefixes`
`http://en.wikipedia.org/wiki/List_of_English_suffixes`

5. CONCLUSIONS

Many core topics in computer science are taught with the aid of toy-sized problems using artificial data. The web is most often cited as a secondary source of information on course content. However, the webgraph is an interesting data structure in its own right, a directed graph with trillions of nodes. We have used search-engine inspired projects to drive problem-based learning in introductory and intermediate computer science courses, at secondary and tertiary level. We have found that it is possible to devise many interesting student projects which treat the web as a primary source of raw data. These projects were an effective way for students to learn otherwise dry topics such as directed graphs, dynamic programming, and algorithmic complexity.

Significantly, all students appeared to benefit from our approach. Students not planning to major in computer science were given an appealing snapshot of the field, experiencing algorithms in an interesting context, and gaining new understanding of the structure of the web. Students planning to major in computer science were also well-served, learning several graph algorithms, and gaining an appreciation for the practical importance of algorithmic complexity. Furthermore, we found it was quite straightforward to split the search engine project across several programming teams, according to the abilities and interests of the students.

The cognitive dissonance of Figure 1 is not resolved. The logarithmic growth of the internet compared to the exponential growth of Moore's law suggests that the Google search time should *decrease* over time. The real story requires an understanding of parallel computation. Students have to come back next year for the full answer.

6. REFERENCES

[1] D. Boud and G. E. Feletti, editors. *The Challenge of Problem-Based Learning*. London: Kogan Page, 1997.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[3] J. R. Curran. Building a Python search engine. School of Information Technologies, University of Sydney.

[4] T. Hickey, A. Kumar, L. Wilkens, A. Beiderman, A. Mahadev, and H. Ellis. Internet-centric computing in the CS curriculum. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 50–51. ACM, 2002.

[5] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison Wesley, 2003.

[6] C. Shannon. Another breadth-first approach to CS I using Python. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 248–251. ACM, 2003.