Final Project

CS 2420

Lia Bogoev & John Call

Introduction

Our final project for CS 2420 was implemented in Python. In addition to completing all of the requirements, we also included some bonus features. The search engine has a ranking algorithm, and returns results sorted by relevancy. It can accommodate extended combinations of boolean operators, including all combinations of AND, OR, and NOT. It eliminates stop words for efficiency. All of these features come packaged in a fancy interface that was implemented using JavaScript, HTML/CSS, and a dash of PHP.

Algorithms

The three main factors that influence the speed of our engine are the number of words in documents (W), the number of documents (D), and the number of words in the query (Q). The size and quantity of documents affects the running time due to the process of reading each file for indexing. The primary data structure that we use is a dictionary (basically a hash table). We use a dictionary for each document, to track the number of occurrences of each word in the given document. We also use a dictionary of arrays, which contain all words in all the documents (excluding stop words) and an array of documents which contain said word as key value pairs. Filling these arrays is $O(s)$, based on the quantity and size of the documents.

The algorithm that we use to address the boolean operators in the search query (parseQuery) is recursive. Because it must iterate over each word in each document for each part of the search query, in the worst case, it is $O(Q \bullet W \bullet D^2)$. Given that the majority of queries will not be very complicated, this is approximately $O(W \bullet D^2)$, which is not the fastest, but is not noticeably slow until the number of documents or the diversity of words within the various documents increases. The code itself is clean, readable and functions well. In larger scales, a more robust method involving a database engine, perhaps such as MySQL, would clearly be desirable.

We also implement document ranking. The algorithm considers two factors: Word Frequency (how often a word from the query appears in the document), and Document Frequency (how often a word appears in all documents). In principle, if a word from the search query appears in the document more times, then it is more likely to be the document the user is searching for. Conversely if the same word exists in many documents, then it is less specific to a given document and, therefore, it is less likely that particular word from the search query has an influence on what the user is searching for. The time to look up both of these factors from the index is *O(1)* because of our use of a hash table, making this portion of the algorithm very nice, indeed.

Instead of writing our own sorting method to sort the documents by ranking, we used Python's built in Timsort, because its time complexity is, in best case, better than both Quicksort and Mergesort: *O(n)*. The average and worst cases are equivalent for the Mergesort: *O(n • log(n))*. Timsort makes use of already sorted subsets of data, storing them in temporary arrays before merging them. The only potential hazard with Timsort is space complexity, which is the worst case is *O(n)*. With such a small scale project, however, space was not of much concern.

Design Issues

At first glance, handling the boolean operators of the search queries seemed trivial. Once we began to write the function, however, it quickly became apparent that we would need to either limit the number of search terms so that we could focus on one boolean operator at a time, or we would need to write a recursive function. We chose to go with recursion so that we could have flexibility in the number of search items. This required a restructuring of the existing parsing algorithm so that it could accommodate recursion.

Our parseQuery function initially takes the dictionary of all the words from the indexing function, and, treating spaces and punctuation as an "AND" operator, uses the "AND", "OR" and "NOT" operators in the query to cut and merge each set obtained from the hash table, returning a list of documents that match the search query. It's a pretty rudimentary implementation of the Set ADT functions.

Another design challenge was the delivery of our search engine. We knew the end result would be a web page, but as we looked at AWS it seemed to be much more sophisticated a service than we needed, making it difficult to configure it as we needed. Instead we setup a server with the basic necessities: apache, PHP, python 2.7.

We know that python is used by many web services and there are frameworks to help python integrate with apache, but similarly these seemed to be much more than we bargained for.

The result was we added a small PHP script to call the python script (since all we ever needed from apache was the POST array sent to the server asynchronously with JavaScript). This does, however, leave the minor issue that shell arguments are executed with user input. This is not the best design practice for a website and would not be advised for a production environment.

Optimization and Scalability

With regards to optimization, not much can be done to improve the indexing. In any situation, all of the words in a document and all of the documents themselves have to be indexed. However, as we've discussed in class, portions of the document could be indexed in parallel across multiple computers to reduce the time required.

Within the searching part of the algorithm, however, some optimization could be achieved with dividing and conquering the search query. The parseQuery function has a worst case of $\approx O(Q \bullet W \bullet D^2)$, if the query is comprised of common words that are present in many documents. If the algorithm could be broken up in such a way that common words are treated differently than specific words, then the overall time complexity could be reduced.

