

Efficient Text Search with Spatial Constraints

by

Dan (Amber) Han

TR14-233, August 18, 2014

This is an unaltered version of the author's MCS thesis
Supervisor: Bradford G. Nickerson

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566

Fax: (506) 453-3566

E-mail: fcs@unb.ca

<http://www.cs.unb.ca>

Copyright © 2014 Dan (Amber) Han

Abstract

This thesis presents a search engine called **TexSpaSearch** that can search text documents having associated positions in space. We defined three query types $Q1(t)$, $Q2(t, r)$ and $Q3(p, r)$ that can search documents with queries containing text t , position p and radius r components. We indexed a sample herbarium database of 40,791 records using a novel R*-tree and suffix tree data structure to achieve efficient text search with spatial data constraints. Significant preprocessing was performed to transform the herbarium database to a form usable by **TexSpaSearch**. We built unique data structures used to index text with spatial attributes that simultaneously support Q1, Q2 and Q3 queries.

This thesis presents a novel approach for simplifying polygon boundaries for packing into R*-tree leaf nodes. Search results are ranked by a novel modified Lucene algorithm that supports free form text indexing in a general way. A novel ranking of Q2 search results combines the text and spatial scores. The architecture of a prototype Java based web application that invokes **TexSpaSearch** is described. A theoretical analysis shows that **TexSpaSearch** requires $O(A^2 \overline{|b|})$ average time for Q1 search, where A is the number of single words in the query string t , and $\overline{|b|}$ is the average length of a subphrase in t . Q2 and Q3 require $O(A^2 \overline{|b|} + Z \log_{\mathcal{M}} \mathcal{D}_n + y)$ and $O(\log_{\mathcal{M}} \mathcal{D}_n + y)$, respectively, where Z is the number of point records in the list \mathcal{P} of text search results, \mathcal{D}_n is the number of data objects indexed in the R*-tree for n records, \mathcal{M} is the maximum number of entries of an

internal node in the R*-tree, and y is the number of leaf nodes found in range in a Q3 query.

Testing was performed with 20 test Q1 queries to compare `TexSpaSearch` to a Google Search Appliance (GSA) for text search. Results indicate that the GSA is about 45.5 times faster than `TexSpaSearch`. The same 20 test queries were combined with a Q2 query radius $r = 5, 50$ and 500m . Results indicate Q2 queries are 22.8 times slower than Q1 queries. For testing Q3 queries, 15 points were chosen in 15 different N.B. counties. The average T_c , T_s and T_e values of 191.5ms, 3603.2ms and 4183.9ms are given in the Q3 test, respectively, and the average value of $N_{pt} + N_{pl}$ is 1313.4.

Acknowledgements

My first and sincere thank goes to my supervisor Dr. Bradford G. Nickerson, who continuously and kindly supported all stages of my Master's thesis. I could not have imagined completing my thesis without Dr. Nickerson's guidance. He is a person open to ideas, and he was always available to shape my work.

I would like to thank the UNB Faculty of Computer Science for providing me many useful courses and resources. It would be hard for me to complete my thesis without the lab equipments they offered. I would also like to thank UNB Harriet Irving library for providing me the permission to use Google Search Appliance and the UNB Department of Biology for providing me the Connell Memorial Herbarium database for testing.

My greatest appreciation also goes to all my family members without whom I could not have made it here. Although they are far away, they always cheer me up and encourage me. Their emotional help inspired me to give my best.

Last but not least, I would like to thank all my friends whoever near or far, for giving me so much fun.

Table of Contents

Abstract	ii
Acknowledgments	iv
Table of Contents	viii
List of Tables	ix
List of Figures	xii
Abbreviations	xiii
1 Introduction	1
1.1 Suffix Trees	3
1.2 Background and Literature Review	5
1.3 Objective	7
2 A System Supporting Text Search with Queries Having Spatial Constraints	11
2.1 Methodology	11
2.2 Test Methodology	13
3 Data Preprocessing	16
3.1 UNB Connell Memorial Herbarium database	16

3.2	Transforming the collection	18
3.2.1	Work station on FCS network	19
3.2.2	Transform database records	20
3.2.3	Adding Spatial Information to the Records	22
3.2.3.1	Adding locations to the records	23
3.2.3.2	Adding polygons to the records	25
3.2.3.3	Polygon Simplification	26
3.2.3.4	PackPolygon algorithm can give a non-simple polygon for an input simple polygon	42
4	Indexing	45
4.1	Text Indexing	45
4.1.1	Suffix tree Leaf Nodes	46
4.1.2	Constructing a suffix tree	47
4.1.3	Querying a Suffix Tree	50
4.1.3.1	Reverse Document-Subphrases Hashtable	51
4.1.3.2	Ranking text search results	53
4.1.3.3	The Format for the Q1 Returned List	58
4.2	Spatial Index Using Packed R* Tree	58
4.2.1	R* Tree Leaf Nodes	59
4.2.2	Constructing an R* Tree	60
4.2.3	Querying an R* Tree	63
4.2.3.1	The Format for the Q3 Returned List	66
4.3	Combined Text and Spatial Query Q2	68
4.3.1	Data Structures for Maintaining Q2 Search Results	68
4.3.2	Combined Score for the Text + Spatial Search	71

4.3.3	Algorithm for Q2 query	72
4.4	Web Server architecture	75
4.4.1	Server Side Architecture	75
4.4.2	Client Side Architecture	79
4.4.2.1	Display Q1 Results	80
4.4.2.2	Display Q2 Results	81
4.4.2.3	Display Q3 Results	83
4.5	Source Code Summary	85
4.6	Search Complexity	85
5	Google Search Appliance (GSA) Indexing	90
5.1	PageRank Algorithm	90
5.2	Result Biasing	91
5.3	GSA operation	94
5.3.1	Configuring the GSA for Crawling and Indexing	94
5.3.2	Build Result Biasing Policy	98
6	Test Results	102
6.1	R* Tree and Suffix Tree Construction	102
6.2	Q1 Test Results	105
6.2.1	Comparing with GSA test results	107
6.3	Q2 Test Results	111
6.4	Q3 Test Results	116
7	Summary and Conclusions	124
7.1	Summary	124
7.1.1	Contributions	125

7.2	Conclusions	125
7.3	Future Work	126
	References	131
	A Stopwords list	132
	Vita	

List of Tables

3.1	The number of points for describing the county boundaries before and after the PolygonSimplification algorithm.	40
4.1	TexSpaSearch search engine source code summary.	89
6.1	Q1 test results for the 20 sample queries. All times are shown in ms.	106
6.2	The comparison of the TexSpaSearch engine and the Google Search Appliance (GSA) on the 20 sample queries. All times are shown in ms.	108
6.3	Time (in ms) per returned record for Q1 search results of TexSpaSearch and the GSA.	109
6.4	Q2 test results for the 20 sample queries used for the Q1 query tests, with radius 2m, 20m and 200m, respectively. All times are shown in ms.	112
6.5	Q3 test results for the 15 sample query points with radius 5m, 50m, 500m and 5000m, respectively. All times are shown in ms.	118

List of Figures

1.1	SPIRIT search engine architecture (from [38]).	2
1.2	A suffix tree for the string “BANANA” from [22]	4
1.3	An example database with records consisting of plain text and associated spatial information	8
1.4	Positions on Google Map of records in Figure 1.3	9
1.5	Examples of the three query types using database as shown in Figure 1.3.	10
2.1	A possible system architecture for text+spatial query.	12
2.2	An example for ranking method.	13
2.3	Testing system.	14
3.1	The preprocessing process for GSA index.	17
3.2	An example database record from the UNB Connell Memorial Herbarium database FileMaker version as viewed at http://herbarium.biology.unb.ca/fmi/iwp/cgi [21].	18
3.3	The tree structure under the directory http://131.202.243.11	20
3.4	IBM PC Extended ASCII Set code page 437, character 80 ₁₆ (128) through FF ₁₆ (255) [5].	21
3.5	Extended ASCII table for ISO-8859-1 standard, character 80 ₁₆ (128) through FF ₁₆ (255) [13].	22
3.6	Architecture of the Transformer.java program which is used to transform database records to required web pages.	23
3.7	An example of a generated web page 52686.htm.	24
3.8	The user interface for projection in ArcCatalog as used for transforming N.B. Stereographic Double projection coordinates to (ϕ, λ) pairs.	26
3.9	An example of how the Ramer–Douglas–Peucker algorithm works for a piecewise linear curve [7].	27
3.10	An example of the PackPolygon algorithm with the input polygon P containing 6 points and $w_B = 5$	31
3.11	Two examples of the PackPolygon algorithm. Figure (a) shows the worst case example. Figure (b) shows the example when the points are equally distributed along a circle.	35
3.12	Example showing the effect of the ShiftPolygon algorithm with $w_B = 6$	38

3.13	The comparison of the polygon data before and after simplification for counties of New Brunswick, Canada on Google Map. The map on the left side shows the original polygon data, the map on the right side shows the corresponding simplified data with $w_B = 500$	39
3.14	The comparison of the polygon data before and after simplification for the area 1 and 2 in Figure 3.10. The map on the left side shows the original polygon data, the map on the right side shows the corresponding simplified polygon data.	41
3.15	An example of the PackPolygon algorithm with the input polygon P containing 10 points and $w_B = 9$	42
4.1	The text + spatial indexing scheme using an R* tree and a suffix tree. . . .	46
4.2	An example suffix tree.	47
4.3	The structure of a suffix tree leaf node.	48
4.4	The sequence diagram for constructing a suffix tree.	49
4.5	The sequence diagram for a Q1 query.	51
4.6	The data structure for constructing the reverse document-subphrases hashtable.	52
4.7	An example of the generating of the reverse document-subphrases hashtable.	54
4.8	The data structure used in text search ranking process.	57
4.9	Data structures used in R* tree construction.	59
4.10	The sequence diagram for constructing an R* tree.	60
4.11	The sequence diagram for a Q3 query.	64
4.12	The data structure of the <code>RankedSpatialSearchResult</code> . Both L_1 and L_2 are stored in the structure. For L_1 results, <code>isPolygon</code> = “false” and <code>countyName</code> = \emptyset . For L_2 results, <code>currentPoint</code> = \emptyset	66
4.13	An example of a Q2 point results.	69
4.14	An example of a Q2 polygon results.	69
4.15	The data structures of the <code>TexSpaPointResult</code> and <code>TexSpaPolygonResult</code> . These are denoted as r_1 and r_2 in Alg. 4.2.	70
4.16	An example of the <code>TexSpaPolygonResult</code> when there is more than one disk arising from a Q2 text query intersecting a polygon. <code>Rqd</code> is the text score for a document as defined by equation (4.10) and <code>finalScore</code> is defined by equation (4.11).	72
4.17	The flow chart for Algorithm 4.2.	74
4.18	An example of a Q2(t, r) search with t = “Mc” and r = 0.6 km.	75
4.19	Architecture diagram of the <code>TexSpaSearch</code> web application server.	76
4.20	Lines 6-9 from <code>web.xml</code> file.	77
4.21	Three example queries sent to the TCP Socket.	78
4.22	Sequence diagram of server and client interaction for the <code>TexSpaSearch</code> engine.	80
4.23	The welcome page of the <code>HTMLHandler</code> web application.	81
4.24	Q1 search result for string “Carex magellanica”, with <code>listNum</code> = 5.	82

4.25	An example showing the link to display all 6 associated records for point 2: (45.905, -66.26) from a Q2 point result.	83
4.26	A sample of a Q2 point result showing two distinct points with distance 0 m to the Q2 string search results having primary keys 49846 and 58468, respectively.	84
4.27	An example of Q2 polygon results for query string “Carex magellanica” and radius 2m displayed on the web page.	84
4.28	An example of linking to display all the associated records for a Q2 polygon result, in county Sunbury.	85
4.29	An example of Q3 point results for point (46.888, -65.513) and radius 5000m displayed on a web page.	86
4.30	An example of Q3 polygon results for point (46.888, -65.513) and radius 5000m displayed on the web page.	87
5.1	The Metadata biasing user interface from a GSA model GB-7007-1M running version 6.8.0.G.30 of the GSA software.	92
5.2	The page Crawl and Index > Crawl URLs in the Admin Console of GSA software.	95
5.3	The page Crawl and Index > Collections in the Admin Console of GSA software.	96
5.4	The XSLT code which is used to associate <code>herbarium_collection</code> to the <code>test</code> front end.	97
5.5	The XSLT code which is used to associate <code>herbarium_collection</code> to the <code>test</code> front end.	98
5.6	An example query result from the <code>unb_herbarium_collection</code> with the query string “Asteraceae” using the <code>test</code> front end. Only the first five results are shown.	99
5.7	The user interface for configuring Source Biasing.	100
5.8	The user interface for configuring Data Biasing.	101
6.1	Several key points in the entire query process.	103
6.2	Several key points in the query process for subsequent queries sent by clicking a page number.	104
6.3	Search engine server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number N_r of returned search results for Q1 queries.	107
6.4	Search engine C++ processing time T_c , server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number of returned search results for Q2 queries $R_{pt} + R_{pl}$	117
6.5	Search engine C++ processing time T_c , server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number of returned search results for Q3 queries $R_{pt} + R_{pl}$	123

List of Symbols, Nomenclature or Abbreviations

Note that the following sections 1.2 uses symbols and notation as they appeared in the original author's articles, which means they are inconsistent with the List of Symbols, Nomenclature or Abbreviations.

A	the number of single words in a phrase
a_1	the first point in the sorted list <code>allPoints</code>
b	a subphrase
$ b $	the length of the subphrase b
$\overline{ b }$	the average length of subphrases in the query string t
B	the maximum number of data points contained in one leaf node in the R* tree
bb	a bounding square
B_b	the user-specified boost factor on a subphrase b
B_d	the boost factor for document d
B_t	the boost factor for query t
c	for each (ϕ, λ) pair, the average number of keys indexed
$\mathcal{C}1$	the time complexity for the Q1 query
$\overline{\mathcal{C}1}$	the average time complexity for Q1 query
$\mathcal{C}3$	the time complexity for the Q3 query
CSV	comma separated values
C_{td}	a score factor based on how many of the subphrases in the query string t are found in the document d
d	a document
D	set of records in the index
d_1, d_2 and d_3	objects of <code>DocObject</code>
D_b	number of documents containing the specific subphrase b
\mathcal{D}_n	the number of data objects indexed in the R* tree for n records
e_1, e_2 and e_3	the values of influence strength for tags in GSA Metadata Biasing
ϵ	tolerance
f	the general degree of influence that the GSA Metadata Biasing has
<code>finalScore</code>	the combined score for the <code>TexSpaSearch</code>

f_{max}	the maximum value of the general degree of influence that the GSA Metadata Biasing has
\mathcal{G}	a specific web page
\mathcal{G}_i	a web page
GIS	geographic information system
GRS84	1984 Geodetic Reference System
GSA	Google Search Appliance
I_b	inverse document frequency, $I_b = 1 + \log(\frac{n}{D_b+1})$
I_{chosen}	the index in the ordered point set P of the point we choose to add to P_{chosen} in one loop
I_{insert}	the index of the element right before I_{chosen} in P_{chosen} , so I_{chosen} is inserted between elements $P_{chosen}[I_{insert}]$ and $P_{chosen}[I_{insert} + 2]$
I_L	$P_{chosen}[I_{insert}]$
I_{min} and I_{max}	the indices for the two points having the greatest distance between them on the boundary of a polygon P
IR	information retrieval
I_R	$P_{chosen}[I_{insert} + 2]$
\mathcal{J}	the number of leaf nodes in the suffix tree of input string \mathcal{S}
JSON	JavaScript Object Notation
JSP	Java Server Pages
k	a loop in the PackPolygon algorithm
ℓ	an element in list L_1 or L_2
L	a polyline
L_1	sorted list for Q3InternalSearch point results
L_2	sorted list for Q3InternalSearch polygon results
L_b	subphrase length normalization, which is computed in accordance with the number of words in the subphrase b
L_l and L_r	the two polylines that the polyline L is divided into at point O
l_p	the maximum length of the primary key
L_s	a list of LatLngPairs, which stores the points that have already been used as inputs of the Q3InternalSearchs in the entire Q2 query process.
m	number of subphrases in the query string t
M	the point having the greatest perpendicular offsets from a segment S in the polyline L
\mathcal{M}	the maximum number of entries of an internal node in the R*-tree
m_{td}	the total number of subphrases of the query string t found in a document d
n	the number of records in the index, $n = D $
N_{bd}	encapsulates the document boost B_d and subphrase length normalization L_b
N_g	total number of search results returned by a GSA query

n_p	the number of subsequent page queries we performed for a query
n_{pl}	the number of subsequent page queries we performed for a Q2 or Q3 polygon query
N_{pl}	the total number of counties returned for Q2 and Q3 polygon results
n_{pt}	the number of subsequent page queries we performed for a Q2 or Q3 point query
N_{pt}	the total number of (ϕ, λ) pairs returned for Q2 and Q3 point results
N_r	the total number of search results returned in Q1 query
N_t	a normalizing factor used to make scores between queries comparable
O	the greatest perpendicular offsets from a segment S to points in the polyline L
p	(latitude, longitude) point
\mathcal{P}	the set of ranked text search results
P	a polygon or an ordered point set for a polygon
P'	the simplified polygon containing w_B or fewer points
P_{chosen}	the vector used to maintain the indices of chosen points of the simplified polygon
(ϕ, λ)	latitude and longitude pairs
p_i	a point on a polygon or polyline boundary
\mathcal{P}_i	an element in \mathcal{P}
P_L	an input polyline of the RDP algorithm
P'_L	the simplified polyline generated by the RDP algorithm
prk	primary keys
$P_{shifted}$	the shifted polygon of P generated by the algorithm <code>ShiftPolygon</code>
\mathcal{Q}	a query string to the suffix tree of string \mathcal{S}
Q	a query
Q1 = (t)	text only query
Q2 = (t, r)	text + radius query
Q3 = (p, r)	text + radius query
r	radius
r_1, r_2	two records
R_1	a list for <code>TexSpaPointResult</code> objects
R_2	a list for <code>TexSpaPolygonResult</code> objects
RDP	Ramer–Douglas–Peucker algorithm
R_i	the final rank score of the web page \mathcal{W}_i
R_{pl}	the total number of records returned for Q2 and Q3 polygon results
R_{pt}	the total number of records returned for Q2 and Q3 point results
R_{td}	the ranking score of query t for a document d
\mathcal{S}	the input string for constructing a suffix tree
S	single line segment
s_1, s_2 and s_3	objects of <code>subphraseResult</code>
SBST	Suffix Binary Search Tree

$ScoreMax$	the maximum text ranking score returned by the Q1 text search.
$sepF$	field separator
$sepN$	node separator
$sepP$	primary key separator
S_{index}	stores information of segments defining by the adjacent points in P_{chosen}
S_L	line segment from I_L to I_{chosen}
S_{offset}	stores information of segments defining by the adjacent points in P_{chosen}
S_R	line segment from I_{chosen} to I_R
t	query string
τ	the length of query string t
T_{bd}	$T_{bd} = \sqrt{\text{termFrequency}}$
T_c	the C++ side processing time
\mathcal{T}_e	the equally distributing case time for the PackPolygon algorithm
T_e	the total query time from the user clicking the serch button to the search results displayed on the web page
termFrequency	the number of times the SubPhrase text appears in a document
T_g	the total query time for a GSA query
T_p	the page query timing, which is the timing for subsequent requests sent by clicking a page number
T_s	the server side (including C++ processing time and Java processing time) processing time
\mathcal{T}_w	the worst case time for the PackPolygon algorithm
u	the length of the input string \mathcal{S} for a suffix tree
U	the set for point results for a r_star_tree query
v	the length of the query string \mathcal{Q} to the suffix tree of string \mathcal{S}
V	the set for polygon results for a r_star_tree query
w	the number of points representing a polyline L or a polygon P
w_B	the maximum number of points that can fit on one disk block
W_{mb}	the score of the first matched tag in the document's metadata in GSA Result Biasing
W_s	the weight for the spatial search
W_t	the weight for the text search
x_1, x_2	distances from the interest point p to the records r_1 and r_2 , respectively
y	the number of leaf nodes found in range in a Q3 query
Z	the number of point records in the list \mathcal{P} of text search results

Chapter 1

Introduction

Many human activities are more or less related to geographic information. For example, most documents stored on the web include references to geographical content, typically names of places. As a result, some applications wish to combine spatial location with text data when searching [35][38]. Traditional search engines treat place names in the search strings in the same way as any other keyword. This may be adequate in most circumstances, but there are situations, for example, when we want to find all the restaurants falling within 10 km of our working place, in which text only search engines could be improved. In this case, we are interested in all the documents that are associated with the region which is specified by the place name and a radius. In this thesis, we investigate efficient indexing method to support text + spatial query.

To support spatial search, in addition to conventional text search functions, we need to add features of spatial queries to the search engine, including [20]

1. Representation of spatial data in the index
2. Filtering by some spatial concept such as a bounding box or other shape

3. Sorting, scoring and boosting by distance from a query point or query region

In 2004, a general architecture for text + spatial search engine called the SPIRIT search engine architecture was presented [38]. SPIRIT consists of the following components: user interface; geographical and domain-specific ontologies; web document collection; core search engine; textual and spatial indexes of document collection; relevance ranking and metadata extraction as shown in Figure 1.1 [38].

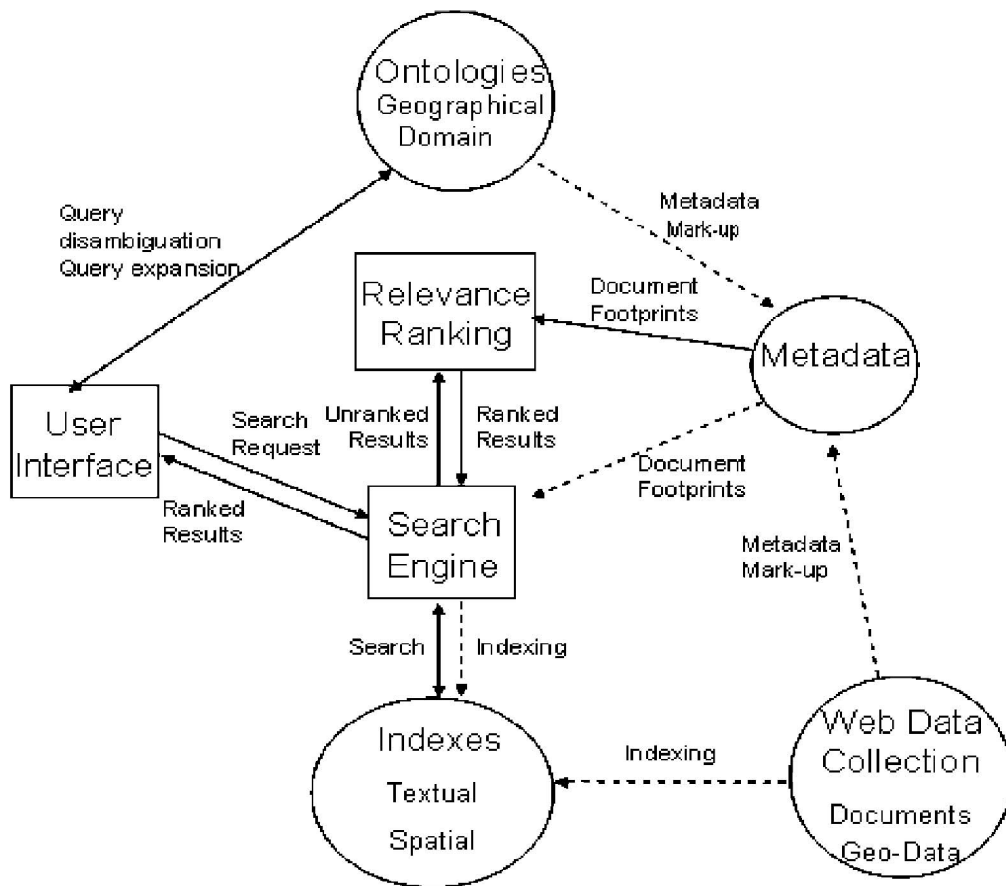


Figure 1.1: SPIRIT search engine architecture (from [38]).