There are some things that need to be addressed due to the nature of web browsers and their interaction with the server. Our website asynchronously loads content based on the results of a given query. As the results get larger, the resources and time required by the browser to dynamically generate and insert that HTML content is significant. The first optimization to be performed, therefore, would be to implement a pagination feature. Similar to how Google only shows a limited number of search results on the first page, and results after that are unlikely to be what you're looking for regardless.

GREETINGS, STRANGER.
WHATEVER QUEST DRIVES YOU, ABANDON IT.
YOU SHALL FIND NO ANSWERS IN THESE DESOLATE WASTES.
I KNEW I WOULDN'T.
I GUESS I ... JUST HAD TO SEE.

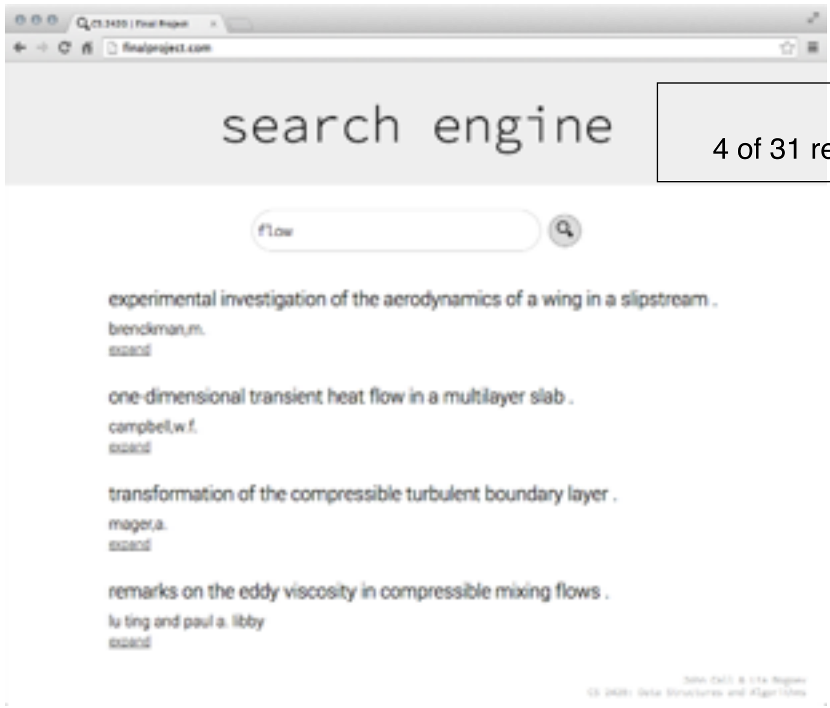
I HATE FEELING DESPERATE ENOUGH TO VISIT THE SECOND PAGE OF GOOGLE RESULTS.

Source: http://xkcd.com/1334/

The second optimization that needs to occur based on the nature of a web server is that the indexing loads with each query, after which memory is released back to the operating system once this script is finished executing. In an ideal scenario our indexer would remain active as long as the server is on, or better yet store the index to the file system for long term usage, and wait for requests from the web server. This requires the use of sockets, or other methods, that we do not yet have much practice in implementing.

Graphic Design Notes

The face of any programming project inherently influences how people react to it. Aside from its incredibly complex and groundbreaking algorithms, Google's simple, minimalist design draws people in. Inspired by Google's originality, we implemented a clean HTML interface to showcase our search engine. We felt that showing the search query in context with highlighting would enhance the usefulness of our engine, and that the design as a whole elevates the experience of searching.