In this thesis, we use UNB Connell Memorial Herbarium database as the basis for a web document collection with spatial information. We compare two approaches. The first uses suffix trees and R*-trees as data structures with a Lucene scoring algorithm for ranking

search results, the second approach applies spatial filtering to rank the results from a Google Search Appliance (GSA) search on a GSA index of the same data.

1.1 Suffix Trees

Suffix tree is a popular data structure for indexing text. Since 1960 tries were used in many computer science applications such as searching and sorting, dynamic hashing, conflict resolution algorithms, leader election algorithms, IP addresses lookup, coding, polynomial factorization, Lempel-Ziv compression schemes, and molecular biology [41]. Also, a number of suffix-based data structures have been proposed to facilitate on-line string searching [36]. Suffix tree is a special kind of trie, which can be used to index all suffixes in a text in order to carry out fast full text searches [23].

A suffix tree for a string \mathcal{S} is a tree whose edges are labeled with strings, such that each suffix of \mathcal{S} corresponds to exactly one path from the tree's root to a leaf. It is thus a Patricia tree for the suffixes of \mathcal{S} [22]. The suffix tree for the string \mathcal{S} of length u is defined as a tree such that: [32]

1. The paths from the root to the leaves have a one-to-one relationship with the suffixes of \mathcal{S} .
2. Edges spell non-empty strings.
3. All internal nodes (except perhaps the root) have at least two children.

Since such a tree does not exist for all strings, \mathcal{S} is padded with a terminal symbol not seen in the string (usually denoted \$). This ensures that no suffix is a prefix of another, and that there will be \mathcal{J} leaf nodes, one for each of the \mathcal{J} suffixes of \mathcal{S} .

An example suffix tree for the string “BANANA” is shown in Figure 1.2. Each substring is terminated with special character \$. The six paths from the root to a leaf correspond to the six suffixes A\$, NA\$, ANA\$, NANA\$, ANANA\$ and BANANA\$. The numbers in the boxes give the start position of the corresponding suffix [22]. Dashed edges link internal nodes.

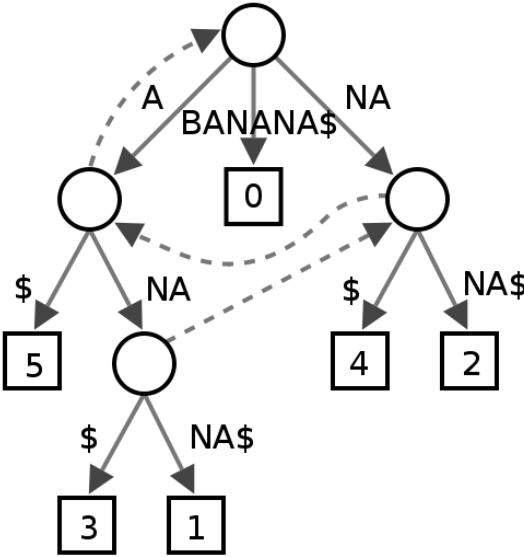


Figure 1.2: A suffix tree for the string “BANANA” from [22]

The classic application for suffix trees is the substring problem. One is first given a text \mathcal{S} of length u . After $O(u)$, or linear, preprocessing time, one must be prepared to take in a query \mathcal{Q} of length v and in $O(v)$ time either find an occurrence of \mathcal{Q} in \mathcal{S} or determine that \mathcal{Q} is not contained in \mathcal{S} . These bounds are achieved with the use of a suffix tree. The suffix tree for the text \mathcal{S} is built in $O(u)$ time during a preprocessing stage; thereafter, whenever a string of length $O(v)$ is input, the algorithm searches for it in $O(v)$ time using that suffix tree [32].

1.2 Background and Literature Review

The first linear time algorithm for constructing suffix trees was presented by Weiner [48] in 1973, although at that time a suffix tree was called a position tree. A few years later, a more space efficient algorithm to build suffix trees in linear time was given by McCreight [40]. More recently, a conceptually different linear-time algorithm was developed by Ukkonen [47], which has all the advantages of McCreight’s algorithm, but allows a much simpler explanation [32]. These classical algorithms [48, 40, 47] construct a suffix tree for a string of length n in $O(n \log |\Sigma|)$ time and $O(n)$ space, where Σ is the alphabet. Given a suffix tree for σ and a pattern α of length m , an algorithm to determine whether the pattern appears in the string can be implemented to run in $O(m \log |\Sigma|)$ time. A more recent algorithm due to Farach [30] removes the dependence on alphabet size [37].

The suffix array was introduced by Manber and Myers [39] in 1993 as an alternative to the classical suffix tree. The main advantage of suffix arrays over suffix trees is that, in practice, they use three to five times less space. The time bounds for construction and search in the case of a suffix array are $O(n \log n)$ and $O(m + \log n)$, with $O(n)$ space used [39]. Although both suffix trees and suffix arrays use linear space, the latter can be represented more compactly [37].

Irving and Love [37] defined the suffix binary search tree (SBST) in 2000 and its variant the suffix AVL tree in 2000. They show empirical evidence suggesting that, in practice, the suffix BST is broadly competitive with suffix trees and suffix arrays in indexing real data, such as plain text or DNA strings. A particular advantage is that a standard suffix BST can easily be constructed so as to represent a proper subset of the suffixes of a text. For example, if the text is natural language, it might be appropriate to present in the

tree only those suffixes that start on a word boundary, resulting in a saving in space and construction time by a factor of the order of $1 + w$, where w is the average word length in the text. For a suitably implemented SBST, a search requires $O(m + k)$ time, where k is the length of the search path in the tree. This gives $O(m + n)$ worst-case complexity, but typically in practice, all search paths will have $O(\log n)$ length, and searching will be $O(m + \log n)$ on average. This becomes a worst-case bound if AVL rotations are used to balance the tree on construction. The construction time for standard SBST can be as bad as $O(n^2)$ in the worst case, but for the refined version, it can be achieved in $O(nh)$ time, where h is the height of the tree. In the worst case, h can be $\Theta(n)$, but for random settings, h can be expected to be $O(\log n)$. In the case of the suffix AVL tree, construction takes $O(n \log n)$ time in the worst case [37].

In order to handle multi-dimensional point data efficiently, a number of structures have been proposed. Cell methods are not good for dynamic structures because the cell boundaries must be decided in advance [33]. A multidimensional binary search tree, or k-d tree, was presented by Bentley [27], which then became one of the prominent data structures for indexing spatial data. In the worst case, it requires $O(n^{1-1/d} + F)$ search time in a range search, where d is the number of dimensions and F is the number of points falling in the region. One drawback is that k-d trees and its variants do not take paging of secondary memory into account [33]. The R-tree is a height-balanced tree that is derived from the B-tree, and provides efficient indexing of multidimensional objects with spatial extent [45]. R-tree represents data objects by intervals in several dimensions, and is designed so that a spatial search requires visiting only a small number of nodes. An improved version of R-tree, the R*-tree was introduced by Beckmann et al [26]. The motivation of R*-tree is that there are several weaknesses of the original R-tree insertion algorithms. R*-tree aims at minimizing the overlap region between sibling nodes and

achieving lower storage utilization.

1.3 Objective

We assume that text documents have associated positions in space and we wish to search such documents with queries containing spatial components. For example, we might have a set of populated place names (e.g. cities), with associated locations (latitude, longitude) on the earth's surface. These place names can be part of larger documents or text based web pages. From here on, we use the word “document” to refer to item (e.g. web page, document, database, record) indexed by the search engine.

An example query might be to find all populated places within 50 km of a specific populated place, or of a given latitude, longitude. Let Q be a query, in this case, we have $Q = (\text{“Fredericton”, } 50\text{km})$ or $Q = ((45.95, -66.633333), 50\text{km})$. Other example queries might be to find restaurants within 10 km of your current position or of a known restaurant, then we have $Q = (\text{“Golden Triangle”, } 10\text{km})$ where “Golden Triangle” is the name of a restaurant. In any case, the search returns a ranked list of cities or restaurants nearby. If we represent search strings (e.g. city or restaurant names) by t , represent spatial information (e.g. latitude, longitude) by p and let r stand for radius, there are three query forms:

1. $Q1 = (t)$, search returns a ranked list of items matching the search string t , along with their associated spatial information (e.g. latitude, longitude).
2. $Q2 = (t, r)$, search returns a ranked list of documents with at least one spatial component having its location falling within the circle of radius r centred at the position p of the ranked documents.

3. $Q3 = (p, r)$, search returns a ranked list of documents with at least one spatial component having its location falling within the circle of radius r centred at position p .

Assume now that there are records consisting of plain text and associated spatial information in a database as shown in Figure 1.3.

Record 1	Record 2	Record 3
McDonald's Restaurants of Canada	McConnell Hall, University of New Brunswick	Head Hall, University of New Brunswick
①(45.934516N, 66.663308W) A ②(45.961817N, 66.643622W) F	(45.946419N, 66.639297W)	(45.949961N, 66.641711W) E
Record 4	Record 5	Record 6
Shoppers Drug Mart	Staple Business Depot	Victory Meat Market Ltd
①(45.942639N, 66.655147W) C ②(45.961817N, 66.643622W) F ③(45.976414N, 66.649003W) I	(45.939778N, 66.662633W)	(45.962739N, 66.645572W) H

Figure 1.3: An example database with records consisting of plain text and associated spatial information

Figure 1.4 shows corresponding positions on a Google map of the records in Figure 1.3.

Figure 1.5 shows examples of the three queries and query answers using the database shown in Figure 1.3.

Ranking of search results becomes important for large amounts of data in the search result. In cases 3 and 4, we have to realize ranked nearest neighbour search and ranked range search.

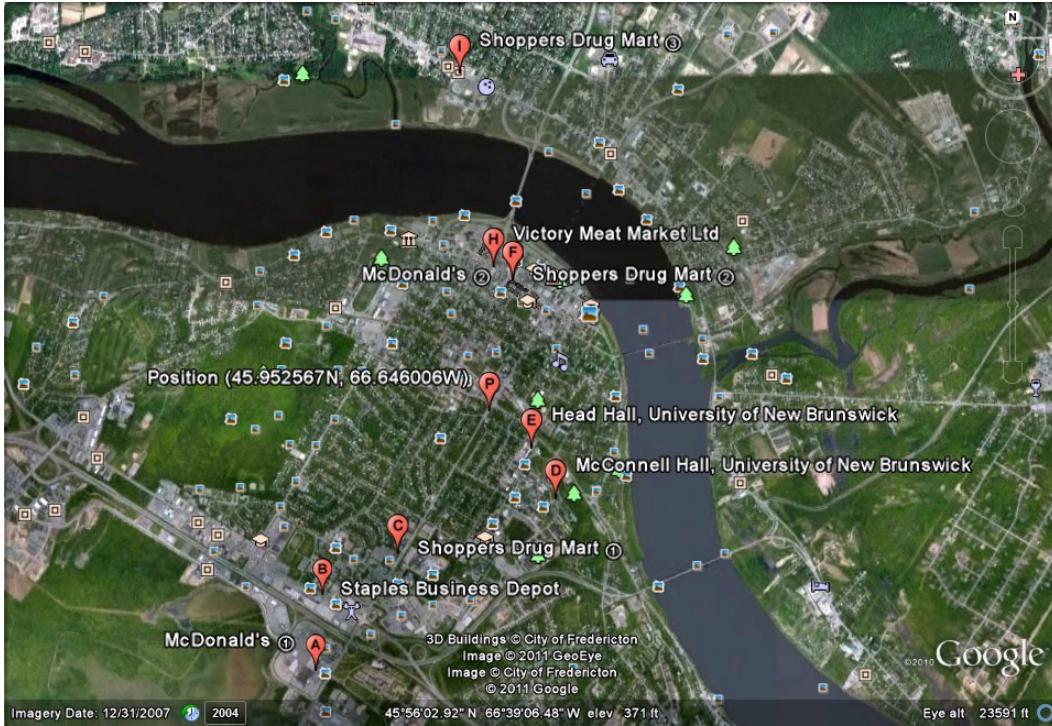


Figure 1.4: Positions on Google Map of records in Figure 1.3

Suffix trees and its variants can be used to efficiently index and search text, but what if we have spatial constraints on the query? Since it is not easy to use suffix trees to represent the spatial data, we need another data structure to index spatial data. The R-tree and its variants the R^* -tree are among the most popular indexing methods supporting range search and nearest neighbour search [44]. The R-tree is a dynamic index structure that provides a way to handle multi-dimensional spatial data efficiently. Other than traditional data structures, R-tree represents data objects by intervals in several dimensions. Thus, we can index the herbarium database using two different data structures. How can we implement and combine these two data structures for efficient text search with spatial data constraints? Can the Lucene text similarity engine be used to effectively rank the text search results [34]? In this thesis, we explore novel data structures and ranking algorithms for efficient combined text and spatial search. Our research objectives focus on achieving efficient worst case and average case search cost using linear space.

Query 1: Q = ("Mc")
<p>Search returns:</p> <ul style="list-style-type: none"> (1). McDonald's Restaurants of Canada, (45.934516N, 66.663308W). (2). McDonald's Restaurants of Canada, (45.961817N, 66.643622W). (3). McConnell Hall, University of New Brunswick, (45.946419N, 66.639297W).
Query 2: Q = ("McDonald", 1.5km)
<p>Search returns:</p> <ul style="list-style-type: none"> (1). McDonald's Restaurants of Canada, (45.934516N, 66.663308W). (2). McDonald's Restaurants of Canada, (45.961817N, 66.643622W). (3). Shoppers Drug Mart, (45.961817N, 66.643622W), 0km from (2). (4). Victory Meat Market Ltd, (45.962739N, 66.645572W), 0.185km from (2). (5). Staple Business Depot, (45.939778N, 66.662633W), 0.592km from (1). (6). Shoppers Drug Mart, (45.942639N, 66.655147W), 1.098km from (1).
Query 3: Q = (position P = (45.952567N, 66.646001W), 1.5km)
<p>Search returns:</p> <ul style="list-style-type: none"> (1). Head Hall, University of New Brunswick, (45.949961N, 66.641711W), 0.441km from P. (2). McConnell Hall, University of New Brunswick, (45.946419N, 66.639297W), 0.862km from P. (3). McDonald's Restaurants of Canada, (45.961817N, 66.643622W), 1.043 km from P. (4). Shoppers Drug Mart, (45.961817N, 66.643622W), 1.043 km from P. (5). Victory Meat Market Ltd, (45.962739N, 66.645572W), 1.122km from P. (6). Shoppers Drug Mart, (45.942639N, 66.655147W), 1.315km from P.

Figure 1.5: Examples of the three query types using database as shown in Figure 1.3.

Chapter 2

A System Supporting Text Search with Queries Having Spatial Constraints

2.1 Methodology

We plan to use the suffix binary search tree and R*-tree because they are efficient on indexing text and spatial data, respectively. To achieve efficient query in the three forms described in Chapter 1, a system is desired as shown in Figure 2.1.

The targeted database will be indexed using two different data structures; the R*-tree for spatial data and suffix binary search tree for text data. The text data will be associated with the corresponding spatial information. This will enable us to do two basic kinds of queries; text query based on SBST and spatial only query based on the R*-tree. The central question of this thesis is how can we perform a text + spatial query efficiently? As we can see from Figure 2.1, to perform a Q_2 (text + radius) query, we first perform pure

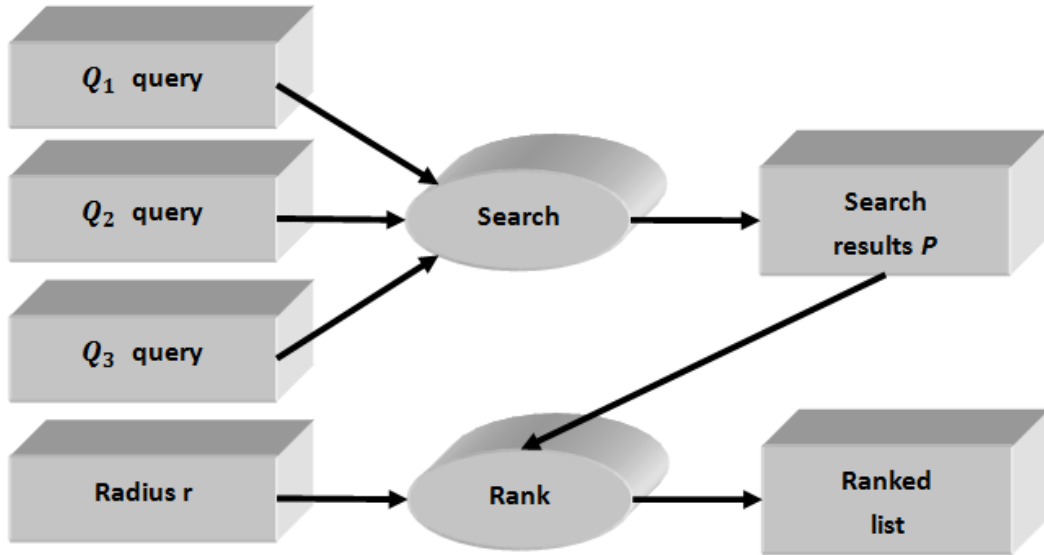


Figure 2.1: A possible system architecture for text+spatial query.

text search, which will return a set \mathcal{P} of search results. We then introduce a radius into the system, for each point in \mathcal{P} , we perform a point + radius search, or nearest neighbour search, which will return all points falling in range.

Since the number of points in range can vary, an important question is how can we rank the results as to their importance. Our way is to rank the results by their positions. This idea is based on the common sense that the nearer a point is to a specific point, the more likely that people will be interested in it. An example is illustrated in Figure 2.2. In this example, two records r_1 and r_2 fall inside a circle of radius r at point p . The distance from p to r_1 and r_2 are x_1 and x_2 , respectively, with $x_1 < x_2$. Thus, document r_1 should be ranked higher than document r_2 .

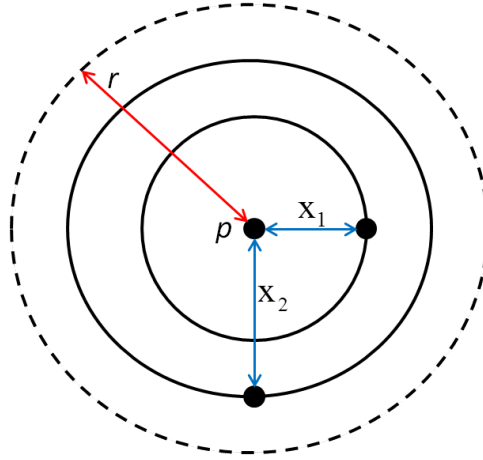


Figure 2.2: An example for ranking method.

2.2 Test Methodology

The system will be tested on a database. The performance of the system will be compared with the Google Search Appliance (GSA) on the targeted data set. The testing system is depicted in Figure 2.3. In our testing, we can have Q1 (text only), Q2 (text + radius) or Q3 (point + radius) search. Since the GSA does not provide a way to perform text plus spatial data search, we have to do the text search first and then introduce spatial constraints in a different way. Steps 2 and 3 in Figure 2.3 are data preprocessing procedures for the GSA search engine. In step 4, we perform text search using GSA search engine. If the given query contains latitude and longitude information, this spatial information should be converted to text information associated with it first, such as a nearby city or place name. In step 5, a ranked set of matching documents will be generated and the ranking will be decided merely on text data during this procedure. In step 6, we introduce the spatial constraint (radius) through a nearest neighbour filter. After this, the ranked set of matching documents with spatial components will be generated.

In the UNB text + spatial search engine, the targeted database will be preprocessed as

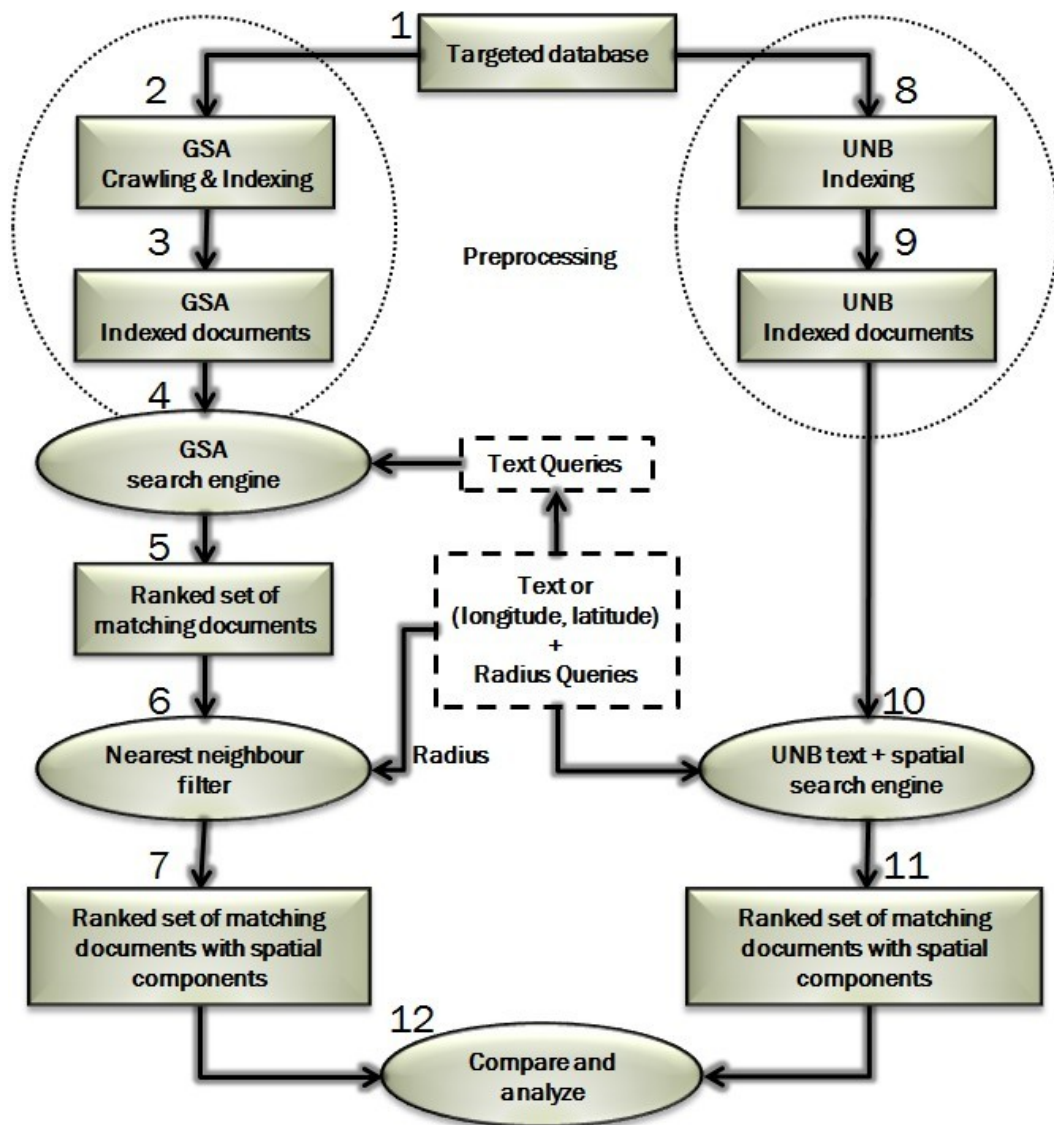


Figure 2.3: Testing system.

shown in steps 8 and 9. The system then directly performs the given query $Q = (\text{text or (latitude, longitude), radius})$ in step 10, and generates a ranked set of matching documents with spatial constraints in step 11. For text search Q1 ranking, we plan to use a text retrieval ranking algorithm such as Lucene scoring [34].

Finally, we will compare and contrast these two search engines on various aspects such as searching time and suitability of ranked results in step 12.

Chapter 3

Data Preprocessing

UNB has a Connell Memorial Herbarium database, each record of which contains text and spatial information (latitude and longitude). In the thesis, the system will be tested on the herbarium database. The performance of the system will be compared with the Google Search Appliance (GSA) on the UNB Herbarium data set. To index the UNB Connell Memorial Herbarium database using a Google Search Appliance (GSA), we have to perform the data preprocessing first. In the data preprocessing stage, we transformed each database record to a webpage with appropriate metadata and content, put all the webpages generated on the web server running on the UNB FCS network, and indexed these webpages using the GSA.

The preprocessing steps are illustrated as shown in Figure 3.1.

3.1 UNB Connell Memorial Herbarium database

The Connell Memorial herbarium is the largest collection of vascular plant specimens from the New Brunswick flora. There are approximately 55,000 vascular plant specimens from

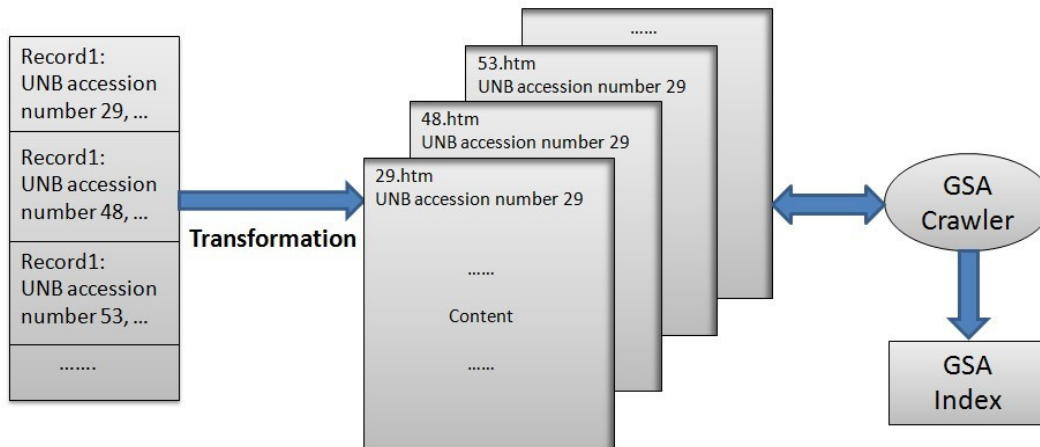


Figure 3.1: The preprocessing process for GSA index.

New Brunswick, 9,000 non-New Brunswick vascular plants, and about 1,000 algae, mainly seaweeds . About one third of the vascular plant specimens from New Brunswick are currently in the database [21]. An example record of the online database is shown in Figure 3.2.

The electronic version of the UNB Connell Memorial Herbarium database was provided to us by Michael Casey (UNB Biology department) as a .csv file on Sep. 13, 2011. The database contains 40,791 records in total. An example database record is as follows:

```
"29","Botrychium multifidum (Gmel.) Rupr.", "", "", "", "", "15", "8", "1844",
"", "Hill pastures", "College Hill, Fredericton", "York", "NB", "",
"Leathery grapefern", "Ophioglossaceae", "Botryche à feuille couchée", "",
"", "", "College Hill, Fredericton", "", ""
```

The meaning of the 22 columns in each record are: UNB accession number, Full name, Latitude, Longitude, Collector(s), Collector's number, Day, Month, Year, Abundance, Habitat, Location, County, Prov/State, Notes, Common Name, Family, French name, Phenology, Rare latitude, Rare longitude, Rare location, CDC status and Synonyms. The above example contains missing values, which are Latitude, Longitude, Collector(s),

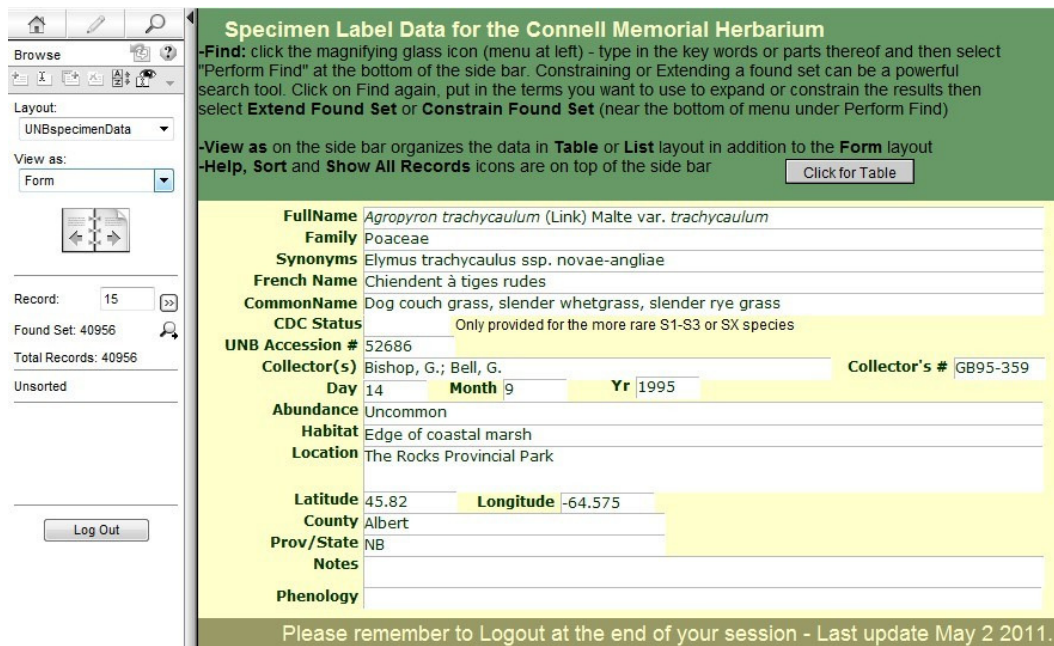


Figure 3.2: An example database record from the UNB Connell Memorial Herbarium database FileMaker version as viewed at <http://herbarium.biology.unb.ca/fmi/iwp/cgi> [21].

Collector's number, Abundance, Phenology, Rare latitude, Rare longitude, CDC status and Synonyms.

Attributes are separated by commas and each attribute is surrounded by quotation marks. The records can be uniquely recognized by their UNB accession numbers.

3.2 Transforming the collection

This section describes how we transformed the UNB Connell Memorial Herbarium database records to a collection of web pages in the form required by the GSA crawler. Generally, there are two parts in the transformation process. First, we set up a web server running on the FCS network. Secondly, we wrote a program to realize database record transformation

to an HTML format suitable for GSA crawling.

3.2.1 Work station on FCS network

Web servers are used to serve web pages requested by client computers. Clients typically request and view web pages using web browsers such as Firefox, Opera, or Mozilla. Apache is the most commonly used Web Server on Linux systems. The goal of Apache is to provide a secure, efficient and extensible server that provides HTTP services in sync with current HTTP standards. The Apache HTTP server is called “httpd”, where the “d” stands for daemon. Apache’s http server is a project of The Apache Software Foundation [2].

Under CentOS Linux, we first installed the httpd package using `yum install httpd`. Then the httpd RPM package was invoked and the `/etc/init.d/httpd` script was installed. We can set the environmental variables in the configuration file for the httpd service, which can be accessed using the `vim /etc/sysconfig/httpd` command, and in the init script, which can be accessed by the command `vim /etc/init.d/httpd`. In our case, we only checked that the port number was correctly set to port 80, and that the paths were set correctly. They were all correct on installation so no actual changes were made. After installing and initializing, we can get access to the `/etc/init.d/httpd` script by using the `/sbin/service` command. To start the httpd server, we used `/sbin/service httpd start`. Command `/sbin/service httpd stop` is used to stop the web server and `/sbin/service httpd restart` is used to restart the web server. To reload the server configuration file, the command `/sbin/service httpd reload` is run when logged in as root.

The URL of the installed web server is decided by the machine’s IP address. For example,

if the IP address is 131.202.243.11, then the web server URL address is:

`http://131.202.243.11/`

The content you want to show on the web server is placed under subdirectory `/var/www/html/` on the machine running the web server `httpd`. People can get access to these contents through the URL directly by typing, e.g. `http://131.202.243.11`. In our case, we created a subdirectory `/test` under `/var/www/html` as shown in Figure 3.3:

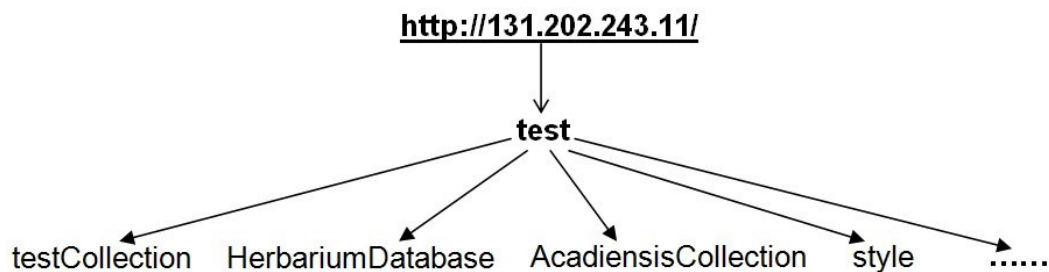


Figure 3.3: The tree structure under the directory `http://131.202.243.11`.

3.2.2 Transform database records

We used a Java program to read database records, recognize attributes, and write each record to a separate `.htm` file with the correct format. The generated web pages are uniquely named by their UNB accession numbers.

For the field “French name”, the original records are encoded using IBM PC Extended ASCII, or MS-DOS code page 437 character set, which is the character set of the original IBM PC [5]. To determine that the code page used in the Connell Memorial Herbarium database was this version of IBM PC Extended ASCII, we viewed the hexadecimal code used to represent French characters such as à, é, ô and ç using the Emacs editor.

Note that the .csv file French characters are different from the UTF-8 version stored in the FileMaker database. We know this as we can view the database contents in a web browser (see e.g. Figure 3.2), and find the hexadecimal value of the displayed character.

Figure 3.4 is the IBM PC Extended ASCII table. In web browsers, the most widely used Extended ASCII standard is ISO-8859-1. The Extended ASCII table for ISO-8859-1 is shown in Figure 3.5 [13].

8-	Ç	ü	é	â	ä	à	å	ç	ê	ë	è	ï	î	ì	Ä	Å
	00C7	00FC	00E9	00E2	00E4	00E0	00E5	00E7	00EA	00EB	00E8	00EF	00EE	00EC	00C4	00C5
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9-	É	æ	Æ	ô	ö	ò	û	ù	ÿ	Ö	Ü	ç	£	¥	¤	ƒ
	00C9	00E6	00C6	00F4	00F6	00F2	00FB	00F9	00FF	00D6	00DC	00A2	00A3	00A5	20A7	0192
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A-	á	í	ó	ú	ñ	Ñ	ª	º	¿	¬	½	¾	¡	«	»	
	00E1	00ED	00F3	00FA	00F1	00D1	00AA	00BA	00BF	2310	00AC	00BD	00BC	00A1	00AB	00BB
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B-	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
	2591	2592	2593	2502	2524	2561	2562	2556	2555	2563	2551	2557	255D	255C	255B	2510
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C-	ℒ	ℓ	ℓ	ℓ	—	†	‡	‡	‡	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ
	2514	2534	252C	251C	2500	253C	255E	255F	255A	2554	2569	2566	2560	2550	256C	2567
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D-	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ	ℒ
	2568	2564	2565	2559	2558	2552	2553	256B	256A	2518	250C	2588	2584	258C	2590	2580
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E-	α	β	Γ	π	Σ	σ	μ	τ	Φ	Θ	Ω	δ	∞	φ	ε	∩
	03B1	00DF	0393	03C0	03A3	03C3	00B5	03C4	03A6	0398	03A9	03B4	221E	03C6	03B5	2229
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F-	≡	±	≥	≤	∫	∫	÷	≈	°	•	•	√	∞	∞	☐	NBSP
	2261	00B1	2265	2264	2320	2321	00F7	2248	00B0	2219	00B7	221A	207F	00B2	25A0	00A0
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F

Figure 3.4: IBM PC Extended ASCII Set code page 437, character 80₁₆ (128) through FF₁₆ (255) [5].

In order to display French names in the correct form, we first built a one-to-one transformation table for the two Extended ASCII standards. For each record, we encode the string in field “French Name” using the getBytes() function of String class in Java, get a sequence of bytes using the IBM PC charset, store the result into a new byte array, then perform lookup in the transformation table and retrieve the corresponding ISO-8859-1

8-																
9-																
A-	NBSP	ı	ç	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯
	00A0	00A1	00A2	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AD	00AE	00AF
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B-	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
	00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C-	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
	00C0	00C1	00C2	00C3	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB	00CC	00CD	00CE	00CF
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D-	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
	00D0	00D1	00D2	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00DA	00DB	00DC	00DD	00DE	00DF
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E-	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
	00E0	00E1	00E2	00E3	00E4	00E5	00E6	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE	00EF
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F-	ø	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ
	00F0	00F1	00F2	00F3	00F4	00F5	00F6	00F7	00F8	00F9	00FA	00FB	00FC	00FD	00FE	00FF
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F

Figure 3.5: Extended ASCII table for ISO-8859-1 standard, character 80_{16} (128) through FF_{16} (255) [13].

codes. Finally, we construct a new string using the constructor `String(byte[] bytes, String charsetName)` of class `String` by decoding the specified array of bytes using the ISO-8859-1 character set and generate the French name which can be recognized by HTML. The architecture for the entire java program is shown in Figure 3.6.

The generated web page 52686.htm with UNB accession number 52686 is shown in Figure 3.7. The time to transform all 40,791 records by the transform.java program was 11.8 seconds.

3.2.3 Adding Spatial Information to the Records

For records in Herbarium database that have latitude and longitude pairs (ϕ, λ) associated, we can directly use (ϕ, λ) pairs in spatial indexing. For those ones that do not have (ϕ, λ)

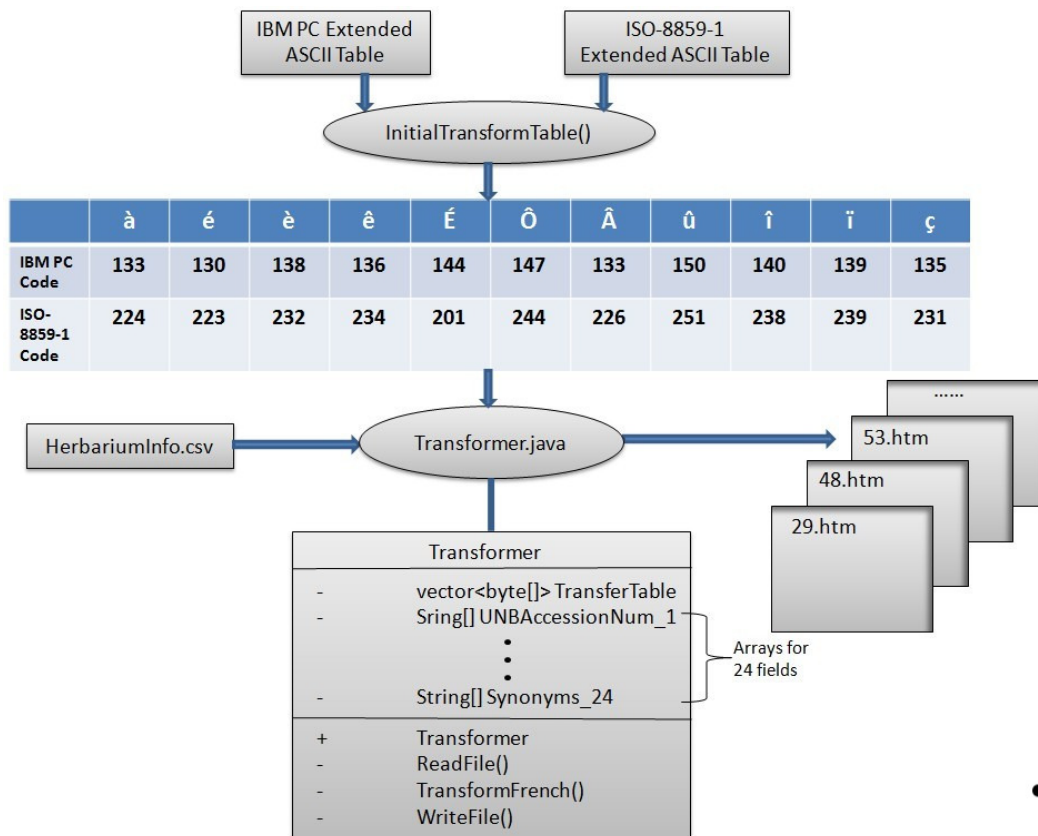


Figure 3.6: Architecture of the Transformer.java program which is used to transform database records to required web pages.

pairs but have locations related, we can acquire the corresponding (ϕ, λ) pairs using a Google Map API. If neither of the (ϕ, λ) pair and location exist, we can add polygon information according to their county or province as spatial information.

3.2.3.1 Adding locations to the records

For the records that have specific locations related to them, such as “Mary-land Road, College Hill, Fredericton, NB”, the Google Map API for web services provides us a way to convert those locations into geographic coordinates, such as a latitude and longitude pair (ϕ, λ) . The process of this conversion is called Geocoding. The Google Geocoding API provides a direct way to access a geocoder via an HTTP request [8]. A Geocoding API

Specimen Label Data for the Connell Memorial Herbarium

UNB Accession Number: 52686	
Basic Information	FullName Agropyron trachycaulum (Link) Malte var. trachycaulum
	Family Poaceae
	Synonyms Elymus trachycaulus ssp. novae-angliae
	French Name Chientent à tiges rudes
	CommonName Dog couch grass, slender whetgrass, slender rye grass
	CDC Status
	Collector(s) Bishop, G.; Bell, G.
	Collector(s) Number GB95-359
	Day 14
	Month 9
	Year 1995
	Abundance Uncommon
	Habitat Edge of coastal marsh
	Notes
Spatial Information	Phenology
	Location The Rocks Provincial Park
	Latitude 45.82
	Longitude -64.575
	County Albert
	Prov/State NB

Figure 3.7: An example of a generated web page 52686.htm.

request is in the following form:

`http://maps.google.com/maps/geo?address&output&key`

where **address** is a string containing the text stored in the **Location**, **County** and **Prov/State** fields (comma separated) of a record, **output** can be one of three formats: **JSON** (JavaScript Object Notation), **XML**, or **CSV** (comma separated values), and **key** is a text string used when authenticating for paying users (i.e. users with >2,500 geolocation requests per day). All parameters are separated by the ampersand (&) character. After sending an HTTP request, the Geocoding response is returned in the format indicated by **output**; we used **CSV** for this research. Then we can analyze the output and get the corresponding latitude and longitude pair (ϕ, λ). In some cases, the Google Geocoding API returns “error”. In “error” cases, we describe the location using a polygon corresponding to the **County** or **Prov/State** field.

3.2.3.2 Adding polygons to the records

After the above processing, some records do not have an address that provides a (ϕ, λ) position. For these records, we add polygons based on their `County` or `Prov/State` fields. The geographic data describing county and N.B. province boundaries was downloaded from SNB Geographic Data & Maps [19]. The downloaded data is in ESRI Shapefile (.shp) format, which we transformed to common text file (.txt) format using the Shapefile API from a Shapefile C library [18]. The output polygon descriptions include a pair of points p_1 and p_2 that define the bounding box of all points defining the polygon.

The transformed data is in the N.B. Stereographic Double projection with the following parameters:

```
Reference System: North American Datum 1983 (CSRS)
False Northing: 7,500,000
False Easting: 2,500,000
Latitude of Origin: 46.5
Central Meridian: -66.5
Scalefactor: 0.999912
Units: Meters
```

In our indexing scheme, the spatial data are represented by latitude and longitude, so we need to convert the Stereographic Double (x, y) (East, North) coordinates into the corresponding geographic coordinates. The projection can be done using the ArcGIS engine, which is a geographic information system (GIS) for working with maps and geographic information [4]. Figure 3.8 shows the user interface for projection in ArcCatalog as we invoked it.

We invoked this process on a total of 15 county boundaries and one province boundary.

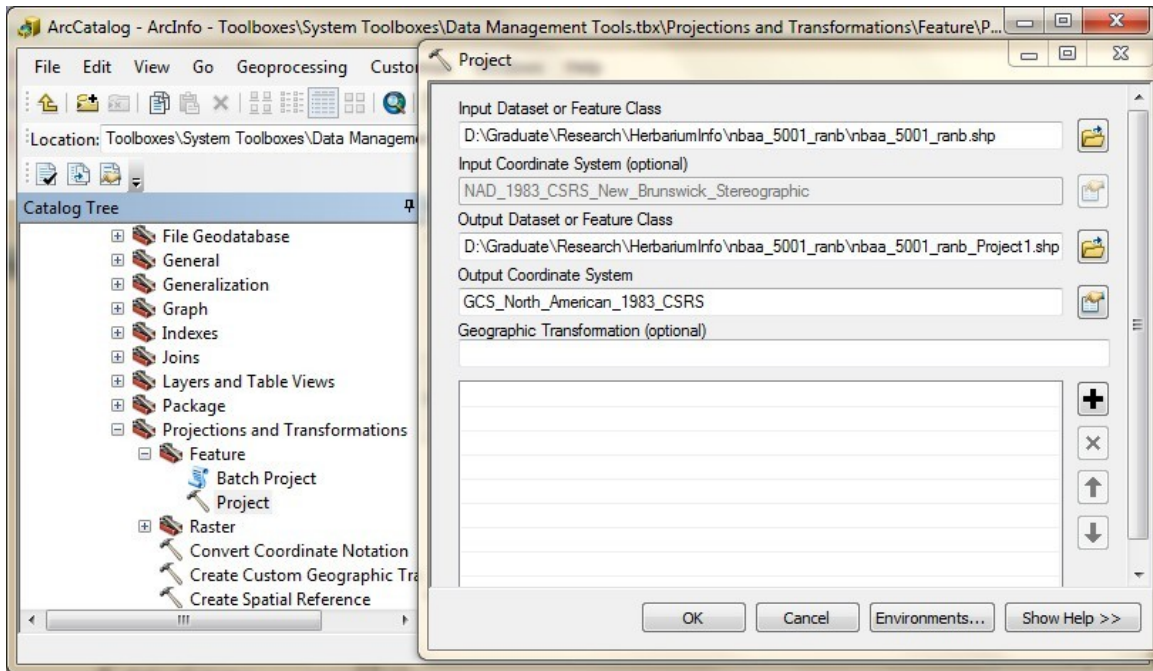


Figure 3.8: The user interface for projection in ArcCatalog as used for transforming N.B. Stereographic Double projection coordinates to (ϕ, λ) pairs.

The input is the original Shapefile using the N.B. Stereographic Double projection coordinate system. The output file is a Shapefile containing (ϕ, λ) geographic coordinate pairs as spatial components. Note that the NAD 1983 CSRS reference frame was used as output as most of the records were collected prior to the existence of GPS and the 1984 Geodetic Reference System (GRS84) [24].