flow
4 of 31 results

search engine

flow

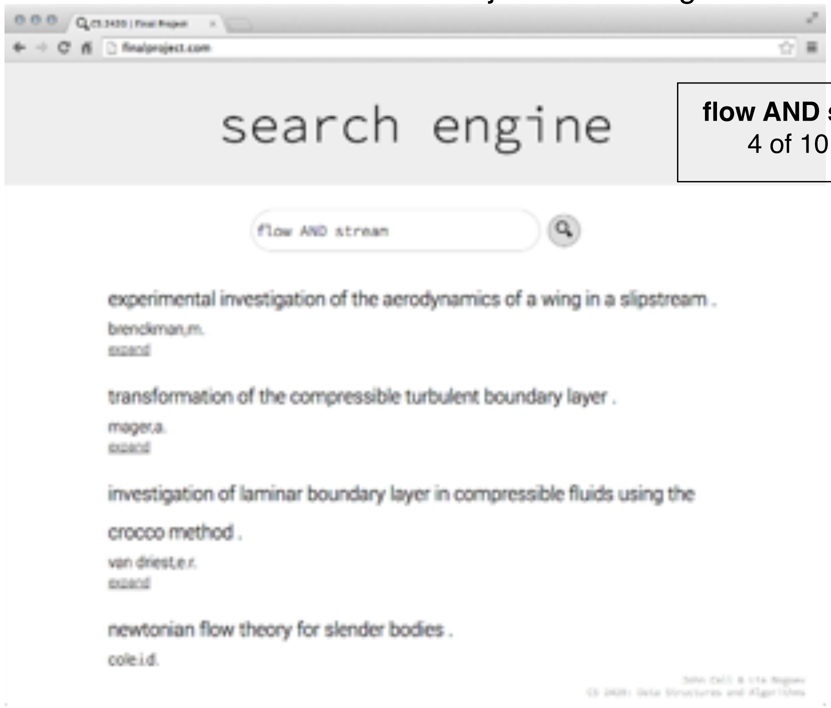
experimental investigation of the aerodynamics of a wing in a slipstream .
brenckman,m.
expand

one-dimensional transient heat flow in a multilayer slab .
campbell,w.f.
expand

transformation of the compressible turbulent boundary layer .
mager,a.
expand

remarks on the eddy viscosity in compressible mixing flows .
lu ting and paul a. libby
expand

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



flow AND stream
4 of 10 results

search engine

flow AND stream

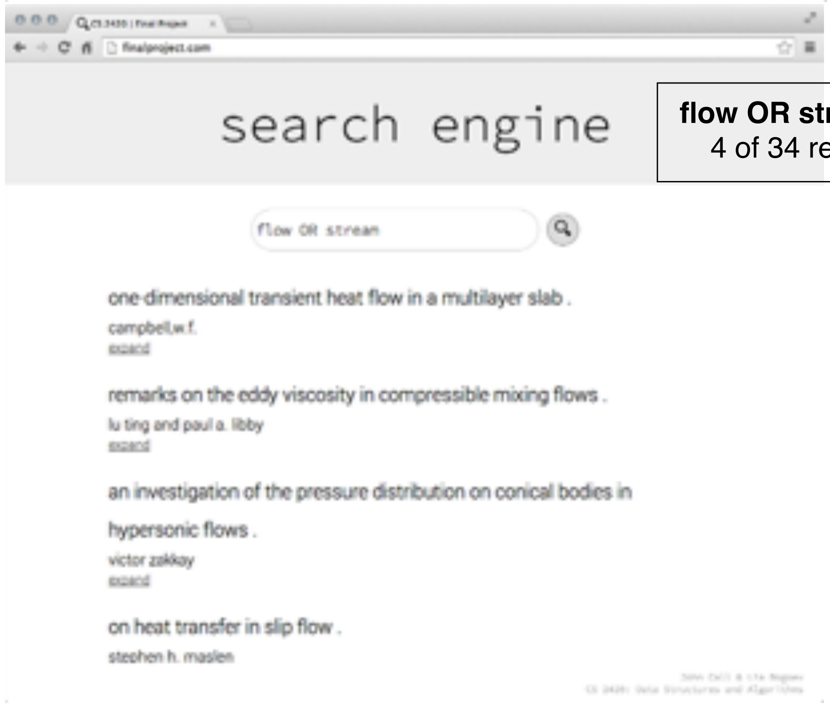
experimental investigation of the aerodynamics of a wing in a slipstream .
brenckman,m.
expand

transformation of the compressible turbulent boundary layer .
mager,a.
expand

investigation of laminar boundary layer in compressible fluids using the crocco method .
van driest,e.r.
expand

newtonian flow theory for slender bodies .
cole,i.d.

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



flow OR stream
4 of 34 results

search engine

flow OR stream

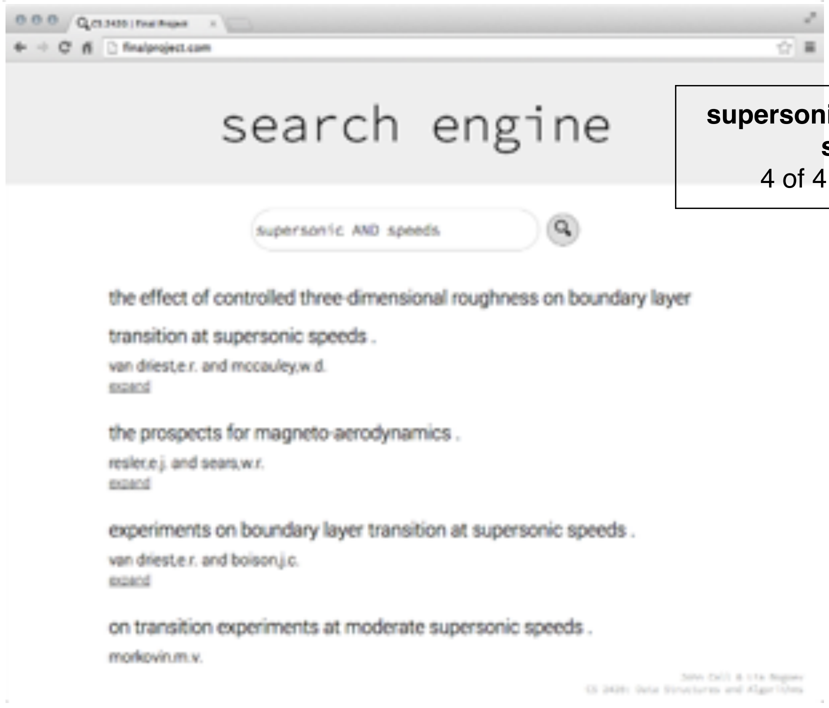
one-dimensional transient heat flow in a multilayer slab .
campbell,w.f.
expand

remarks on the eddy viscosity in compressible mixing flows .
lu ting and paul a. libby
expand

an investigation of the pressure distribution on conical bodies in hypersonic flows .
victor zakkay
expand

on heat transfer in slip flow .
stephen h. maslen

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



supersonic AND speeds
4 of 4 results

search engine

supersonic AND speeds

the effect of controlled three-dimensional roughness on boundary layer transition at supersonic speeds .
van driest,e.r. and mccauley,w.d.
expand

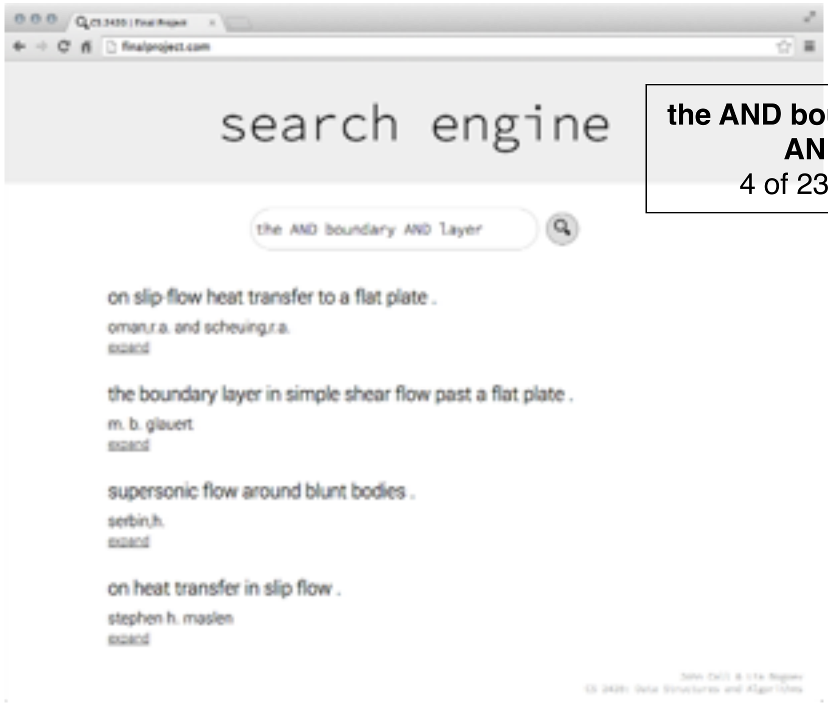
the prospects for magneto-aerodynamics .
resler,e.j. and sears,w.r.
expand

experiments on boundary layer transition at supersonic speeds .
van driest,e.r. and boison,j.c.
expand

on transition experiments at moderate supersonic speeds .
morkovin,m.v.