3.2.3.3 Polygon Simplification

In the output Shapefile, each polygon can be represented by thousands of (ϕ, λ) pairs, which is too large to directly use in our indexing. It is helpful to simplify the polygons before we start to index the data. There is a famous polyline simplification algorithm independently developed in 1972 by Urs Ramer [43] and by David Douglas and Thomas Peucker in 1973 [29] for reducing the number of points in a curve that is approximated by

a series of points. Assume that the input is a polyline L represented by an ordered set of w points and a tolerance ϵ . The algorithm proceeds as follows:

1. Connect the start point and the end point of L using a line segment S .
2. Compute the perpendicular offsets of all points in L from S , and mark the point M with the greatest offset O .
3. If $O < \epsilon$, then S is considered adequate to represent all the points between the start point and the end point in a simplified form. Otherwise, L is divided into two polylines L_l and L_r at point O .
4. The algorithm then recursively repeats this process for the two parts L_l and L_r , from the start point to O , and from O to the end point.

Figure 3.9 shows an example of how the Ramer–Douglas–Peucker (RDP) algorithm works for a piecewise linear curve.

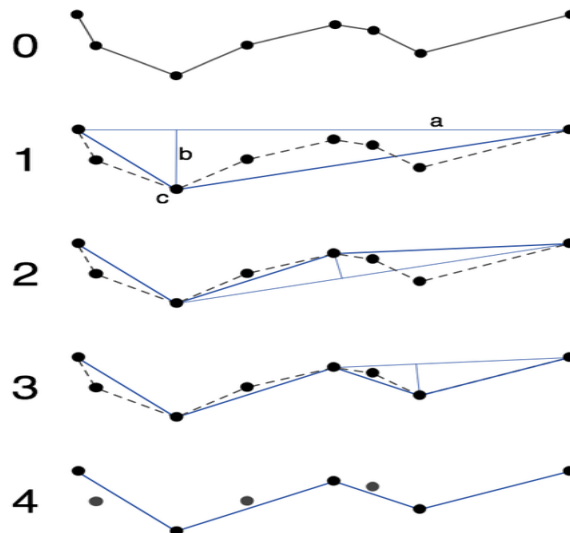


Figure 3.9: An example of how the Ramer–Douglas–Peucker algorithm works for a piecewise linear curve [7].

We can use a similar way to RDP algorithm to reduce the number of (ϕ, λ) pairs describing a polygon, which in turn reduces the space required for representing a polygon in our index. Our requirements for a polygon simplification algorithm are different from the RDP algorithm in the following aspects:

1. The RDP algorithm is designed for polylines and paths, so it requires its input ordered point set to have a distinct start point and end point, while the start and end points for a polygon are the same point geometrically.
2. For the RDP algorithm, ϵ is an input parameter, and a larger ϵ value increases the number of points removed from the original polyline. For I/O-efficient polygon simplification, the overriding concern is that each simplified polygon fits on one disk block. As original polygons have widely varying numbers of points defining them, ϵ cannot be used as an input parameter.
3. For I/O-efficiency, we require that approximately the same number of points are used to represent each simplified polygon. This permits each simplified polygon to fit on one disk block. We assume at most w_B points can fit on one disk block. So instead of ϵ , the input parameter w_B is used to control the number of points describing a simplified polygon.

To use the RDP algorithm for I/O-efficient polygon simplification, we have modified the RDP algorithm. The polygon simplification algorithm `PackPolygon` shown in Algorithm 3.1 is a significant modification of the RDP algorithm. Note that we compute the offset properly using the great-circle distance [11]. For all the polygons P that have more than w_B points, `PackPolygon` returns a simplified polygon, containing exactly w_B points; if P has less than w_B points, the simplified polygon P' is exactly the input polygon P .

Algorithm 3.1: PackPolygon (point[] P , int w_B)

Input:

point[] P : the input polygon describing by an ordered point set;
int w_B : the maximum number of points fitting on one disk block;

Output:

point[] P' : the simplified polygon containing w_B or fewer points ;

```
1 if  $|P| \leq w_B$  then
2    $P' \leftarrow P$  ; return  $P'$  ;
3 Vector  $P_{chosen}$ ,  $S_{offset}$ ,  $S_{index}$ ; initially empty ;
4 Append 0 to  $P_{chosen}$ ; Append  $|P| - 1$  to  $P_{chosen}$  ;
5 int  $I_{max} \leftarrow -1$ ; double  $D_{max} \leftarrow 0$  ;
6 forall the points in  $P$  between 0 and  $|P| - 1$  do
7    $I_{max} \leftarrow$  index of the point having the biggest great circle distance to  $P[0]$  ;
8    $D_{max} \leftarrow$  distance (  $P[0], P[I_{max}]$  ) ;
9 Append  $I_{max}$  to  $S_{index}$ ; append  $D_{max}$  to  $S_{offset}$  ;
10 int  $I_{chosen}$ ,  $I_{insert}$ ,  $I_s$ ,  $I_e$  ;
11 while  $|P_{chosen}| < w_B$  do
12    $I_{chosen} \leftarrow -1$ ,  $I_{insert} \leftarrow -1$  ;
13   int  $index \leftarrow$  the index of the maximum element in  $S_{offset}$  ;
14    $I_{chosen} \leftarrow S_{index}[index]$  ;
15   for int  $i \leftarrow 0$  to  $|P_{chosen}| - 1$  do
16     if  $P_{chosen}[i] < I_{chosen}$  &  $I_{chosen} < P_{chosen}[i + 1]$  then
17       Insert  $I_{chosen}$  to  $P_{chosen}$  at position  $i + 1$ ;  $I_{insert} \leftarrow i$ ; break ;
18    $S_{offset}.removeElementAt(I_{insert})$ ;  $S_{index}.removeElementAt(I_{insert})$  ;
19   for int  $i \leftarrow I_{insert}$  to  $I_{insert} + 1$  do
20     // 1st time  $\Rightarrow$  left side; 2nd time  $\Rightarrow$  right side
21      $I_s \leftarrow P_{chosen}[i]$ ,  $I_e \leftarrow P_{chosen}[i + 1]$  ;
22     if  $I_e == I_s + 1$  then
23       Insert 0.0 to  $S_{offset}$  at position  $i$  ;
24       Insert  $-1$  to  $S_{index}$  at position  $i$  ;
25     else
26       forall the points in  $P$  between  $I_s$  and  $I_e$  do
27          $I_{max} \leftarrow$  the index of the point having the biggest great circle distance
28          $D_{max}$  to the circle from  $P[I_s]$  to  $P[I_e]$ ;
29         Insert  $D_{max}$  to  $S_{offset}$  at position  $i$  ;
30         Insert  $I_{max}$  to  $S_{index}$  at position  $i$  ;
31 for int  $i \leftarrow 0$  to  $w_B - 1$  do
32    $P'[i] \leftarrow P[P_{chosen}[i]]$  ;
33 return  $P'$  ;
```

We use the vector P_{chosen} to maintain the indices of chosen points of the simplified polygon. The vectors S_{offset} and S_{index} store information of segments defining by the adjacent points in P_{chosen} . As we can see from Algorithm 3.1, the PackPolygon has the following steps:

1. Initially, we add the indices of the first and last points, which are 0 and $|P| - 1$, to the vector P_{chosen} , as we can see at line 4. For input polygon P , we assume these two points are the same point geometrically. At lines 6 to 7, we compute the distances from all the other points to $P[0]$, recording the index I_{max} of the point with the biggest distance D_{max} to $P[0]$. At line 8, we add D_{max} to the vector S_{offset} , and add I_{max} to the vector S_{index} .
2. Within all the elements in S_{offset} , we find the biggest one, and record its index as $index$ at line 12. Then we get the value of the element at position $index$ in vector S_{index} , which is recorded as I_{chosen} at line 13. I_{chosen} is the index in P of the point we choose to add to P_{chosen} in this loop. In lines 14 to 16, we insert I_{chosen} to P_{chosen} at the right position, so that P_{chosen} can be kept as an ordered point set. In P_{chosen} , the index of the element right before I_{chosen} is recorded as I_{insert} , so I_{chosen} is inserted between elements $P_{chosen}[I_{insert}]$ and $P_{chosen}[I_{insert} + 2]$. For easy description, we mark $P_{chosen}[I_{insert}]$ and $P_{chosen}[I_{insert} + 2]$ as I_L and I_R .
3. At line 17, we remove the element at position I_{insert} in vectors S_{offset} and S_{index} , since the segment between I_L and I_R is replaced by two new segments S_L and S_R , where S_L is from I_L to I_{chosen} , and S_R is from I_{chosen} to I_R .
4. In the original point set P , for the points between I_L and I_{chosen} (left side), we compute the perpendicular offsets from them to the segment S_L , and record the index of the point having the biggest offset D_{max} as I_{max} . Since segment S_L is the $I_{insert}th$ segment in P_{chosen} , so we insert D_{max} to S_{offset} at position I_{insert} , and add

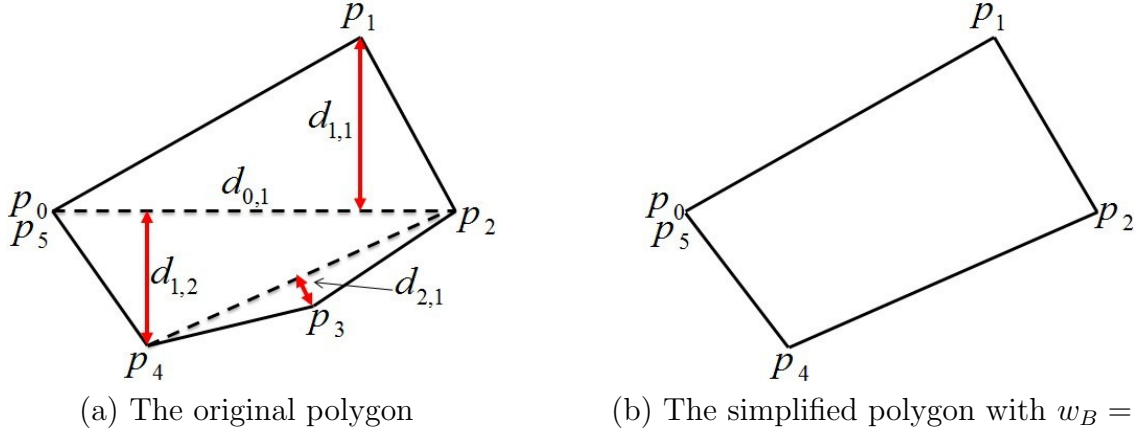


Figure 3.10: An example of the PackPolygon algorithm with the input polygon P containing 6 points and $w_B = 5$.

I_{max} to S_{index} at position I_{insert} . For those points between I_{chosen} and I_R (right side), we compute their perpendicular offsets to the segment S_R , recording the index I_{max} of the point with the biggest offset D_{max} . Then we add D_{max} and I_{max} to S_{offset} and S_{index} at position $I_{insert} + 1$, respectively. This step is completed by the pseudocode between lines 18 and 27, using a for loop from I_{insert} to $I_{insert} + 1$, i.e. execute twice.

5. If the number of points in P_{chosen} is more than w_B , we go back to step 2 (line 11) and repeat, until we get w_B points in P_{chosen} .

Figure 3.10 (a) shows a simple example illustrating how PackPolygon works. In this example, the original polygon has 6 points, so we have $P = \{p_0, p_1, p_2, p_3, p_4, p_5\}$. Assuming $w_B = 5$, the PackPolygon algorithm proceeds as follows:

1. Initially, we add the indices of p_0 and p_5 , which are 0 and 5 to the P_{chosen} vector. Within all the other points, p_2 has the maximum distance $d_{0,1}$ to p_0 , so we append $d_{0,1}$ to S_{offset} , and append the index of p_2 , which is 2 to S_{index} . After initialization, we have $P_{chosen} = \{0, 5\}$, $S_{offset} = \{d_{0,1}\}$, $S_{index} = \{2\}$ and $|P_{chosen}| = 2$. Here we use the notation $d_{i,j}$ to indicate the maximum distance computed at i times through

the while loop at line 10. Subscript j indicates the count (1 or 2) of iterations in the for loop at line 18. Initially, $d_{0,1}$ is computed as D_{max} outside the while loop at line 7.

2. The algorithm enters the while loop. The maximum element in S_{offset} now is $d_{0,1}$, stored as $S_{offset}[0]$. We next get the element at position 0 in S_{index} , which is 2. In the for loop at lines 14 to 16, the index 2 is next inserted into the vector P_{chosen} between 0 and 5. So now we have $P_{chosen} = \{0, 2, 5\}$, and $I_{insert} = 0$. At line 17, we remove the element at position 0 in S_{index} and S_{offset} , leaving both as \emptyset . For point p_1 (which is $P[1]$ in the PackPolygon algorithm), we compute (for loop at lines 24 and 25) its distance D_{max} to the segment $\overline{p_0p_2}$, getting the maximum distance $d_{1,1}$ and the index 1 of the corresponding point p_1 . We insert $d_{1,1}$ to S_{offset} at line 26 and 1 to S_{index} at line 27 at position 0, making $S_{offset} = \{d_{1,1}\}$ and $S_{index} = \{1\}$. For the 2nd time through the for loop at line 18, we compute the distances of points p_3 and p_4 to segment $\overline{p_2p_5}$, getting the maximum offset $d_{1,2}$ and the index of the corresponding point, which is 4. We then insert $d_{1,2}$ and 4 to S_{offset} and S_{index} at position 1, respectively. After the first time through the while loop, we have: $P_{chosen} = \{0, 2, 5\}$, $S_{offset} = \{d_{1,1}, d_{1,2}\}$, $S_{index} = \{1, 4\}$ and $|P_{chosen}| = 3$.
3. In the second iteration of the while loop, the maximum element in S_{offset} is $d_{1,1}$, the *index* of which is 0. At line 13, the index I_{chosen} of the maximum distance point is $S_{index}[0] = 1$. We insert 1 to P_{chosen} between elements 0 and 2, and I_{insert} becomes 0 (line 16). We next remove the element at position 0 in S_{index} and S_{offset} . Since the points p_0 , p_1 and p_2 are adjacent in P , we insert 0.0 to S_{offset} at position 0 and 1, and insert -1 to S_{index} at position 0 and 1. After the second iteration through the while loop, we have : $P_{chosen} = \{0, 1, 2, 5\}$, $S_{offset} = \{0, 0, d_{1,2}\}$, $S_{index} = \{-1, -1, 4\}$ and $|P_{chosen}| = 4$.

4. In the third while loop iteration, the maximum element in S_{offset} is $d_{1,2}$ with corresponding index 2. The index I_{chosen} of this maximum offset point is $S_{index}[2] = 4$. We insert 4 to P_{chosen} between elements 2 and 5, and I_{insert} becomes 2. We remove the element at position 2 in S_{offset} and S_{index} . For point p_3 , the only one left between p_2 and p_4 , we compute its distance $d_{2,1}$ to the segment $\overline{p_2p_4}$. We insert $d_{2,1}$ to S_{offset} at position 2, and insert 3 to S_{index} at position 2. Since p_4 and p_5 are adjacent, we insert 0.0 to S_{offset} , and -1 to S_{index} at position 3, respectively. After the third while loop iteration, we have: $P_{chosen} = \{0, 1, 2, 4, 5\}$, $S_{offset} = \{0, 0, d_{2,1}, 0\}$, $S_{index} = \{-1, -1, 3, -1\}$ and $|P_{chosen}| = 5$.
5. Now we have $|P_{chosen}| = 5$ which is equal to w_B , so the algorithm ends, and the simplified polygon $P' = \{p_0, p_1, p_2, p_4, p_5\}$ is returned. The simplified polygon is shown in Figure 3.10 (b).

In this way, we just add one more point to the chosen point set in each while loop iteration. In the k_{th} loop, there are $k + 2$ points in the chosen point set, and the polygon has been broken into $k + 1$ segments. In the while loop, we just need to recompute the offsets of points in P that have an index between I_L and I_{chosen} , and between I_{chosen} and I_R . Segment S_L is defined by two points having indices I_L and I_{chosen} . Segment S_R is defined by two points having indices I_{chosen} and I_R .

Lemma 3.2.1. *Using the **PackPolygon** algorithm, the worst case time required to simplify a simple polygon P containing w points into a polygon P' containing w_B points ($w > w_B$) is $O(w_B w)$.*

Proof. For an input polygon having w points where $w > w_B$, the algorithm stops when we get w_B points in the chosen point set. In the worst case, for points that have not yet been added to the chosen point set, we select the left-most or right-most one to add each time.

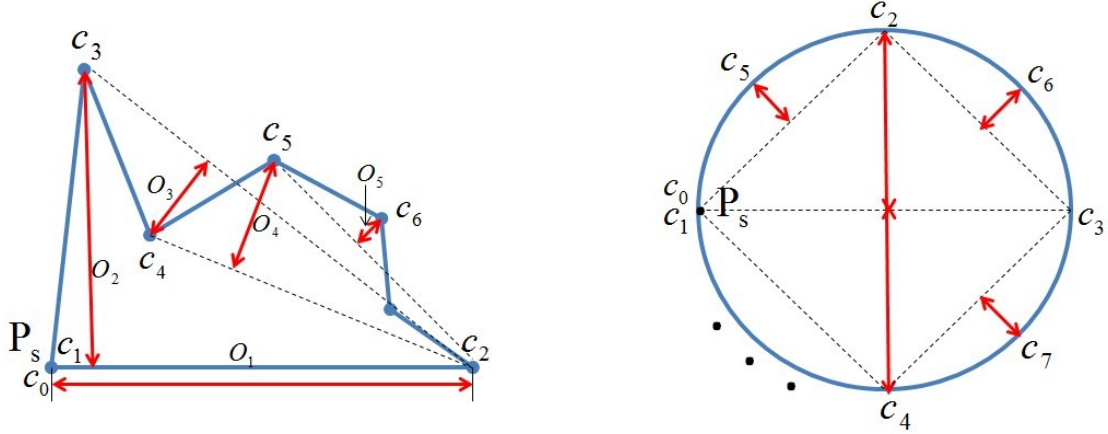
In the k_{th} loop, we need to recompute the offsets of all the left $w - (k + 2)$ points that are not in the chosen point set. The time \mathcal{T}_w for PackPolygon algorithm in the worst case can be computed as following:

$$\begin{aligned}
\mathcal{T}_w &= (w - 2) + (w - 3) + \cdots + (w - (k + 2)) + \cdots + (w - w_B) \\
&= (w - 2) + (w - 3) + \cdots + 1 - [(w - w_B - 1) + (w - w_B - 2) + \cdots + 1] \\
&= \frac{(w-2)(w-2+1)}{2} - \frac{(w-w_B-1)(w-w_B)}{2} \\
&= \frac{(w^2-3w+2)-(w^2-(2w_B+1)w+w_B(w_B+1))}{2} \\
&= (w_B - 1)w + 1 - \frac{w_B^2 + w_B}{2} \\
&= w_B w + 1 - (w + \frac{w_B^2}{2} + \frac{w_B}{2})
\end{aligned}$$

So we have $\mathcal{T}_w \in O(w_B w)$ □

If we treat w_B as a constant, for different polygons, the worst case time efficiency is $O(w)$. An example of the worst case is shown in Figure 3.11 (a). Usually, the start point and end point are the same point geometrically for a polygon, marked as P_s in this example. Assuming that $B = 7$, c_i is used to indicate the i_{th} point that is added to the chosen point set. The start and end points are added to P_{chosen} in the initialization, which are marked as c_0 and c_1 . We use o_j to indicate the maximum offset in the vector S_{offset} in the j_{th} while loop iteration. After c_2 is added to P_{chosen} , the remaining points are added on the left side, which is the first worst-case scenario of Lemma 3.1.1.

Assume the points of a polygon are equally distributed along a circle, as shown in Figure 3.11 (b). For points that have equal offsets to their segments, we add the first one we encounter to the chosen point set each time. In Figure 3.11 (b), c_i indicates the i_{th} point added to the chosen point set. For an input polygon having w equidistant points, the time \mathcal{T}_e for the PackPolygon algorithm can be computed as following:



(a) Worst case example.

(b) Equidistant distributing example.

Figure 3.11: Two examples of the PackPolygon algorithm. Figure (a) shows the worst case example. Figure (b) shows the example when the points are equally distributed along a circle.

$$\mathcal{T}_e = \underbrace{(w-2) + 2\left(\frac{w-3}{2}\right) + 4\left(\frac{w-5}{4}\right) + \cdots + 2^k\left(\frac{w-1-2^k}{2^k}\right)}_{(w_B-1) \text{ items in total}}$$

Initially, we add 2 points (start and end point) to the chosen point set at one time. Then in each while loop, we just add one more point in the chosen point set. When the algorithm ends, we have exactly w_B points in the chosen point set, so we have:

$$w_B - 1 = 2^0 + 2^1 + 2^2 + \cdots + 2^{k-1} + c$$

$$\text{where } 2^{k-1} < c \leq 2^k$$

Assuming, without loss of generality, that k is the largest positive integer value $\leq \log_2 w_B - 1$, we have:

$$\begin{aligned} \mathcal{T}_e &\leq (w-2) + 2\left(\frac{w-3}{2}\right) + 4\left(\frac{w-5}{4}\right) + \cdots + 2^k\left(\frac{w-1-2^k}{2^k}\right) \\ &= (w-1-1) + (w-1-2) + (w-1-2^2) + \cdots + (w-1-2^k) \\ &= (k+1)(w-1) - (1+2+2^2+\cdots+2^k) \end{aligned}$$

$$\begin{aligned}
&= (k+1)(w-1) - \frac{1(1-2^{k+1})}{1-2} \\
&= (k+1)(w-1) - (2^{w+1} - 1)
\end{aligned}$$

Thus, we have:

$$\mathcal{T}_e \leq \log_2 w_B (w-1) - (w_B - 1) \in O(w \log_2 w_B)$$

This proves the following lemma:

Lemma 3.2.2. *For a simple polygon P containing w points that are equally distributed along a circle, the time \mathcal{T}_e for simplifying it into a polygon P' containing w_B points ($w > w_B$) using the **PackPolygon** algorithm is $O(w \log_2 w_B)$.*

If we treat w_B as a constant, then for polygons defined by a sequence of points distributed equidistant along a circle, the time efficiency for the **PackPolygon** algorithm is $O(w)$.

The complete polygon simplification algorithm is shown in Algorithm 3.2.

Algorithm 3.2: PolygonSimplification(point[] P)

Input:

point[] P : an ordered point set for describing a polygon;
int w_B : the maximum number of points fitting on one disk block;

Output:

point[] P' : an ordered point set for the simplified polygon ;

- 1 **if** $P.length < w_B$ **then**
- 2 $Q = P$
- 3 **else**
- 4 point[] $P_{shifted} \leftarrow \text{ShiftPolygon} (P) ;$
- 5 $P' \leftarrow \text{PackPolygon}(P_{shifted}, w_B) ;$
- 6 **return** P' ;

Note that in PolygonSimplification algorithm, we call the procedure **ShiftPolygon**, which is shown in Algorithm 3.3, before the procedure **PackPolygon** is called. In the Algorithm **ShiftPolygon**, we first compute the distances between any two points in point

set P , and find out the two points indexed by I_{min} , I_{max} that have the greatest distance between them. Then we shift the points in P , letting it start from the point with the index I_{min} . The shifted polygon is returned as $P_{shifted}$. The shifted polygon is used as the input of the `PackPolygon` algorithm. In doing so, we can guarantee that in the `PackPolygon` algorithm, the point added to P_{chosen} in the first while loop iteration has the maximum distance to the start point $P[0]$ among all the point pairs in P . This can, in turn, optimize the simplified results of the `PackPolygon` algorithm in some cases, as illustrated below.

Algorithm 3.3: ShiftPolygon(point[] P)

Input:

point[] P is an ordered point set for describing a polygon

Output:

point[] $P_{shifted}$: int I_{min} and int I_{max} are the indexes for the two points having the greatest distance between them in P , $P_{shifted}$ is the shifted point set that contains the same points as P , but starts from I_{min} ;

```

1 double  $max \leftarrow 0$ ;
2 int  $I_{min} \leftarrow -1$ ,  $I_{max} \leftarrow -1$  ;
3 for int  $i \leftarrow 0$  to  $P.length - 2$  do
4   for int  $j \leftarrow i + 1$  to  $P.length - 1$  do
5     double  $dist \leftarrow \text{distance}(P[i], P[j])$  ;
6     if  $dist > max$  then
7        $max \leftarrow dist$ ,  $I_{min} \leftarrow i$ ,  $I_{max} \leftarrow j$  ;
8 for int  $m \leftarrow 0$  to  $P.length - I_{min} - 1$  do
9    $P_{shifted}[i] \leftarrow P[i + I_{min}]$  ;
10 for int  $i \leftarrow P.length - I_{min}$  to  $P.length - 2$  do
11    $P_{shifted}[i] \leftarrow P[i - P.length + I_{min} + 1]$  ;
12  $P_{shifted}[P.length - 1] \leftarrow P_{shifted}[0]$  ;
13 return  $P_{shifted}$  ;
```

The example in Figure 3.12 shows the effect of the `ShiftPolygon` algorithm. The original polygon P containing 11 points is shown in Figure 3.12 (a). Assuming that $w_B = 6$, we directly run the `PackPolygon` algorithm for the input P and w_B , as a result, the simplified

polygon $P' = p_0, p_1, p_3, p_7, p_9, p_{10}$ is returned, which is shown in Figure 3.12 (b). In Figure 3.12 (a), o_i indicates the maximum offset in the S_{offset} vector in the i_{th} while loop iteration. If we call the ShiftPolygon algorithm before PackPolygon, the resulted polygon $P_{shifted}$ is as shown in Figure 3.12 (c), in which p_0 and p_6 consist the points pair that has the maximum distance between them. We then call the PackPolygon procedure for $P_{shifted}$ and w_B , the resulted simplified polygon P'_s is shown in Figure 3.12 (d). As we can see, comparing with P' , P'_s can better reflect the features of the original polygon P .

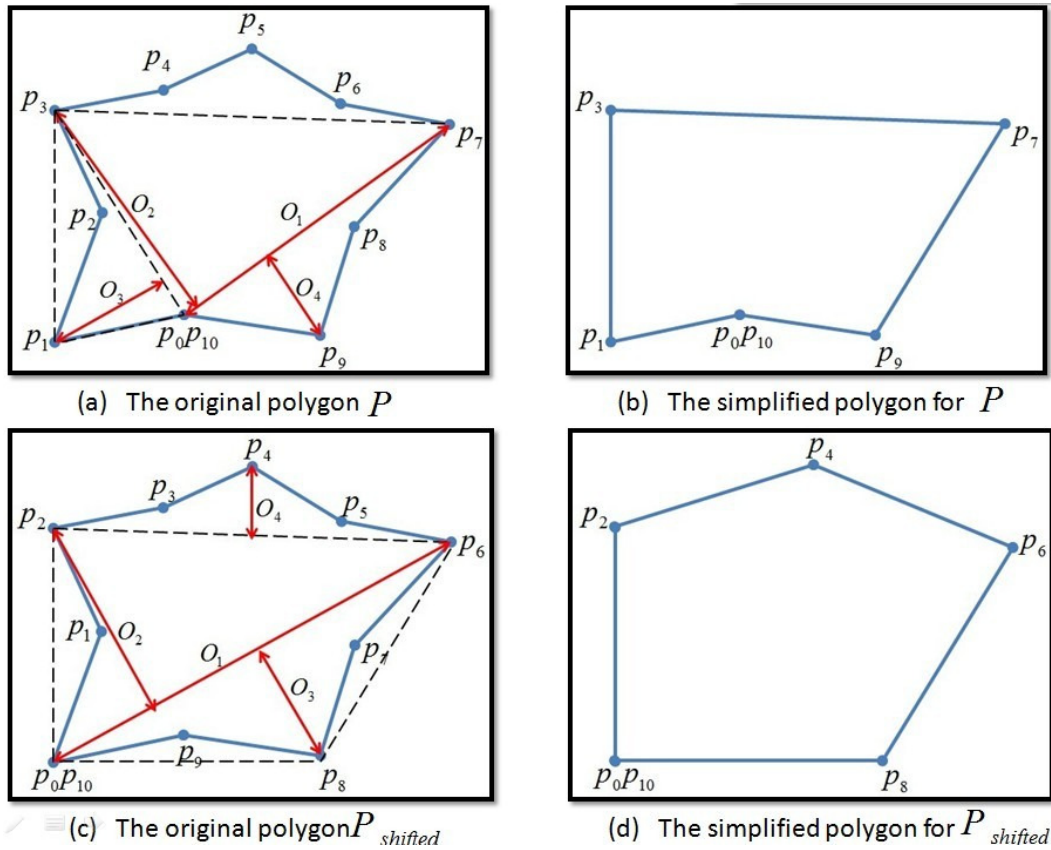


Figure 3.12: Example showing the effect of the ShiftPolygon algorithm with $w_B = 6$.

Figure 3.13 shows the comparison of the polygon data before and after simplification for counties of New Brunswick, Canada. The detailed comparisons for the area 1 and area 2 in Figure are shown in Figure 3.14 (a) and (b), respectively. These figures are generated

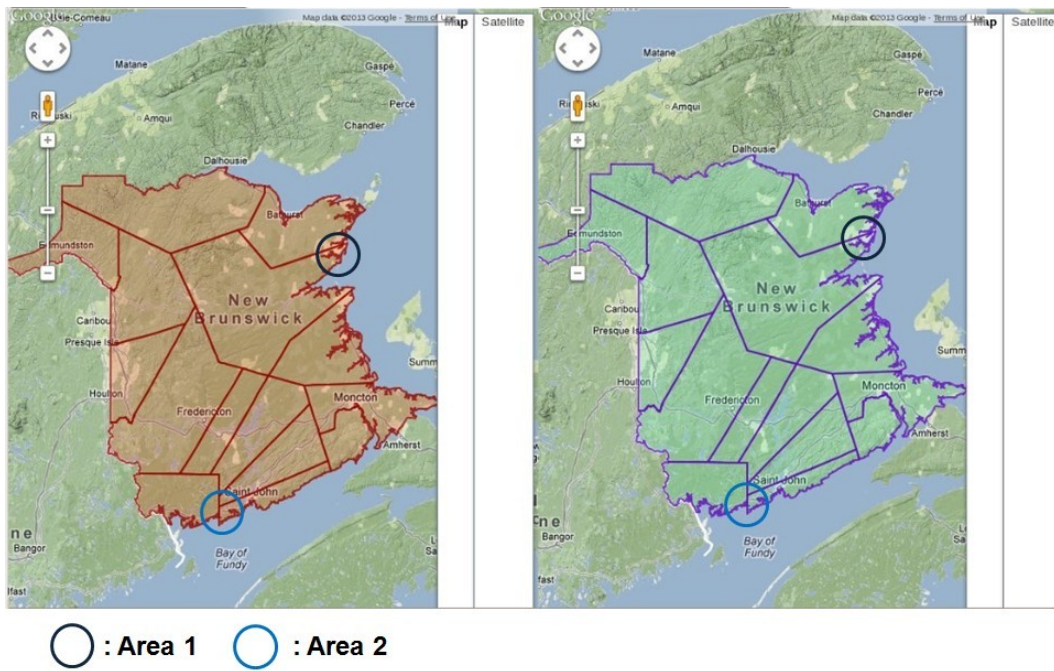


Figure 3.13: The comparison of the polygon data before and after simplification for counties of New Brunswick, Canada on Google Map. The map on the left side shows the original polygon data, the map on the right side shows the corresponding simplified data with $w_B = 500$.

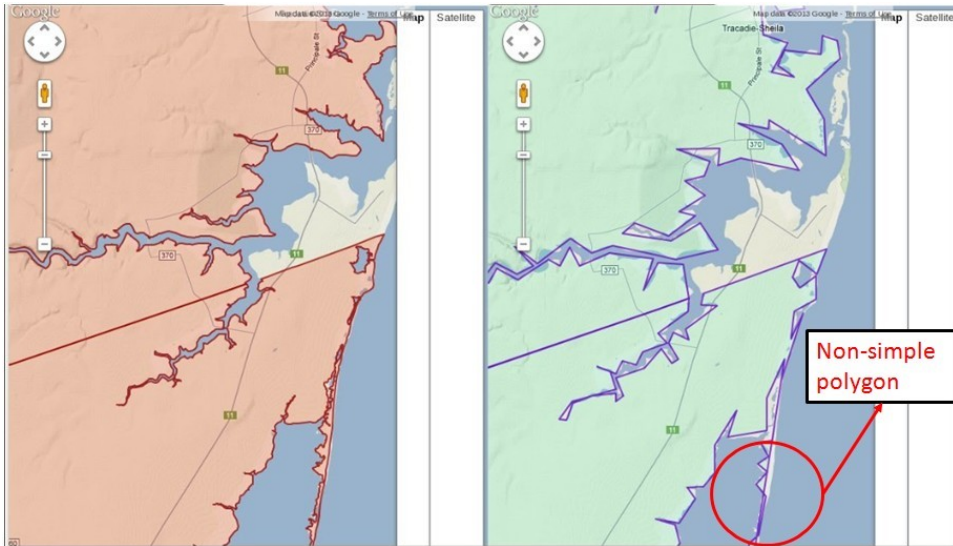
from the latitude and longitude values of county boundaries using the Google Maps API.

There are 15 county boundaries shown in Figure 3.13. Their corresponding county names, the number of (ϕ, λ) pairs in the original data, and the number of (ϕ, λ) pairs in the simplified data are listed in Table 3.1. As we can see, using our `PolygonSimplification`

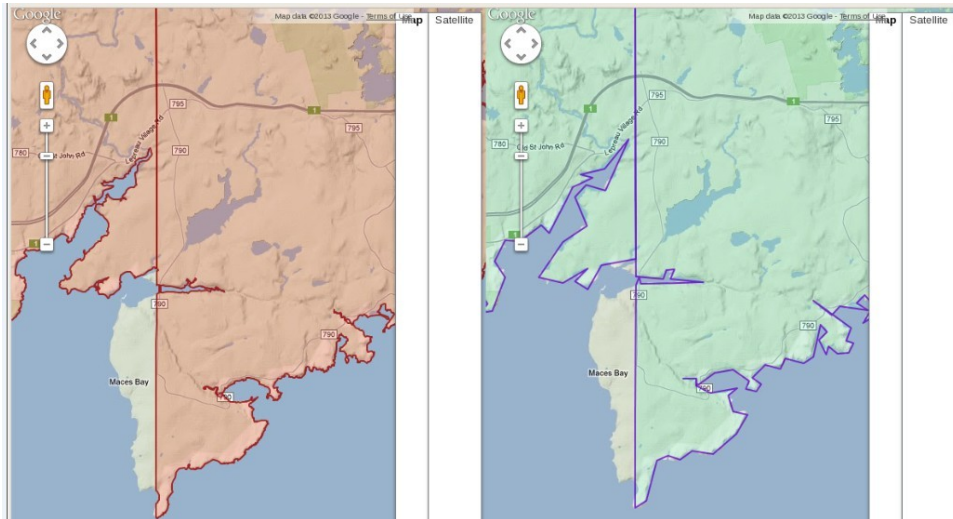
Table 3.1: The number of points for describing the county boundaries before and after the `PolygonSimplification` algorithm.

County name	Num of points in original polygon	Num of points in simplified polygon
Saint John	30412	500
Charlotte	2088	500
Sunbury	20158	500
Queens	1593	500
Kings	49404	500
Albert	38759	500
Westmorland	472	472
Kent	511	500
Northumberland	44277	500
York	1133	500
Carleton	1902	500
Victoria	27756	500
Madawaska	454	454
Restigouche	13429	500
Gloucester	25774	500

algorithm, the number of (ϕ, λ) pairs in the polygon data is significantly decreased, while the important features for the polygons are well maintained. This, in turn, guarantees the I/O-efficiency in our spatial indexing as each polygon definition is stored on one disk block with $w_B = 500$.

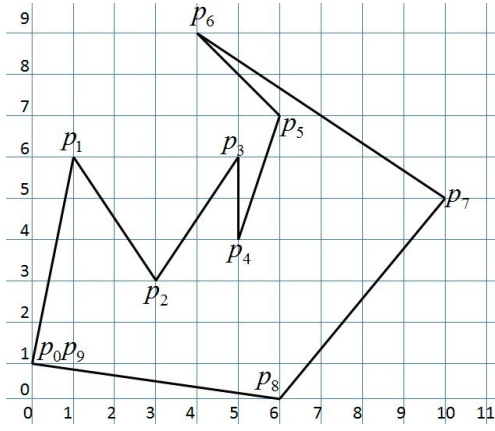


(a) The detailed comparison for area 1 in Figure 3.10.

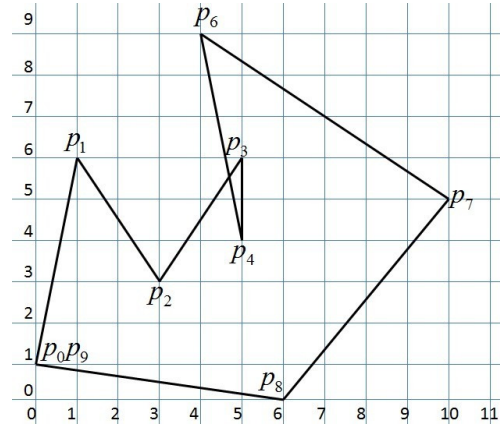


(b) The detailed comparison for area 2 in Figure 3.10.

Figure 3.14: The comparison of the polygon data before and after simplification for the area 1 and 2 in Figure 3.10. The map on the left side shows the original polygon data, the map on the right side shows the corresponding simplified polygon data.



(a) The original polygon



(b) The simplified polygon with $w_B = 9$.

Figure 3.15: An example of the PackPolygon algorithm with the input polygon P containing 10 points and $w_B = 9$.

3.2.3.4 PackPolygon algorithm can give a non-simple polygon for an input simple polygon

For an input simple polygon, PackPolygon algorithm can return a non-simple polygon. The following lemma is proven using an example.

Lemma 3.2.3. *The PackPolygon algorithm can produce a non-simple polygon from an input simple polygon.*

Proof. An example is shown in Figure 3.15. Assume that the input polygon P contains 10 points, the (x, y) coordinates of which are: $p_0(0, 1)$, $p_1(1, 6)$, $p_2(3, 3)$, $p_3(5, 6)$, $p_4(5, 4)$, $p_5(6, 7)$, $p_6(4, 9)$, $p_7(10, 5)$, $p_8(6, 0)$ and $p_9(0, 1)$. We use algorithm PackPolygon for P with $w_B = 9$. Initially, 0 and 9 are appended to P_{chosen} . Since p_7 has the biggest distance 10.77 to p_0 , we append 7 to S_{index} and 10.77 to S_{offset} . The changes of the vectors P_{chosen} , S_{offset} and S_{index} in each while loop iteration are listed as follows:

Iteration 1

Point p_7 is chosen to add. $P_{chosen} = \{0, 7, 9\}$, $S_{index} = \{6, 8\}$, $S_{offset} = \{5.94, 3.16\}$.

Iteration 2

Point p_6 is chosen to add. $P_{chosen} = \{0, 6, 7, 9\}$, $S_{index} = \{4, -1, 8\}$, $S_{offset} = \{3.13, 0, 3.16\}$.

Iteration 3

Point p_8 is chosen to add. $P_{chosen} = \{0, 6, 7, 8, 9\}$, $S_{index} = \{4, -1, -1, -1\}$, $S_{offset} = \{3.13, 0, 0, 0\}$.

Iteration 4

Point p_4 is chosen to add. $P_{chosen} = \{0, 4, 6, 7, 8, 9\}$, $S_{index} = \{1, 5, -1, -1, -1\}$, $S_{offset} = \{3.77, 1.56, 0, 0, 0\}$.

Iteration 5

Point p_1 is chosen to add. $P_{chosen} = \{0, 1, 4, 6, 7, 8, 9\}$, $S_{index} = \{-1, 2, 5, -1, -1, -1\}$, $S_{offset} = \{0, 1.79, 1.56, 0, 0, 0\}$.

Iteration 6

Point p_2 is chosen to add. $P_{chosen} = \{0, 1, 2, 4, 6, 7, 8, 9\}$,
 $S_{index} = \{-1, -1, 3, 5, -1, -1, -1\}$, $S_{offset} = \{0, 0, 1.79, 1.56, 0, 0, 0\}$.

Iteration 7

Point p_3 is chosen to add. $P_{chosen} = \{0, 1, 2, 3, 4, 6, 7, 8, 9\}$,
 $S_{index} = \{-1, -1, -1, -1, 5, -1, -1, -1\}$, $S_{offset} = \{0, 0, 0, 0, 1.56, 0, 0, 0\}$.

Now $|P_{chosen}| = 9$ is equal to w_B , so the algorithm stops. The output polygon $P' = \{p_0, p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_9\}$ is shown in Figure 3.15 (b). As we can see, the resulting polygon P' is a non-simple polygon although the input polygon is a simple one. \square

Figure 3.14 (a) also shows a non-simple polygon arising from a input simple polygon.

The above example also applies to the RDP algorithm, from which we can observe the following corollary:

Corollary 3.2.1. *The Ramer-Douglas-Peucker algorithm can give a non-simple polyline for an input simple polyline.*

Proof. An example for Corollary 3.2.1 uses the polyline P_L consisting of the points from p_0 to p_7 in Figure 3.15 (a). We use the RDP algorithm (see section 3.2.3.3) for the input P_L and $\epsilon = 1.6$. The simplified polyline P'_L contains points $p_0, p_1, p_2, p_3, p_4, p_6$ and p_7 , which is a non-simple polyline. □

Chapter 4

Indexing

We used a suffix tree and a packed R^* tree to realize the text + spatial search, as shown in Figure 4.1. We index the full text of the .html web pages in the collection and generate a suffix tree and its associated data structures. To generate an R^* tree, if the latitude and longitude pair (ϕ, λ) exists, we index the spatial data by this; otherwise, we index by the polygon of the county or province where the object is located. While building the packed R^* tree, an R^* tree leaf node hash table is generated at the same time. The hash table links the R^* -tree with the suffix tree as explained in section 4.2.

4.1 Text Indexing

We use a suffix tree [22] for indexing English language text from a database. Figure 4.2 shows an example suffix tree for the phrase “the cat in the hat”. In the text index, we first skip all the stop words, such as “the”, “a”, and “and”. The complete list of the stop words is given in Appendix A. In this example, we will then get the compact phrase “cat in hat”. After that, to speed up the text search, we index all the possible sub-phrases of this compact phrase. In this example, we have 6 sub-phrases in total; they are: “cat”,

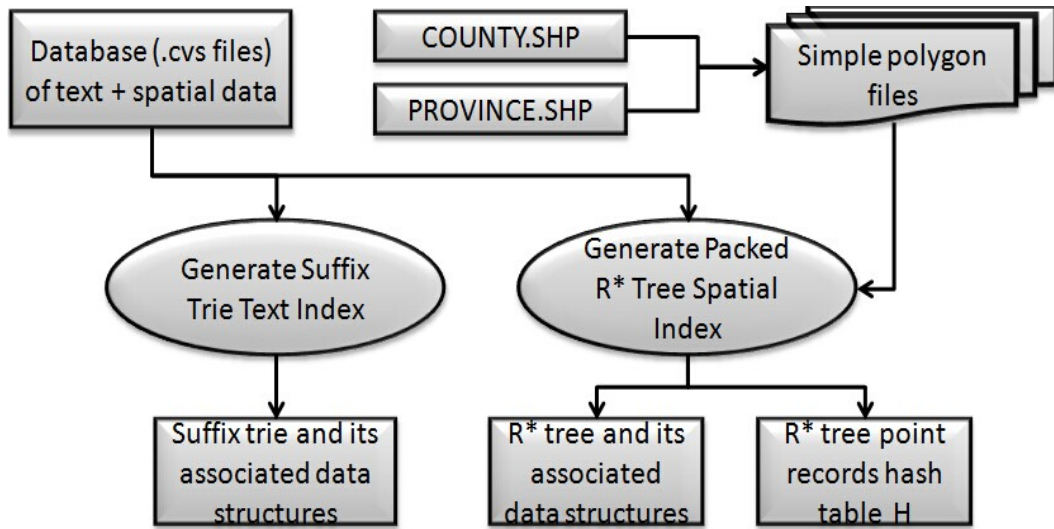


Figure 4.1: The text + spatial indexing scheme using an R* tree and a suffix tree.

“cat in”, “cat in hat”, “in”, “in hat” and “hat”.

All sub-phrases are stored in the leaf nodes of our suffix tree. Figure 4.3 shows the structure of a suffix tree leaf node.

4.1.1 Suffix tree Leaf Nodes

We use the data structure `SuffixTrieDataObject` to represent the data object stored in the leaf, which contains a string for the corresponding sub-phrase, the integer value of the document frequency and a vector of `DocObjects` in which the sub-phrase appears. The document frequency is a counter of the total number of documents containing a specific subphrase. `DocObject` is a data structure that indicates the relationship between the stored sub-phrase and a document containing this sub-phrase. A `DocObject` consists of the text of the subphrase, the primary key (e.g. record number) for the document and the `termFrequency`, defined as the count of the number of times the text shows up in this document. The `Primary Key` is a string containing the database record accession number. The complete text for each indexed document is stored in a text array as discussed next.

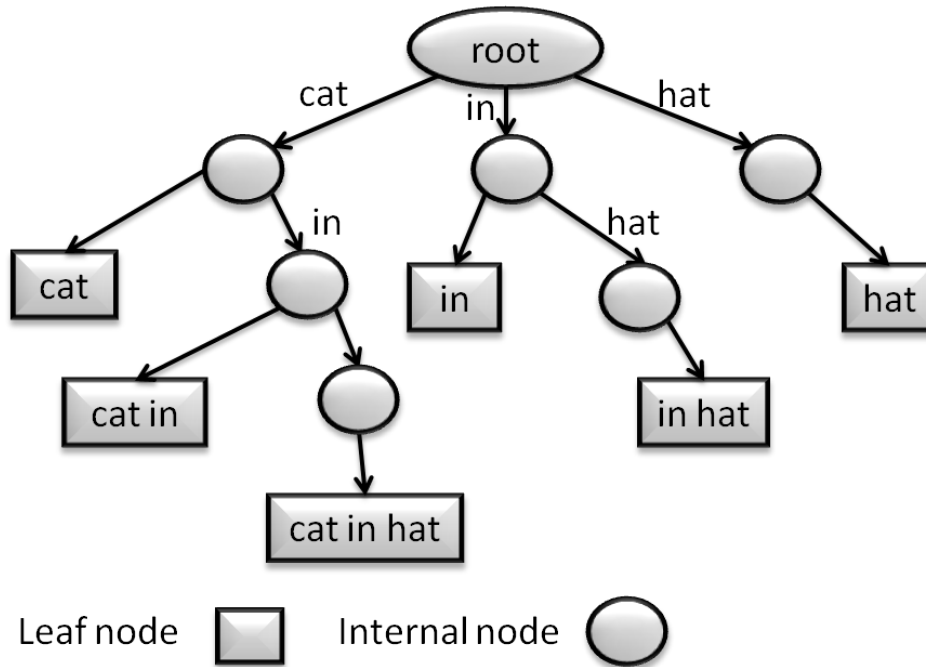


Figure 4.2: An example suffix tree.