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms

Final Project — Lia Bogoev & John Call



the AND boundary AND layer
4 of 23 results

CS 2420 | Final Project

finalproject.com

search engine

the AND boundary AND layer

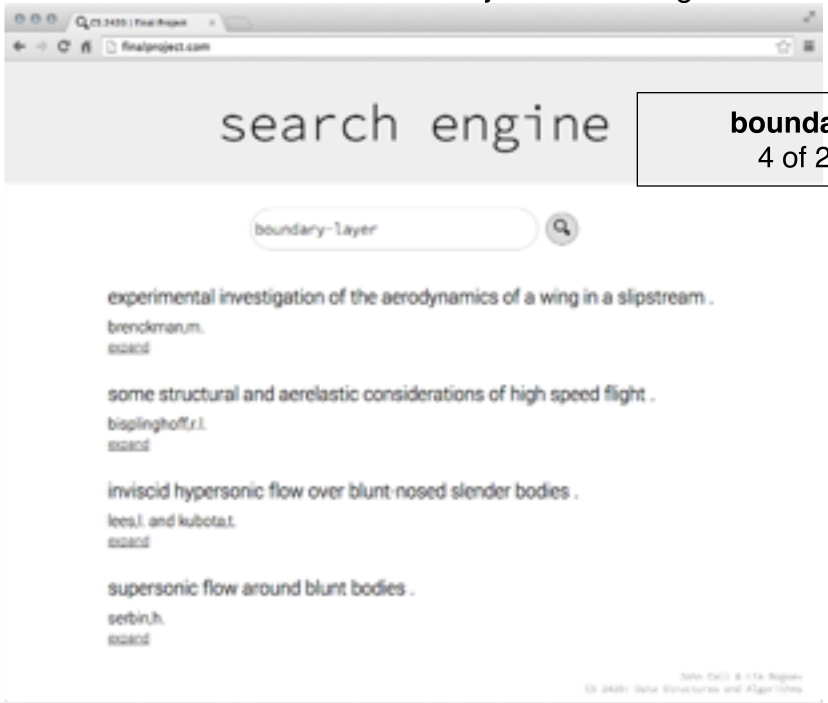
on slip-flow heat transfer to a flat plate .
oman,r.a. and scheuing,r.a.
expand

the boundary layer in simple shear flow past a flat plate .
m. b. glauert
expand

supersonic flow around blunt bodies .
serbin,h.
expand

on heat transfer in slip flow .
stephen h. maslen
expand

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



boundary-layer
4 of 23 results

CS 2420 | Final Project

finalproject.com

search engine

boundary-layer

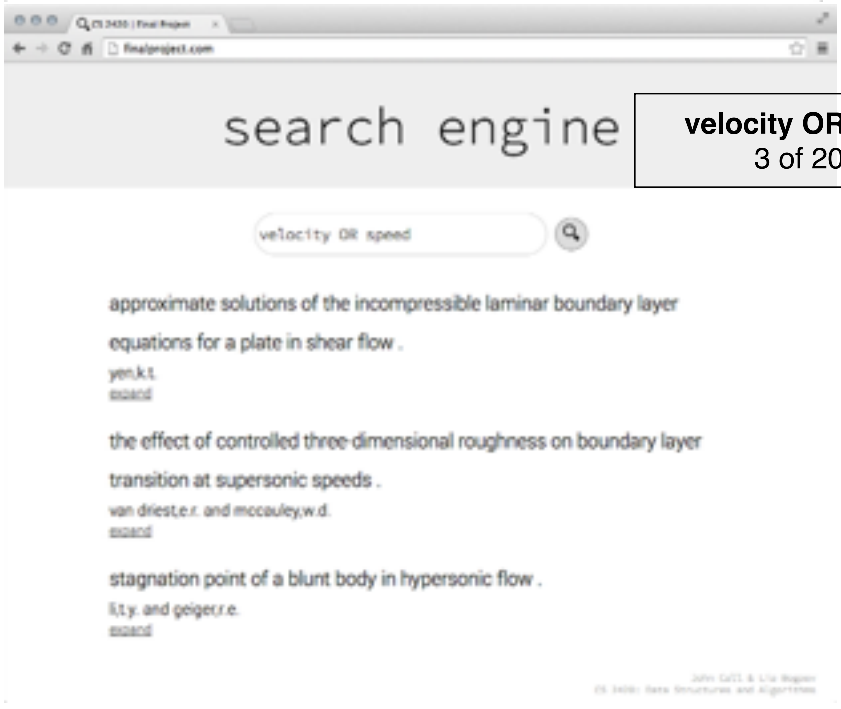
experimental investigation of the aerodynamics of a wing in a slipstream .
brenckman,m.
expand