4.1.2 Constructing a suffix tree

We implement the Patricia trie template class from [31] for constructing a suffix tree. Figure 4.4 shows the sequence diagram for the suffix tree constructor. To build the suffix tree, we first get the stop words list. In computing, stop words are words which are filtered out prior to, or after, processing of natural language data (text) [42]. The English stopwords list is obtained from [6]. Then the program gets the document input from the input file via `getDocInput(input_file)`, where the `input_file` is a `.csv` file as described in section 3.1, and generates a list of document contents. Each document's text is stored in one element of the `doc.text_array`. The suffix tree is built afterwards.

For each entry of the `doc.text_array`, there are 22 fields as described in section 3.1, and

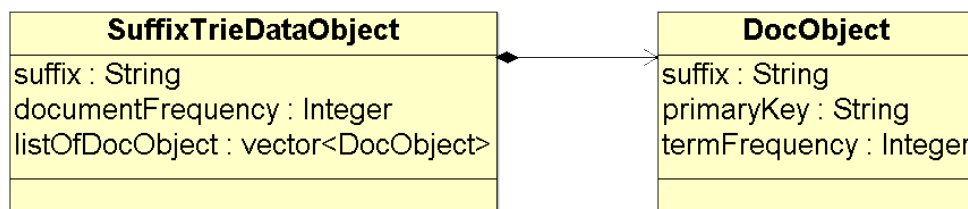


Figure 4.3: The structure of a suffix tree leaf node.

each field contains multiple phrases. We first transform each character in one document text to lower case using `convertToLower(str_doc)`. The complete text for one document is then split into 22 fields by the function `getSingleField(str_doc)` using the "," as a field separator. For each field, we get single phrases by splitting the fields with separators ";", ":" and "," using `getSinglePhrase(str_field)`. For each phrase, we then remove the punctuation such as "&", "\", "." and parentheses to arrive at a compact phrase. Then we get a list of single words using `getSingleWord(str_compact_phrase)` by separating the compact phrase by spaces. The function `getAllSubphrases` will then generate a list of all possible subphrases consisting of the single words. In the inner loop, if the subphrase contains only one word, the program will check if the word is in the stopwords list first; if it is, this single word will not be inserted to the suffix tree. We then insert all the other subphrases to the suffix tree.

Here we show an example of constructing a suffix tree . An example Herbarium database record is as follows:

```

"283","Prenanthes trifoliolata (Cass.) Fern. ","45.9635895","-66.6431151","Taylor,
A.R.A.",",",
"14","9","1946",",", "roadside of woodlot", "Fredericton", "York", "NB", ",", "Gall-of-the-earth",
"Asteraceae", "Pattes d'oe", "Specimen has Fruit but no Flowers", ",", ",", "Fredericton", ",",
  
```

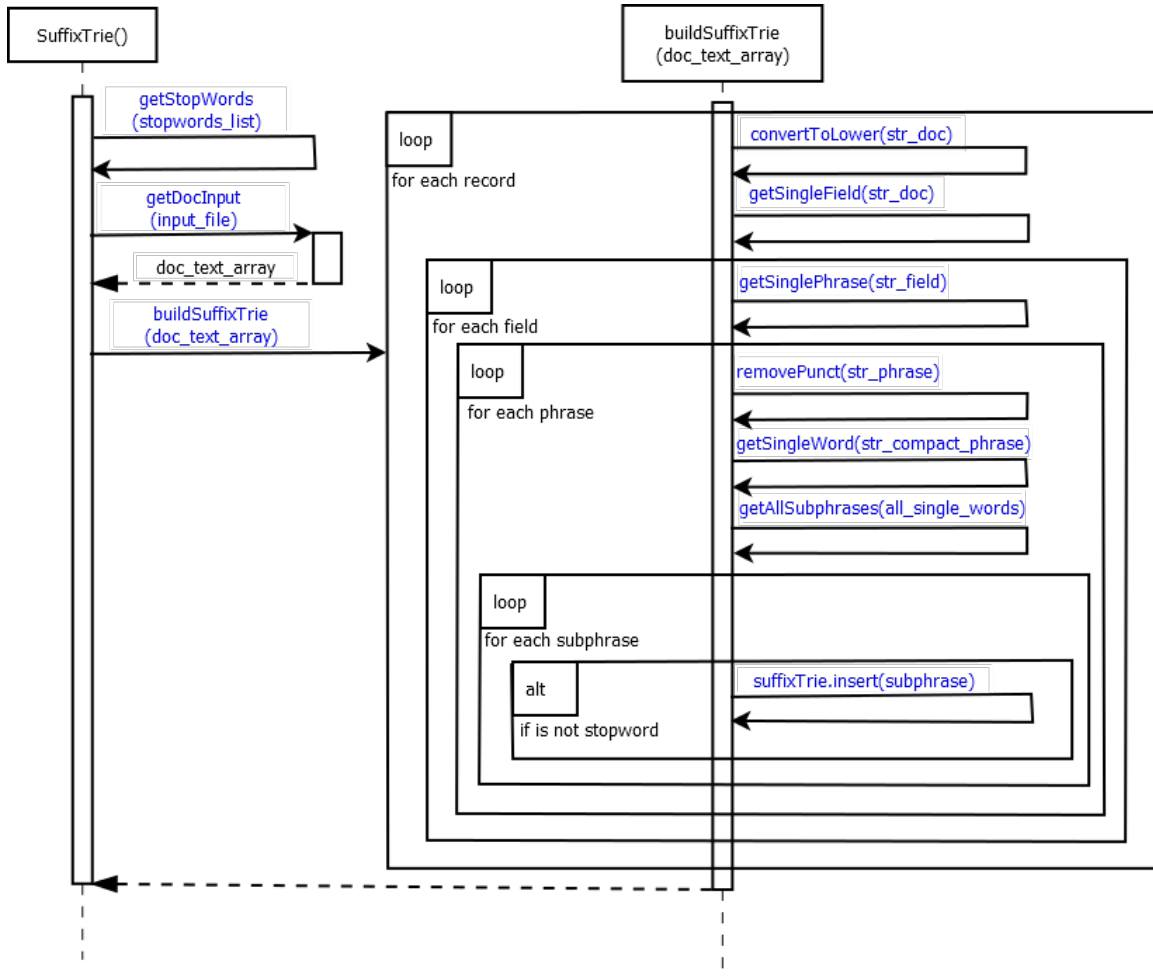



Figure 4.4: The sequence diagram for constructing a suffix tree.

"Scirpus americanus auct. non Pers.; Scirpus pungens"

We take the last field as an example, the content of which is "scirpus americanus auct. non pers.; scirpus pungens" (the characters have been transformed to lower cases by `convertToLower`). The field content is split into two phrases "scirpus americanus auct. non pers." and "scirpus pungens" first. The punctuation in the phrases are then removed, and we generate the compact phrases "scirpus americanus auct non pers" and "scirpus pungens". For the phrase "scirpus americanus auct non pers", we get the list of single words by splitting it with spaces. The generated list is: "scirpus", "americanus", "auct",

"non", "pers". After that, the list of all possible subphrases is generated, which is:
"scirpus", "scirpus americanus", "scirpus americanus auct", "scirpus americanus auct non",
"scirpus americanus auct non pers.", "americanus", "americanus auct", "americanus auct
non", "americanus auct non pers", "auct", "auct non", "auct non pers", "non ", " non pers",
"pers"

Let A be the number of single words, there are always $\frac{A(A+1)}{2}$ subphrases.

Since there is no single stopword in the list, we directly insert all of the subphrases in the list to the suffix tree. For the second phrase "scirpus pungens", we go through the same process and get the three subphrases:

"scirpus ", "scirpus pungens", "pungens"

There is no single stopword in the list, so we insert all three subphrases to the suffix tree.

4.1.3 Querying a Suffix Tree

Figure 4.5 shows the sequence diagram for a Q1 query $Q(t)$, where t is the query string. We first transfer the the query string to an all lower case query string, and remove the punctuation in the string, obtaining a compact query string. The program then invokes `Q1InternalSearch`, which uses the compact query string as the input parameter. The `Q1InternalSearch` method first generates the subphrases list of the compact query string, using the same method as described in subsection 4.1.2. For each subphrase, we perform a tree look up, which will return a suffix tree data object if this subphrase has been indexed. As shown in Figure 4.3, each suffix tree data object contains an integer value of document frequency and a vector of `DocObjects`. Since one `DocObject` can show up in the query results of several different subphrases, we build a reverse document-subphrases hashtable for each document as described in section 4.1.3.1.

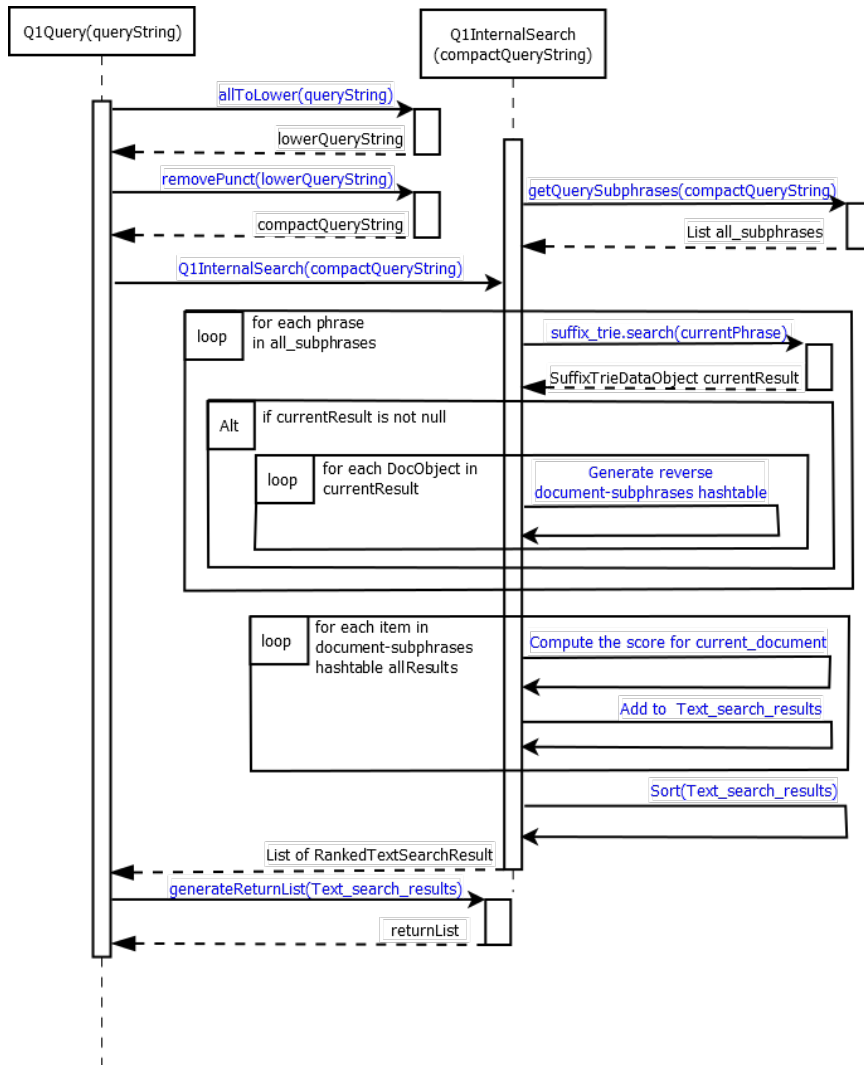


Figure 4.5: The sequence diagram for a Q1 query.

4.1.3.1 Reverse Document-Subphrases Hashtable

Figure 4.6 shows the structure for constructing the reverse document-subphrases hashtable. The data object `SubphraseResult` contains the text of the corresponding subphrase and the factors indicating the relationship between a document d and a subphrase b .

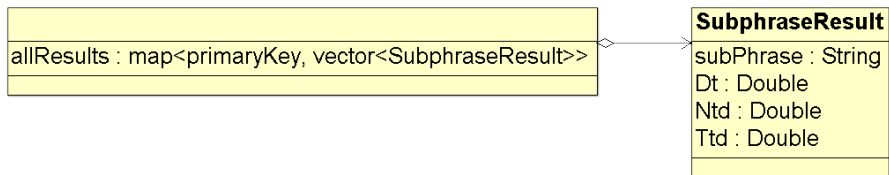


Figure 4.6: The data structure for constructing the reverse document-subphrases hashtable.

D_b stands for the number of documents containing the specific subphrase b , i.e. the `documentFrequency` in the corresponding `SuffixTrieDataObject`. The parameter T_{bd} is defined as follows:

$$T_{bd} = \sqrt{\text{termFrequency}} \quad (4.1)$$

where `termFrequency` is the number of times the `SubPhrase` text appears in the document (here called a record). N_{bd} encapsulates the length factor, defined as the square root of the number of words in the subphrase b . These three factors are used to compute the scores in the ranking process. Each generated `SubphraseResult` for a Q query refers to a specific document for which D_b , N_{bd} and T_{bd} are computed.

As shown in Figure 4.6, `allResults` is a hashtable that relates a specific document to a vector of `SubphraseResults` by using the document primary keys as the hashtable key value. While performing a Q1 query, when searching for a subphrase b , if we get a non-empty `SuffixTrieDataobject` returned, it provides a list of `DocObjects` at the same time. For each `DocObject` in the list, we then create a `SubphraseResult` object, and

insert to the hashtable `allResults`.

Figure 4.7 shows an example of how to generate the reverse document-subphrases hashtable. Assume that the tree structure is as shown in Figure 4.2. We now query the suffix tree with the string “in hat”, ignoring the single stopword “in”. The Q1 query now has two subphrases “in hat” and “hat” in the list of `all_subphrases`. Following the path in the suffix tree, two nodes “in hat” and “hat” are found, with their corresponding `SuffixTrieDataObject` returned as `currentResult`. The generated reverse document-subphrases hashtable `allResults` is as shown in the Figure 4.7. For the `DocObject` d_1 , we initially find there is no entry having the key value “33” in `allResults`. We create a new vector containing only one `SubphraseResult` s_1 and add it to hashtable `allResults`. We then search for the subphrase “hat” and get two `DocObjects` returned. For `DocObject` d_2 , we look for the key value “33” in `allResults`, and find the key value already exists. We then create a `subphraseResult` s_2 and insert to the corresponding vector. For the `DocObject` d_3 , we find no entry with key value “65” exists in `allResult`, so we add a new vector containing only one element s_3 with the key value “65” to `allResults`.

4.1.3.2 Ranking text search results

After getting the document-subphrases hashtable for query $Q(t)$, we need to analyze the data stored in the hashtable to get the final ranking.

We use the Lucene ranking algorithm [34] to compute the scores of text search results. Apache Lucene [3] is an open source information retrieval (IR) software library, originally created by Doug Cutting [14]. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform [3]. Lucene provides a scoring algorithm to find the best matches to document queries, which ranks documents resulting from a

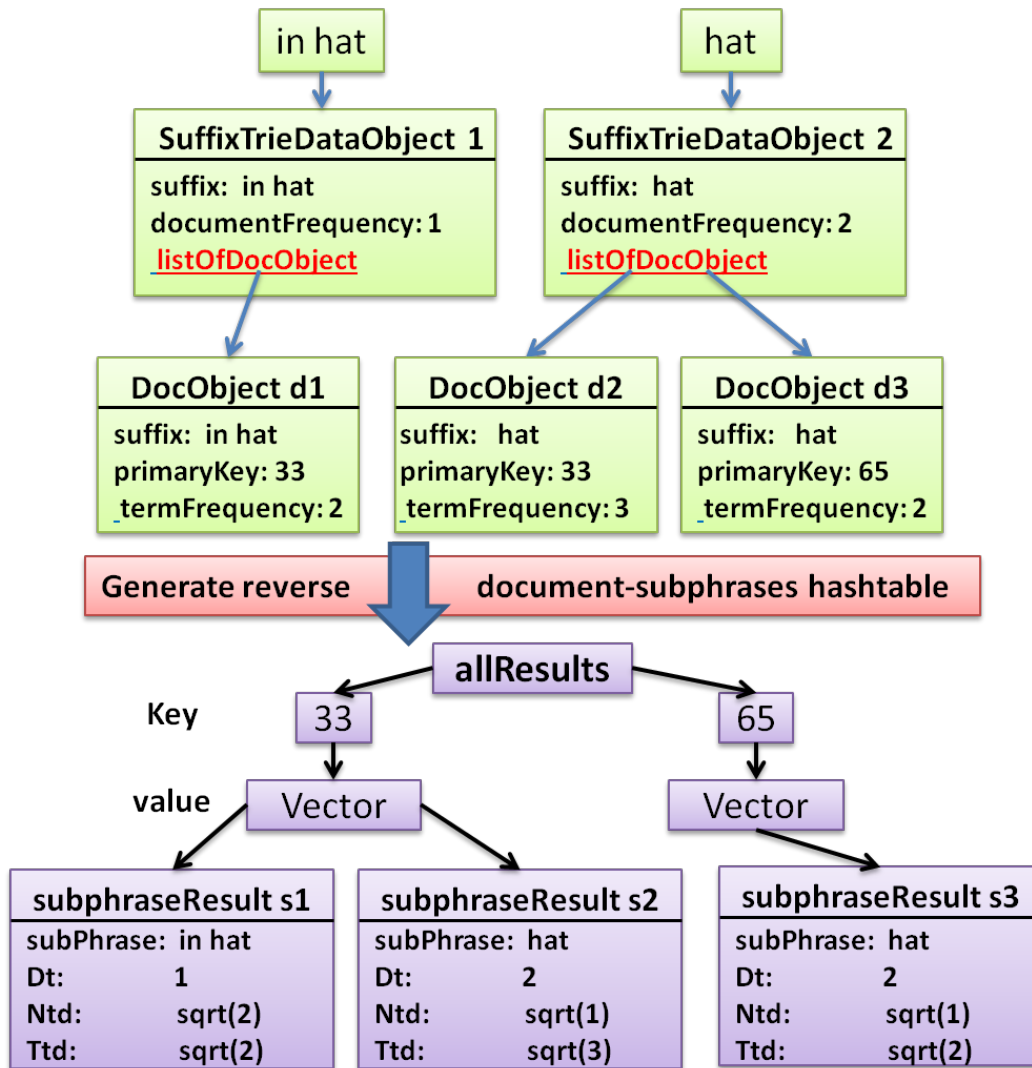


Figure 4.7: An example of the generating of the reverse document-subphrases hashtable.

search query based on their content. The default scoring algorithm considers such factors as the frequency of a particular query term within individual documents and the frequency of the term in the total population of documents. The Lucene scoring algorithm considers the rarity of a matched term within the global space of all terms for a given field. In other words, if you match a term that is not very common in the data then this match is given a higher score [15].

We modified the Lucene ranking algorithm to adapt to our system. The original version can be found in [34]. To illustrate Lucene’s scoring algorithm, we have definitions as follows:

Definition 4.1.1.

m: number of subphrases in the query string *t*

D: set of records in the index, and $n = |D|$

b: a possible subphrase in the query string *t*

D_b: number of documents containing the specific subphrase *b*

Let R_{td} stand for the ranking score of query *t* for a document *d*, R_{td} is computed as follows:

$$R_{td} = C_{td}N_t \sum_{\forall b \in t} (T_{bd}I_b^2 B_b N_{bd}) \quad (4.2)$$

The parameters influencing the score are as follows:

1. As described in 4.1.3.1, T_{bd} is computed as:

$$T_{bd} = \sqrt{\text{termFrequency}} \quad (4.3)$$

Documents that have more occurrences of a given *b* receive a higher score.

2. I_b stands for inverse document frequency, which can be computed as:

$$I_b = 1 + \log\left(\frac{n}{D_b + 1}\right) \quad (4.4)$$

where D_b stands for the number of documents containing subphrase *b*, as described in section 4.1. Rarer subphrases give a higher contribution to the total ranking score, as $I_b \geq 1.5$ and the term I_b^2 appears in equation (4.2).

3. C_{td} is a score factor based on how many of the subphrases in t are found in the specific document d . Typically, a document that contains more of the query’s subphrases will receive a higher score than another document with fewer subphrases. C_{td} is computed as:

$$C_{td} = \frac{m_{td}}{m} \quad (4.5)$$

where m_{td} is the total number of subphrases of query string t found in a document d .

4. B_b is the user-specified boost factor on a subphrase b . In our system, we use the default boost factor of 1.
5. N_t is a normalizing factor used to make scores between queries comparable. This factor does not affect document ranking (since all ranked documents are multiplied by the same factor), but rather just attempts to make scores from different queries (or even different indexes) comparable. The default computation of N_t in Default-Similarity is [16]:

$$N_t = \frac{1}{\text{sumOfSquaredWeights}} \quad (4.6)$$

The sum of squared weights is computed as:

$$\text{sumOfSquaredWeights} = B_t^2 \sum_{\text{all}bint} (I_b B_b)^2 \quad (4.7)$$

B_t is the boost factor for query t , the default value of which is 1.

6. N_{bd} encapsulates two factors: document boost B_d and subphrase length normalization L_b . B_d is the boost factor for document d . L_b is computed in accordance with the number of words in subphrase b , so the longer subphrases contribute more to the score. The longer the matching subphrase is, the greater the matching document’s

score will be. L_b is implemented as follows:

$$L_b = \sqrt{\text{numOfWords}} \quad (4.8)$$

N_{bd} is computed as follows:

$$N_{bd} = B_d L_b \quad (4.9)$$

Assume that we use the default value 1 for all the boost factors, R_{td} can be computed as follows:

$$R_{td} = \frac{m_{td}}{m} \frac{1}{\sum_{\forall b \in t} (I_b)^2} \sum_{\forall b \in t} (\sqrt{\text{termFrequency}} I_b^2 L_b) \quad (4.10)$$

In Figure 4.5, the loop on the bottom shows the ranking process. For each item in `allResults`, we first compute the score for the `current_document`, then an object of `RankedTextSearchResult` is created. The structure of `RankedTextSearchResult` is shown in Figure 4.8, which contains the primary key of the document, the final score R_{td} and the query string t . The `RankedTextSearchResult` for the `current_document` will then be

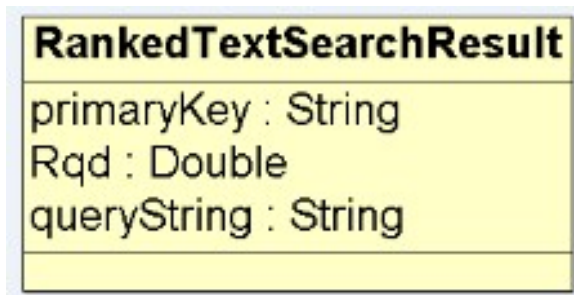


Figure 4.8: The data structure used in text search ranking process.

added to the list `Text_search_results`, which is a list of `RankedTextSearchResult`. After all the matched documents are inserted to `Text_search_results`, we sort this list by R_{td} and get the ranked list, which will be then returned to the process `Q1query(queryString)`. As shown in Figure 4.5, `Q1query` finally generates the returned list based on the ranked `Text_search_results`.

4.1.3.3 The Format for the Q1 Returned List

As shown in Figure 4.5, a Q1 query finally generates a returned list `returnList` based on the sorted `Text_search_results`. The `returnList` is a long string containing all the information in the `Text_search_results` that is useful for the user. Using `sepF` to stand for field separator and `sepN` to stand for node separator, the format for the `returnList` is shown as follows:

$$\text{Q1: } \underbrace{\text{prk1 sepF } R_{td1}}_{\text{Result1}} \text{ sepN } \underbrace{\text{prk2 sepF } R_{td2}}_{\text{Result2}} \text{ sepN } \dots$$

For the text search result, each node corresponds to a `RankedTextSearchResult` (see Figure 4.8). The `prk` and `Rtd` values indicate the values of the attributes `primaryKey` and `Rtd` in the corresponding `RankedTextSearchResult`, respectively. In the program, we use “@@@” as node separator and “;;;” as field separator. We do not use spaces, newline characters or normal punctuation as separators because they might be part of the search results.

4.2 Spatial Index Using Packed R* Tree

After adding locations to the records as explained in chapter 3, we will have two different types of records: records that have (ϕ, λ) locations describing them, and records that have polygons related to them. For the records having specific locations, we first pack B points in the smallest bounding box that encloses them, where B is the maximum number of data points contained in one leaf node. Then we insert the generated bounding boxes with associated B point data as leaf nodes into an R* tree. For the records having polygon descriptions (e.g. York County), we directly get the bounding box of the polygon, and insert it together with the polygon data as a leaf node of the same R* tree.

4.2.1 R* Tree Leaf Nodes

Figure 4.9 shows the data structures used in R* tree construction. We use struct

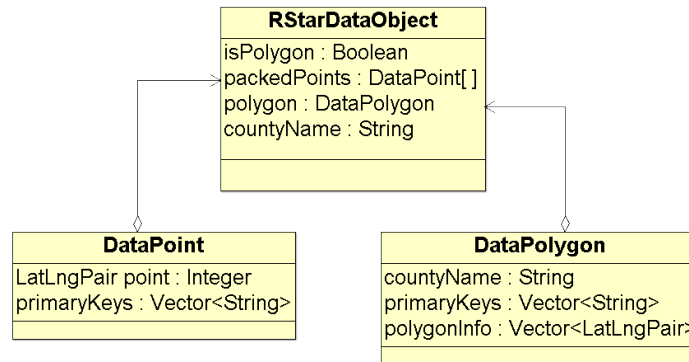


Figure 4.9: Data structures used in R* tree construction.

`DataPoint` to stand for point data in the records, each with a `LatLngPair` and a list of primary keys of documents associated with it. The `LatLngPair` stores the (ϕ, λ) location of the point, the data type of which is a pair of double values. Struct `DataPolygon` represents polygon objects, which contains a string of county name, a list of primary keys of related documents and spatial data for corresponding polygons. The spatial data for polygons is represented using a vector of `LatLngPairs` in the program.

We use the data structure `RStarDataObject` to represent the data object stored on the R* leaf nodes. The Boolean flag `isPolygon` is used to differentiate leaf nodes storing points from those storing polygons. If `isPolygon` is true, the data object contains the

name of the polygon and the associated spatial information for that polygon; otherwise, it contains a list of B packed points.

4.2.2 Constructing an R* Tree

The library we use for constructing an R* tree is from [46]. Figure 4.10 shows the sequence diagram for the R* tree constructor. Before building an R* tree, there are several data

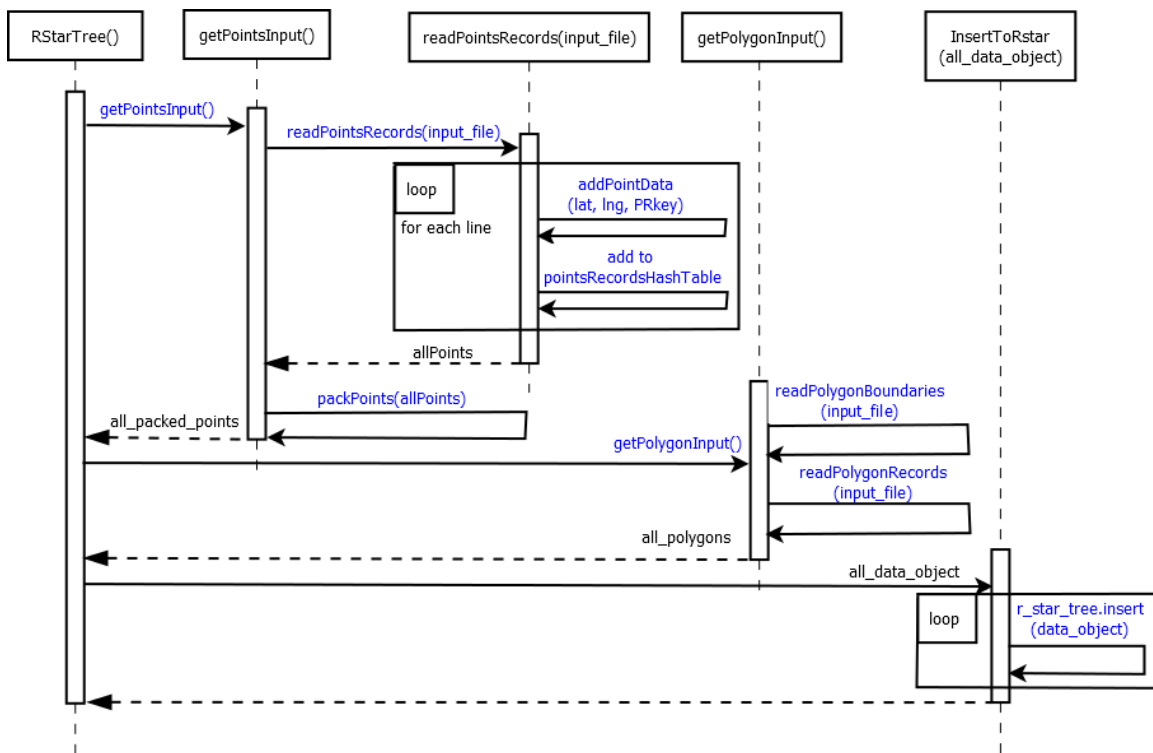


Figure 4.10: The sequence diagram for constructing an R* tree.

structures created for later use, they are:

- 1 **allDataObjects**: a vector of `RStarDataObject`, which will be used to store all the R* data objects.
- 2 **allPoints**: a hashtable from `LatLngPairs` to `DataPoint` objects.

- 3 `pointsRecordsHashTable`: a hashtable from records' primary keys to their corresponding `LatLngPairs`. The key value only exists when the record has an associated point data (ϕ, λ) pair. This hashtable will be used in a Q2 type query for checking if a record has point information in it.
- 4 `nameOfCounties`: an array of strings containing the names of the 15 counties in New Brunswick, Canada.
- 5 `polygonsData`: an array of `DataPolygons`, which contains the polygon defining the county boundaries for the 15 counties in New Brunswick, Canada, and the primary keys of their associated records.

To build an R^* tree, the constructor first invokes the process `getPointsInput`, which will in turn invoke another process `readPointsRecords(input_file)`. The method `readPointsRecords(input_file)` obtains records line by line from input files that contain records having associated (ϕ, λ) pairs giving their locations. For each line, the program retrieves the values of (ϕ, λ) and primary key, which are used as input parameters of the method `addPointData(lat, lng, PRKey)`. The method `addPointData(lat, lng, PRKey)` first checks if a point with this `LatLngPair` (ϕ, λ) already exists in `allPoints`. If the current `LatLngPair` already exists, the program will get the `DataPoint` object pointed to by the `LatLngPair`, and add `PRKey` to the primary key list stored in the corresponding `DataPoint`. Otherwise, the current `LatLngPair` appears for the first time in the data, so we create a new `DataPoint` object containing only one primary key, and add this `LatLngPair`, `DataPoint` pair to the `allPoints` hashtable. In each loop of `readPointsRecords(input_file)`, the program will also add the primary key, `LatLngPair` pair to the `pointsRecordsHashTable` for later use. At the end of `readPointsRecords`, the hashtable `allPoints` is returned to `getPointsInput`.

The process `getPointsInput` then invokes the method `packPoints(allPoints)`, which will pack every nearest B `DataPoint` objects into one `RStarDataObject`. The `isPolygon` attributes of the generated `RStarDataObjects` are set to `false`, indicating that these `RStarDataObjects` store point records. For each `RStarDataObject`, the nearest B points are chosen as follows:

- 1 Sort `allPoints` by their latitude.
- 2 Pack the first point a_1 in the sorted list `allPoints` to a `RStarDataObject` object, then remove point a_1 from `allPoints`.
- 3 Sort `allPoints` by their distances to a_1 .
- 4 Pack the first $B - 1$ points in `allPoints` to the same `RStarDataObject`, then remove these points from `allPoints`.

The process will loop until the list `allPoints` is empty. The generated `RStarDataObject` objects are returned to the R^* tree constructor in the list `all_packed_points`.

After inserting the points records, the R^* tree constructor invokes the process `getPolygonInput`. The `getPolygonInput` first calls the method `readPolygonBoundaries(input_files)`, which will read the polygon boundary information consisting of (ϕ, λ) pairs for the 15 counties in New Brunswick, Canada. The input file for the `readPolygonBoundaries` contains the (ϕ, λ) pairs of the simplified polygon boundaries as described in section 3.2.3.3. The obtained county boundaries are stored in the array `polygonsData` as vectors of `LatLngPairs`, along with their corresponding county names. The `polygonsData` array has 15 elements, each corresponding to a county in New Brunswick, Canada.

The `getPolygonInput` method invokes the method `readPolygonRecords` next, to get the records line by line from input files that contain records having associated polygons. For each record, the program retrieves its “County name” field, and checks if it matches the county name of any entry in the `polygonsData` array. If there is a matched county name, the program adds the primary key of this record to the primary key list of the matching `DataPolygon` object in the `polygonsData` array. After inputting all the records having polygons related, the 15 `DataPolygon` objects in `polygonsData` will have the primary keys of all the records associated to their corresponding county stored in their primary key lists. Finally, 15 `RStarDataObject` objects are created, with their `isPolygon` attributes set to true. The objects in `polygonsData` are assigned to `RStarDataObject` objects as their attributes `polygon`. The generated `RStarDataObject` objects are returned to the R^* tree constructor in the list `all_polygons`.

Now we have all the records along with their spatial information encapsulated as `RStarDataObject` objects. The R^* tree constructor combines the two lists `all_packed_points` and `all_polygons` as a new list `all_data_objects`, and uses it as the input of the method `InsertToRstar`. For each `RStarDataObject` in `all_data_objects`, `InsertToRstar` gets the bounding box of this `RStarDataObject`, and calls the process `r_star_tree.insert` to insert all the `RStarDataObject` to the R^* tree.

4.2.3 Querying an R^* Tree

Figure 4.11 shows the sequence diagram for a Q3 query $Q(p, r)$, where p is our interest point and r is the radius. A Q3 query returns the ranked documents with their locations falling within the circle of radius r centered at position p . To perform a Q3 query, the `Q3Query(p, r)` first invokes `Q3InternalSearch(p, r, L1, L2)`, which returns two lists L_1 and L_2 of ranked results, with L_1 for point results, and L_2 for polygon results. The

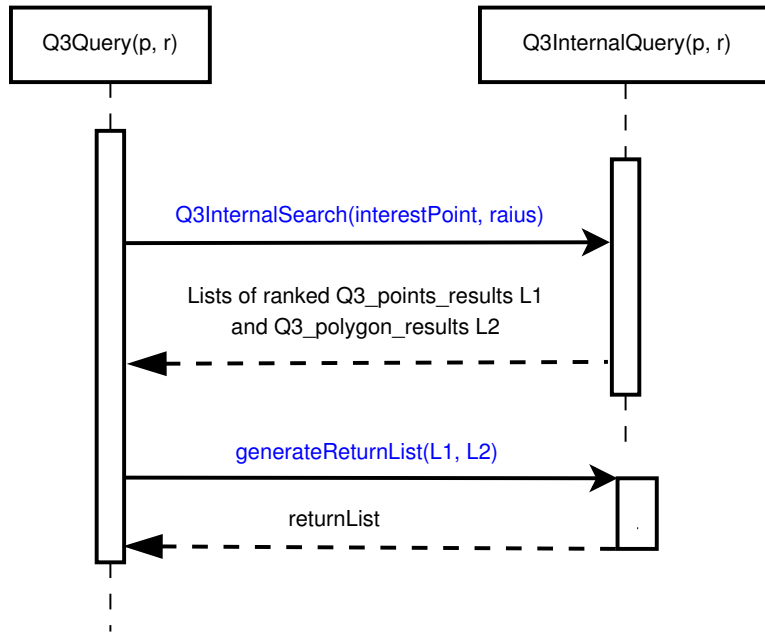


Figure 4.11: The sequence diagram for a Q3 query.

`Q3Query` generates the returned list `returnList` based on L_1 and L_2 , which is a long string containing all the useful information in L_1 and L_2 . The method `Q3InternalSearch(p , r , L_1 , L_2)` is shown in Algorithm 4.1.

As we can see from Algorithm 4.1, the lists L_1 and L_2 are initially empty. The bounding square bb defined by the circle $Q(p, r)$ is first computed at line 2. The `r_star_tree.query(bb)` method is then called, which returns two sets of query results: U for point results and V for polygon results intersecting bb . Each element of U and V is an `RStarDataObject` (see Figure 4.9). In the program, we use the visitor design pattern to perform a range query in the R^* tree, and keep two global lists U and V to maintain the results.

If a Q3 query has point results returned (i.e. $|U| \geq 1$), the program goes into the for loop at line 4. For each point record in U , the program first computes the geographic distance

Algorithm 4.1: Q3InternalSearch (p, r, L_1, L_2)

Input:Interest point p , radius r defining circle $Q(p, r)$;**Output:**List of points L_1 and polygons L_2 intersecting $Q(p, r)$;

```
1  $L_1$  and  $L_2$  are empty lists ;
2  $bb \leftarrow$  square encompassing circle  $Q(p, r)$  ;
3  $U, V \leftarrow$  r_star_tree.query( $bb$ ) ;
4 for  $int\ i \leftarrow 0$  to  $|U| - 1$  do
5    $dist \leftarrow$  ApproxDistance( $U_i, p$ ) ; // geographic distance
6   if  $dist \leq r$  then
7      $L_1 \leftarrow L_1 + U_i$  ;
8 Sort  $L_1$  on  $dist$  ;
9 for  $int\ j \leftarrow 0$  to  $|V| - 1$  do
10  if  $p \in V_j$  then
11    // if  $p$  is inside  $V_j$ 
12     $L_2 = L_2 + V_j$  ;
13  else
14    // if  $p$  falls outside or on the boundary of  $V_j$ 
15     $dist \leftarrow$  minDist( $V_j, p$ ) ; // from  $p$  to polygon boundary
16    if  $dist \leq r$  then
17       $L_2 = L_2 + V_j$  ;
16 Sort  $L_2$  on  $dist$  ;
17 return  $L_1, L_2$  ;
```

$dist$ (on the earth's surface) from p to the current point U_i . If $dist \leq r$, the current point U_i is added to the list L_1 . After checking all the points $U_i \in S$, L_1 is sorted on $dist$. If the set V for polygon results is not empty, we enter the for loop from lines 9 to 15. For all the polygon results $V_j \in V$, the program first checks if p is inside the polygon V_j . If p is falling within V_j , we set the distance $dist$ to 0 and add the record to the list L_2 . Otherwise, the program computes the minimum distance $dist$ from p to the polygon boundary of V_j . If $dist \leq r$, the current record V_j is then added to the list L_2 . The generated list L_2 is sorted on $dist$ after adding all polygons in V that intersect $Q(q, r)$. Finally,

the lists L_1 and L_2 are returned to the process `Q3Query` by the method `Q3InternalSearch`.

The ranked results in the lists L_1 and L_2 are maintained by the struct `RankedSpatialSearchResult`, which is shown in Figure 4.12. The attributes `score`, `distance` and `primaryKeys` are valid for both point and polygon results. Vector `primaryKeys` stores the primary keys for records having the position indicated by `currentPoint` (for L_1) or county with `countyName` (for L_2). For a point result $\ell \in L_1$, attribute `distance` is the distance from point p to ℓ 's (ϕ, λ) pair `currentPoint`. For polygon result $\ell \in L_2$, `distance` is the distance from p to the nearest edge of the polygon ℓ 's boundary. The boolean attribute `isPolygon` indicates the result type for the `RankedSpatialSearchResult`. Only one of `LatLngPair` or `countyName` is valid for each `RankedSpatialSearchResult`.



Figure 4.12: The data structure of the `RankedSpatialSearchResult`. Both L_1 and L_2 are stored in the structure. For L_1 results, `isPolygon` = “false” and `countyName` = \emptyset . For L_2 results, `currentPoint` = \emptyset .

4.2.3.1 The Format for the Q3 Returned List

As shown in Figure 4.11, The `Q3Query(p, r)` finally generates a returned list `returnList` from the point results list L_1 and polygon results list L_2 . The `returnList` is a long string which contains all the information for a valid Q2 search result. The format for the

`returnList` is shown as follows:

```
Q2:PointResultsHeader###PointResultsContentents%%PolygonResultsHeader###PolygonReusltsContentents
```

As we can see, the query type “Q2” is separated by the “:” with the search result contents. There are two parts in the search result contents: point search results consisting of `PointResultsHeader` and `PointResultsContentents`, and polygon search results consisting of `PolygonResultsHeader` and `PolygonResultsContentents`. The corresponding results header and results contents are split by the separator ”###”.

Assume that we use `sepN`, `sepF` and `sepP` to represent node separator, field separator and primary key separator, respectively. The format for the `PointReusltsCoutents` is shown as follows:

```
lat1 sepF lng1 sepF dist1 speF prk1 sepP prk2 sepP ... sepN
                                primary keys
┌───────────────────────────────────────────────────────────────────────────────────┐
Node 1
lat2 sepF lng2 sepF dist2 speF prk1 sepP prk2 sepP ... sepN
                                primary keys
┌───────────────────────────────────────────────────────────────────────────────────┐
Node 2
:
:
```

For the point search results, each node corresponds to a `RankedSpatialSearchResult` (see Figure 4.12) having the `isPolygon` attribute set to false. The `lat`, `lng` and `dist` fields indicate the values of the attributes `currentPoint` and `distance`. All the primary keys in the vector `primaryKeys` are encoded as one single field, in which every primary key is separated using `sepP`. Similarly, the `PolygonResultsContentents` are formatted as follows:

```
countyName1 sepF dist1 speF prk1 sepP prk2 sepP ... sepN
                                primary keys
┌───────────────────────────────────────────────────────────────────────────────────┐
Node 1
countyName2 sepF dist2 speF prk1 sepP prk2 sepP ... sepN
                                primary keys
┌───────────────────────────────────────────────────────────────────────────────────┐
Node 2
:
:
```

4.3 Combined Text and Spatial Query Q2

As explained in Section 1.3, a Q2 query $Q2(t, r)$ returns the ranked list of records having their locations intersecting a circular disk of radius r centered at the locations of the records matching search string t . To perform a $Q2(t, r)$ query, a text query $Q1(t)$ needs to be performed first, which will return a set \mathcal{P} of search results. We then perform a set of Q3 point + radius queries for points $\mathcal{P}_i \in \mathcal{P}$, which will return all points and polygons falling in range.

4.3.1 Data Structures for Maintaining Q2 Search Results

For the Q3 point results and Q3 polygon results, we have two types of results for a Q2 query, which are Q2 point results and Q2 polygon results. As an example of Q2 point results, assume we have 4 points in a database as shown in Figure 4.13. In the example, we search for $Q2(t, r) = Q2(\text{“Mc”}, 0.5\text{km})$. We first perform text search for $Q1(\text{“Mc”})$, which returns two point results matching the text “Mc” in the list \mathcal{P} , point A with the text “McDonald’s Restaurant” and point C with the text “McConnell Hall”. The Q2 query then searches for the points intersecting the disk of radius 0.5km centered at points A and C, which then finds points B and D in range. In Figure 4.13 and Figure 4.14, we use L_1 to represent Q3 point results and L_2 to represent Q3 polygon results. In Figure 4.13, $\mathcal{P}_1.L_1$ is the Q3 point results list centered at the first element in \mathcal{P} , and $\mathcal{P}_2.L_1$ is the Q3 point results list centered at the second element in \mathcal{P} .

Assume we have 4 points A, B, C, D and 1 polygon E in a database, an example of Q2 polygon results is shown in Figure 4.14. In the example, we search for the text $t = \text{“Mc”}$

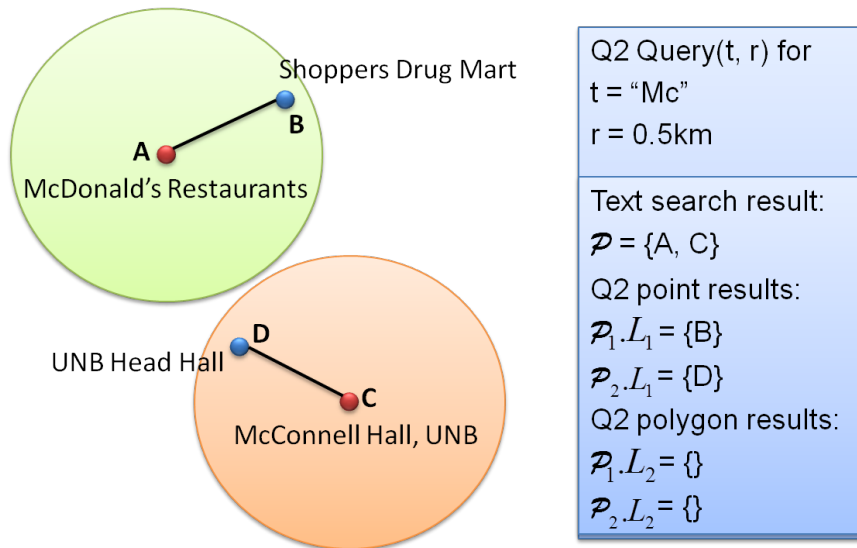


Figure 4.13: An example of a Q2 point results.

and radius $r = 0.3km$, which first returns points A and C as the text search results. We then perform a search for the polygon intersecting the disk of radius 0.3km centered at A and C. The Carleton county E is finally returned as the polygon result found in range. In Figure 4.14, $\mathcal{P}_1.L_2$ is the Q3 polygon results list centered at the first element in \mathcal{P} , and $\mathcal{P}_2.L_2$ is the Q3 polygon results list centered at the second element in \mathcal{P} .

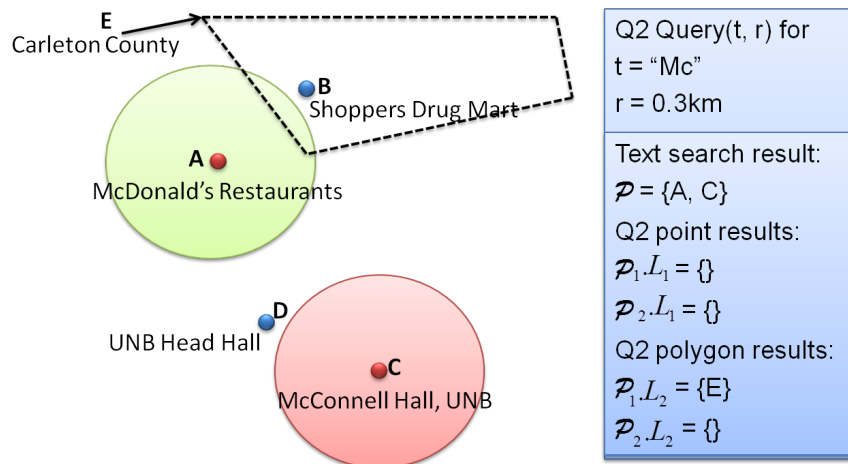


Figure 4.14: An example of a Q2 polygon results.

In Figure 4.14, assume that the distances from A to B and from D to C are both less than 0.6km. If we perform a Q2 query $Q2(\text{“Mc”}, 0.6)$, we will get points B and D returned as Q2 point results $\mathcal{P}_1.L_1 = B$, $\mathcal{P}_2.L_1 = D$, and polygon E returned as a Q2 polygon result $\mathcal{P}_1.L_2 = E$. The Q2 point results and Q2 polygon results are maintained by the data structures `TexSpaPointResult` and `TexSpaPolygonResult`, respectively, as shown in Figure 4.15.