some structural and aerelastic considerations of high speed flight .
bisplinghoff,r.l.
expand

inviscid hypersonic flow over blunt-nosed slender bodies .
lees,l. and kubota,t.
expand

supersonic flow around blunt bodies .
serbin,h.
expand

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



velocity OR speed
3 of 20 results

CS 2420 | Final Project

finalproject.com

search engine

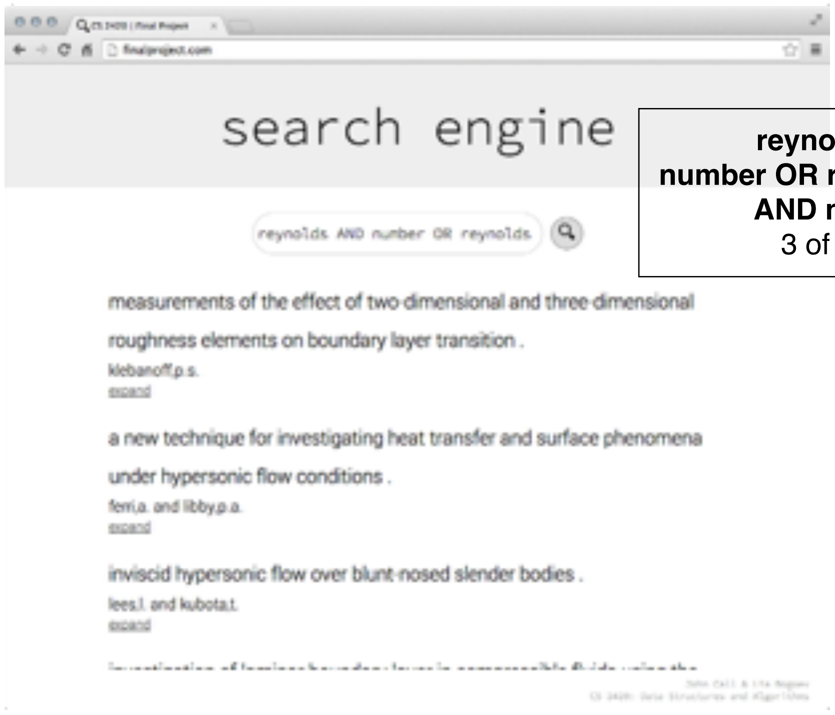
velocity OR speed

approximate solutions of the incompressible laminar boundary layer equations for a plate in shear flow .
yen,k.t.
expand

the effect of controlled three-dimensional roughness on boundary layer transition at supersonic speeds .
van driest,e.r. and mccauley,w.d.
expand

stagnation point of a blunt body in hypersonic flow .
li,t.y. and geiger,r.e.
expand

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms



reynolds AND number OR reynolds AND numbers
3 of 7 results

CS 2420 | Final Project

finalproject.com

search engine

reynolds AND number OR reynolds

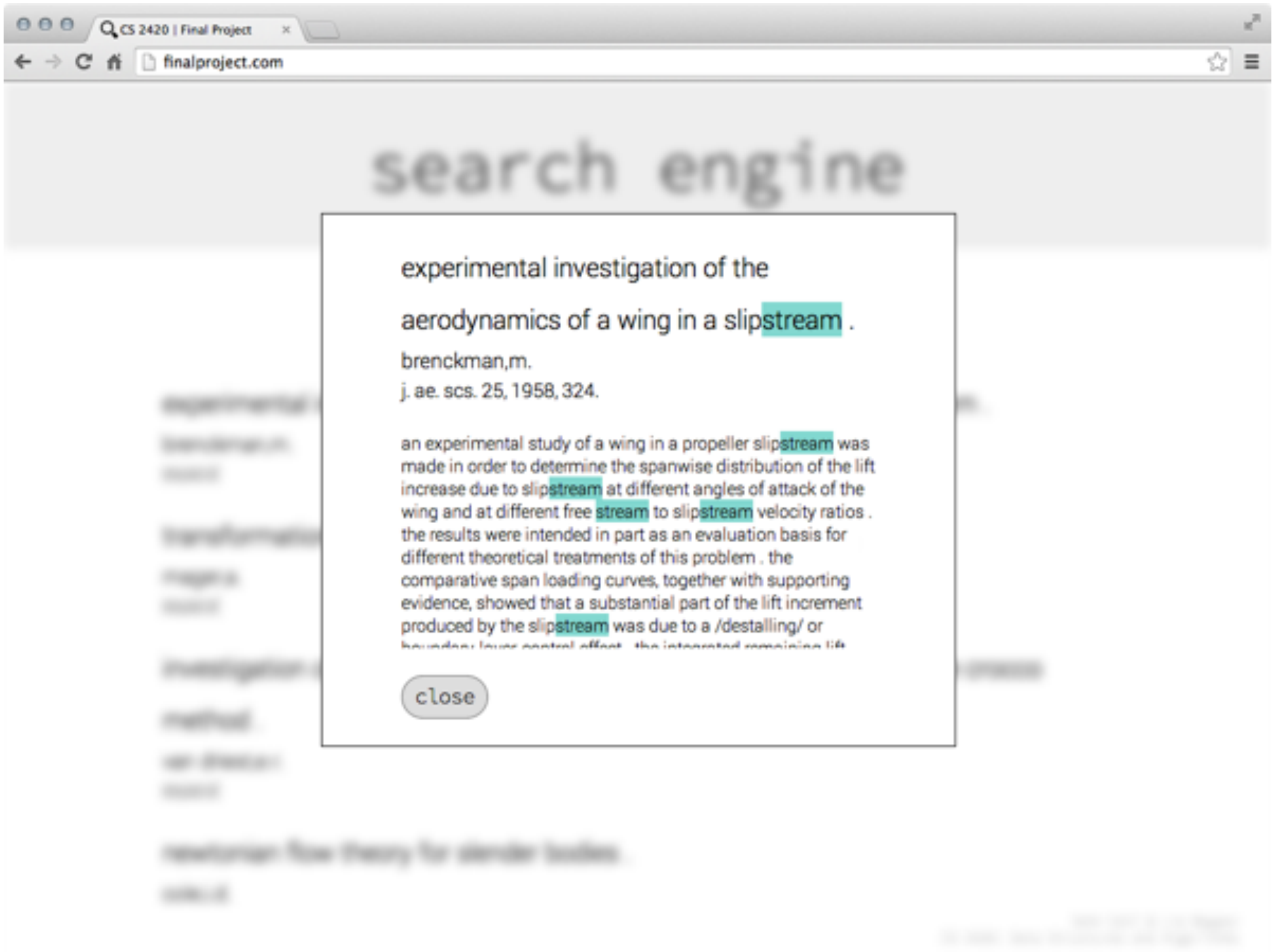
measurements of the effect of two-dimensional and three-dimensional roughness elements on boundary layer transition .
klebanoff,p.s.
expand

a new technique for investigating heat transfer and surface phenomena under hypersonic flow conditions .
ferri,a. and libby,p.a.
expand

inviscid hypersonic flow over blunt-nosed slender bodies .
lees,l. and kubota,t.
expand

John Call & Lia Bogoev
CS 2420: Data Structures and Algorithms

Keyword highlighting:



experimental investigation of the aerodynamics of a wing in a slipstream .

brenckman,m.

j. ae. scs. 25, 1958, 324.

an experimental study of a wing in a propeller slipstream was made in order to determine the spanwise distribution of the lift increase due to slipstream at different angles of attack of the wing and at different free stream to slipstream velocity ratios . the results were intended in part as an evaluation basis for different theoretical treatments of this problem . the comparative span loading curves, together with supporting evidence, showed that a substantial part of the lift increment produced by the slipstream was due to a /destalling/ or