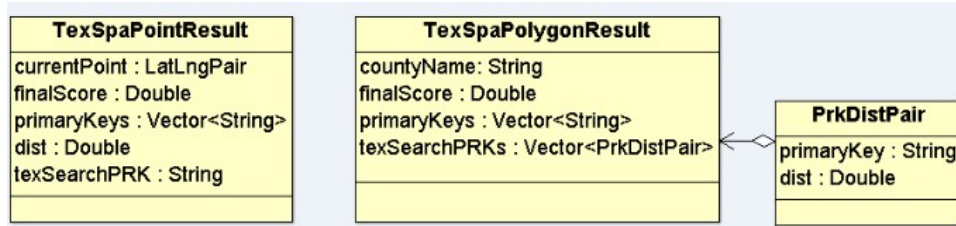


Figure 4.15: The data structures of the `TexSpaPointResult` and `TexSpaPolygonResult`. These are denoted as r_1 and r_2 in Alg. 4.2.

In `TexSpaPointResult`, `currentPoint` is the `LatLngPair` that defines the location of records with this (ϕ, λ) matching the Q2 query. The attribute `finalScore` is the combined text + spatial score for the result, as defined in equation 4.11. The list of primary keys `primaryKeys` stores all records with this (ϕ, λ) pair within radius r of query point p has the primary key `texSearchPRK`. Query point p is not recorded explicitly. For more than one centering disk intersecting the (ϕ, λ) pair, the attribute `texSearchPRK` is the primary key of the *nearest* text search result that has a circular disk of radius r centered at its location covering the `currentPoint`. The `dist` indicates the distance from the `currentPoint` to the point associated with the record having `texSearchPRK`.

A `TexSpaPolygonResult` uses the `countyName` to identify its location. The `TexSpaPolygonResult` also contains a `finalScore` and a list of primary keys of its associated records. For a `TexSpaPolygonResult`, the vector `texSearchPRKs` of struct

`PrkDistPair` contains all the primary keys of the text search results having the circular disks of radius r centered at their locations intersecting the county boundary, along with the nearest distances from their locations to the county boundary. The `PrkDistPair` is a struct containing 2 attributes: a string `primarykey` and a distance `dist`. For a Q2 polygon result, a distance `dist` in the `PrkDistPair` of 0 means the location of the text search result is inside the polygon or on the polygon boundary.

Figure 4.16 shows an example in which there is more than one disk arising from text search results of a Q2 search intersecting a polygon. Assume that points A and B are point records matching the text from a Q2 search. Both disks of radius r centered at A and B intersect the polygon `Charlotte County`. The generated `TexSpaPolygonResult` P is shown in Figure 4.16, which contains a `PrkDistPair` vector of length 2, for text search results A and B, respectively. The primary key list `prk_list_p` contains all the records without a (ϕ, λ) pair (i.e. having only this county name describing their location).

4.3.2 Combined Score for the Text + Spatial Search

For the text ranking score R_{td} from a text result in \mathcal{P} and distance `dist` obtained from the Q3 point + radius search, the combined score `finalScore` of `TexSpaSearch` is computed as follows:

$$\text{finalScore} = W_t * \frac{R_{td}}{\text{ScoreMax}} + W_s * \cos\left(\frac{\pi \text{dist}}{2r}\right) \quad (4.11)$$

where W_t is the weight for the text search and W_s is the weight for the spatial search, with the further restriction that $W_t \in [0, 1]$, $W_s \in [0, 1]$ and $W_t + W_s = 1$. By default, W_t and W_s are both set to 0.5. `ScoreMax` is the maximum text ranking score returned by the Q1 text search. Since $Q1(t)$ search returns a ranked list \mathcal{P} of text search results, we can obtain `ScoreMax` as $\text{ScoreMax} = \mathcal{P}_0.R(q, d)$ (see Figure 4.5), which means the

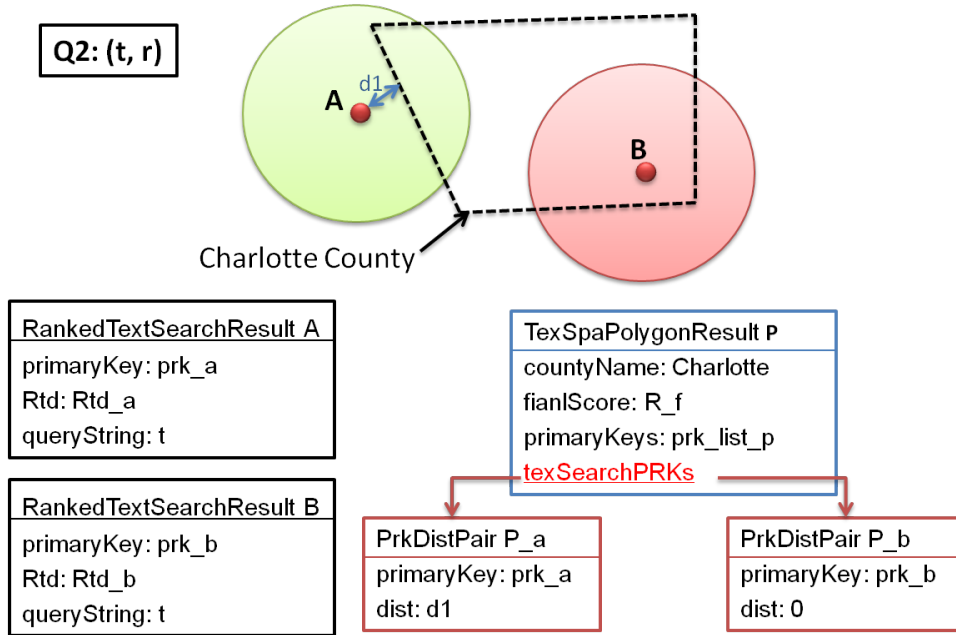


Figure 4.16: An example of the `TexSpaPolygonResult` when there is more than one disk arising from a `Q2` text query intersecting a polygon. `Rqd` is the text score for a document as defined by equation (4.10) and `finalScore` is defined by equation (4.11).

first result in the ranked list. For the text search score, we have $\frac{R_{td}}{ScoreMax} \in [0, 1]$. For the spatial search score, since only points and polygons having $dist \leq r$ are returned, we have $\cos(\frac{\pi}{2} \frac{dist}{r}) \in [0, 1]$, where $dist \in [0, r]$ is inversely proportional to the spatial score.

4.3.3 Algorithm for `Q2` query

Algorithm 4.2 shows the pseudo-code for a `Q2` query $Q(t, r)$, where t is the query string and r is the radius. The query string t is first transferred to an all lower case query string t_{lower} at line 4. Any punctuation is then removed to obtain a compact query string $t_{compact}$ at line 5. At line 6 of Algorithm 4.2, the `Q2` search first performs a `Q1InternalSearch` $Q(t)$ to return a ranked list \mathcal{P} of text search results. The `Q1` process is explained in Section 4.1.3. If the i_{th} record \mathcal{P}_i is a point record (i.e. not a polygon), then we perform a search for other point or polygon records intersecting the disk of radius

r centered at \mathcal{P}_i from lines 13 to 24.

For a `RankedTextSearchResult` (see Figure 4.8) \mathcal{P}_i , we first obtain the text ranking score R_{td} and corresponding primary key $textPrk$ at lines 9 and 10. We can then retrieve the `LatLngPair` p associated with the $textPRK$ in the R^* tree using the `pointsRecordsHashTable`, which is a hash table from records' primary keys to their corresponding `LatLngPairs`, as described in 4.2.2. List L_s is a list of `LatLngPairs`, which stores the points that have already been used as inputs of the `Q3InternalSearchs` in the entire Q2 query process. If p is not in L_s (p is first used as the input point in a Q3 point + radius search in the Q2 process), we perform a `Q3InternalQuery(p,r)`, getting the lists L_1 for Q3 point results and L_2 for Q3 polygon results.

R_1 and R_2 are empty lists for `TexSpaPointResult` and `TexSpaPolygonResult` initially. At lines 17 and 22, Algorithm 4.2 first computes the combined score using the text score $R(q,d)$ and `dist`. The corresponding `TexSpaPointResult` $r1$ and `TexSpaPolygonResult` $r2$ are then generated based on the text result \mathcal{P}_i , the Q3 internal search results $L_1[j]$ or $L_2[j]$, and the `finalScore` at lines 18 and 23, respectively. Algorithm 4.3 and 4.4 detail these two result generation processes. Figure 4.15 illustrates the structure used to hold $r1$ and $r2$. The generated $r1$ and $r2$ results are added to the list R_1 and R_2 , respectively. Finally, we sort R_1 and R_2 on the `finalScore` at lines 25 and 26, and generate a return list `returnList` based on the sorted R_1 and R_2 at line 27. The `returnList` is a long string containing all the information in R_1 and R_2 that is useful for the user. Figure 4.17 shows the flow chart for Algorithm 4.2.

Assume we have 4 points A, B, C, D and 1 polygon E in a database, Figure 4.18 shows an example for a $Q2(t, r)$ search with $t = \text{“Mc”}$ and $r = 0.6$ km. The Q2 query first

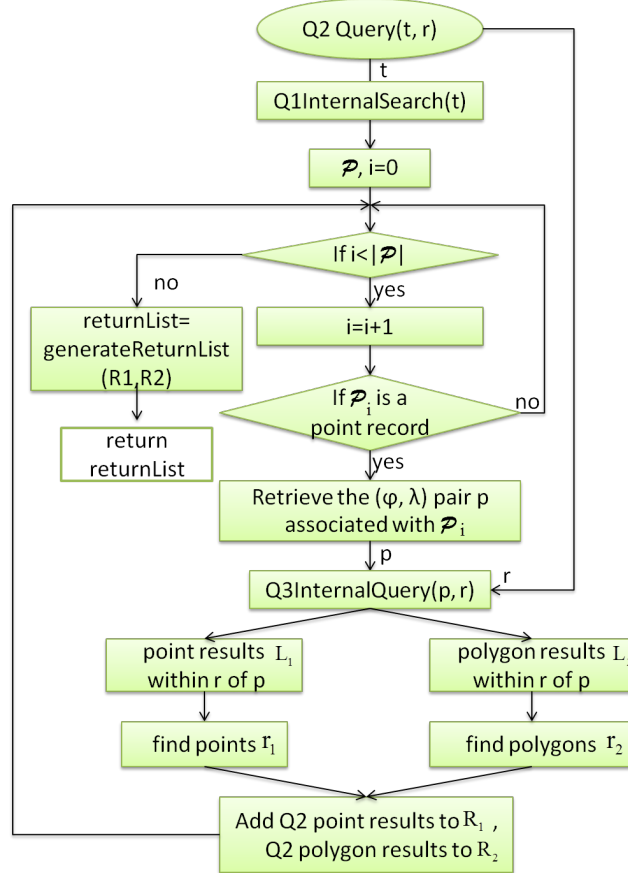


Figure 4.17: The flow chart for Algorithm 4.2.

invokes a `Q1InternalQuery`, which returns a list of `RankedTextSearchResults` containing results A and C. We then perform a search for the points and polygons intersecting the disk of radius 0.6 km centered at A and C. The `TexSpaPointResults` R_b , R_d and the `TexSpaPolygonResult` R_e are finally returned as the Q2 search results. The primary key vectors `prk_list_b`, `prk_list_d` and `prk_list_e` store the primary keys of the records associated with the point B, D and the polygon E in the R^* tree, respectively. Primary key lists are necessary as point records can have identical (ϕ, λ) coordinates, and some records have only a county name indicating their location (see section 3.2.3.2).

The format and processing for generating the `returnList` for Q2 query is similar to that

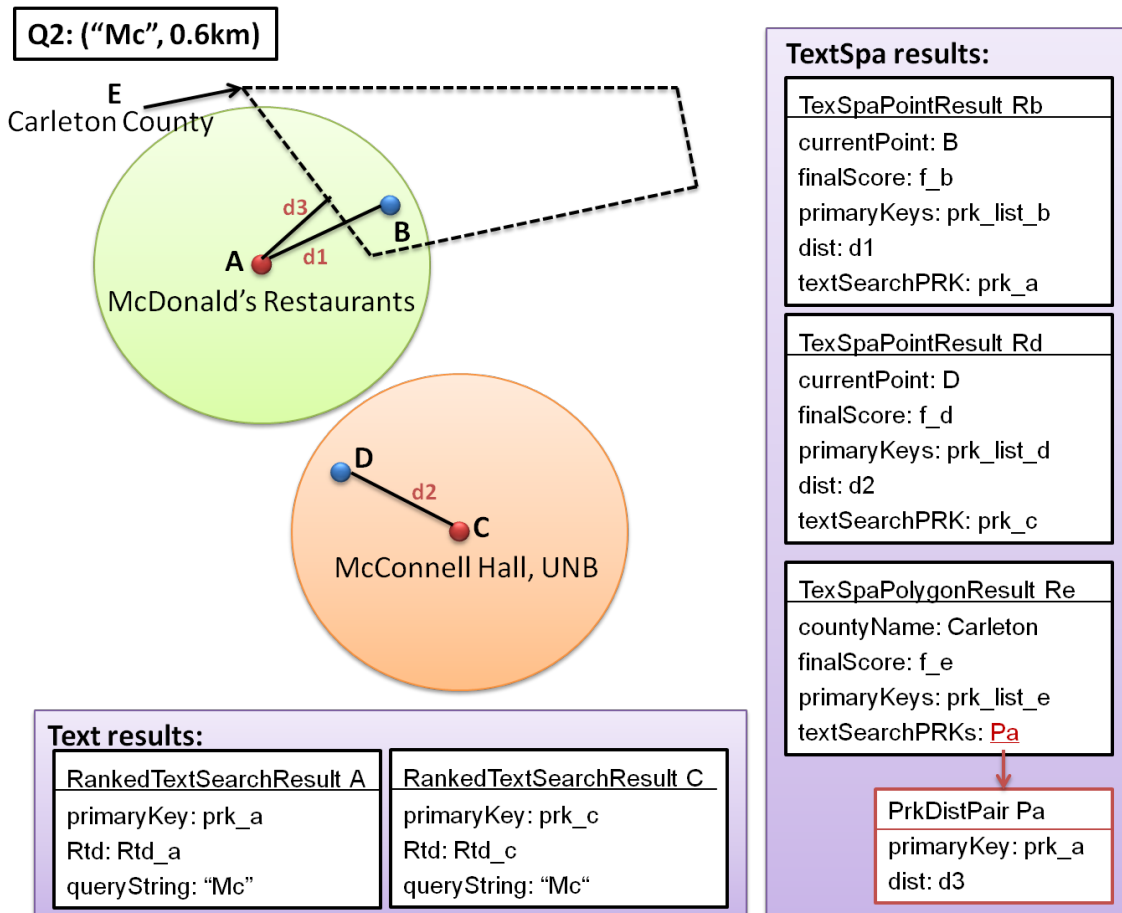


Figure 4.18: An example of a $Q2(t, r)$ search with $t = \text{"Mc"}$ and $r = 0.6$ km.

shown in Section 4.2.3.1 (for $Q3$, spatial search) and 4.1.3.3 (for $Q1$, text search).

4.4 Web Server architecture

4.4.1 Server Side Architecture

We wrote a Java and C++ server called `TexSpaSearch` that provides a web user interface for our search engine. The web server we used is Apache Tomcat. `HTMLHandlerServlet` is the server side Java servlet that handles HTML requests. The server architecture is shown in Figure 4.19.

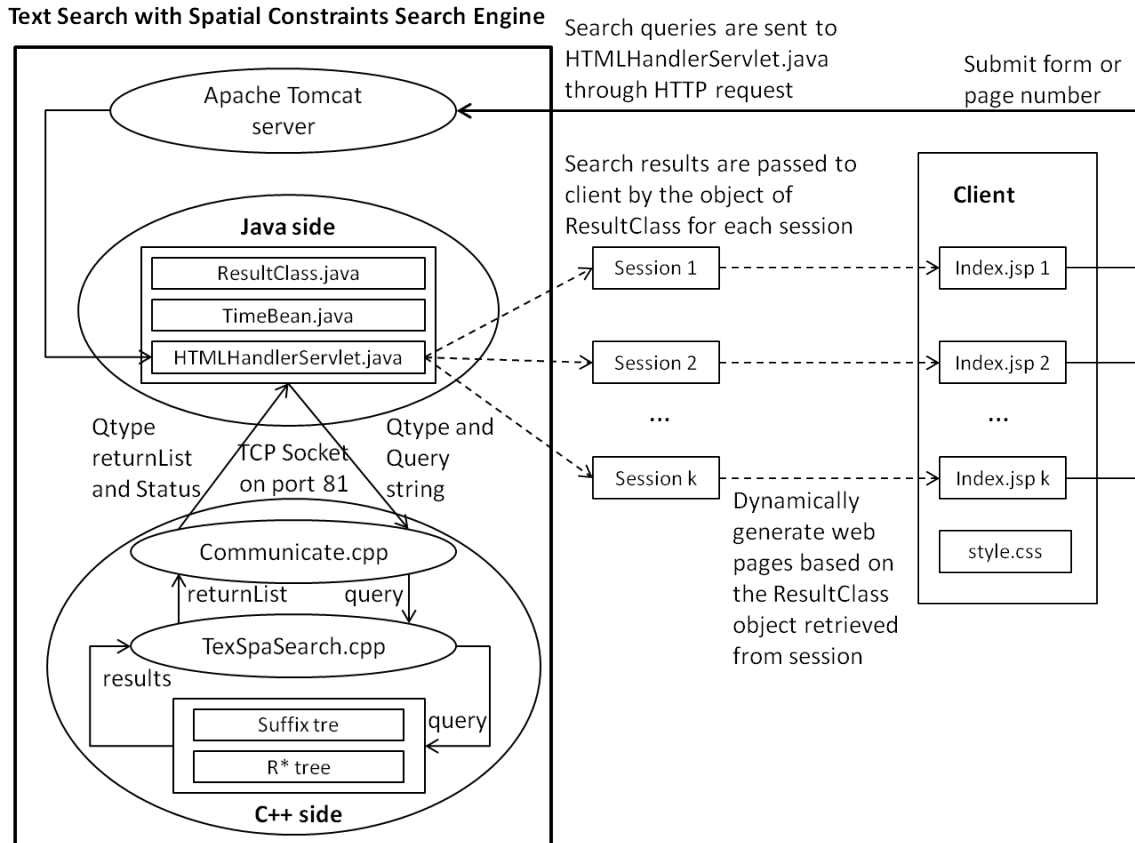


Figure 4.19: Architecture diagram of the TexSpaSearch web application server.

Each time after a servlet is instantiated, a method `init()` is called exactly once to indicate that the servlet is being placed into service [12]. We can change the servlet configuration by editing the `web.xml` file. On lines 6-9 of `web.xml`, we set the tag `load-on-startup` to be a non-zero value so that the `init()` method is called when the Servlet is started on the Tomcat web server. If the `load-on-startup` tag is set to zero, `init()` is called when the Servlet is created, which is appropriate on startup. Our preprocessed data is read into a Java HashMap in the `init()` method, which guarantees that the reading process is executed only once after the servlet is loaded (which means it does not execute every time the user sends a request). Part of the configuration file `web.xml` for the servlet is shown

in Figure 4.20.

```
6 <servlet>  
7   <servlet-name>HTMLHandlerServlet</servlet-name>  
8   <servlet-class>HTMLHandlerServlet</servlet-class>  
9   <load-on-startup>2</load-on-startup>
```

Figure 4.20: Lines 6-9 from `web.xml` file.

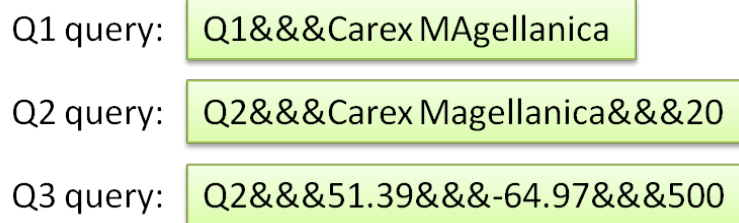
We use JSP (Java Server Pages) to implement the client page code. JSP allows Java code to be interleaved with static web markup content, so we can define a `ResultClass` to store the search results. It is very helpful to be able to associate some data with each client in a web server. For this purpose, a session can be used in JSP. A session is an object associated with a client. Data can be put in the session and retrieved from it, and operates like a hash table. In our servlet, an object of `ResultClass` called `finalResult` is used to store the search result for each session. The session can be obtained from the HTTP request. Each client has a session with our server, so each client has their own `finalResult` object. In this way the server can process queries for each client separately without causing critical section issues.

In the JSP file `index.jsp`, we have an input form named `inputForm`. The action of the form is `HTMLHandlerServlet`, with the submit button click sending HTTP POST requests to the servlet. When clients click on the search button, `index.jsp` will first check whether the user input is valid or not by calling `validateForm()` (longitudes and latitudes must be in range). If the input is valid, the user query strings are sent to `HTMLHandlerServlet`. The search queries are passed to `HTMLHandlerServlet` by the `doGet(request, response)` method. The method `doGet(request, response)` will be called when the Servlet receives an HTTP GET request.

When clients send requests via clicking the search button, `HTMLHandlerServlet.java` creates a TCP socket connection to the `TexSpaSearch` C++ program through port 81. The query strings and other parameters (e.g. `radius`, ϕ , λ) are maintained in the parameter `request`, which is of type `HttpServletRequest`. `Qtype` determines the query type. In `HTMLHandlerServlet`, four `String` variables `qString`, `radius`, `lat`, `lng` are defined to store user queries. The search request that is being sent out is `Qtype + query`. `HTMLHandlerServlet` checks the query type first. The format of the query differs for the three `Qtypes`, as follow:

1. If `Qtype` equals `Q1`, the query is formed with `qString`.
2. If `Qtype` equals `Q2`, the query is formed with `qString` and `radius`.
3. If `Qtype` equals `Q3`, the query is formed with `lat`, `lng` and `radius`.

The different fields in the search request are separated by the separator string “&&&”. Figure 4.21 shows three example queries sent to the TCP Socket. The query requests are



Q1 query: Q1&&&Carex MAgellanica

Q2 query: Q2&&&Carex Magellanica&&&20

Q3 query: Q2&&&51.39&&&-64.97&&&500

Figure 4.21: Three example queries sent to the TCP Socket.

sent to the `TexSpaSearch` C++ back end program via a socket connection through port 81.

The search results are returned by the `TexSpaSearch` program through the same socket via a string `recvMsg` through the TCP socket. The exact content of `RecvMsg` for each query type is explained in Section 4.2.3.1 and 4.1.3.3. We initialize `finalResult` after receiving a long string `returnList`. In the constructor of `ResultClass`, we separate the

long string into single results. For a Q1 search, each result corresponds to a record. For Q2 and Q3 search, each result corresponds to either a point or a polygon. All the separated results are stored in the arrays `q1FullResults` (for Q1 search) or `pointFullResults` and `polygonFullResults` (for Q2 and Q3 search). The size of the above arrays might be too large to return to the client at once, so we only return the results to be displayed on the current page. If the user sends the request by clicking the search button, the server returns the results of the first page. If the user sends the request by clicking a page number, the server returns the results of the requested page (see e.g. Figure 4.24).

A request is invoked by clicking the search button or a page number. Method `getResults()` is called to interpret the corresponding entries of the results arrays `q1FullResults`, `pointFullResults` and `polygonFullResults` based on the query type and the page number. The interpreted results are stored in vectors `q1CurrentResults` (for a Q1 search), or `pointCurrentResults` and `polygonCurrentResults` (for Q2 and Q3 search).

The `currentResults` vectors contain the human readable search results meta information such as the count of found records, URLs associated with the returned primary keys, and the contents of pages of the URLs. A sequence diagram showing how the server and client interact is shown in Figure 4.22.

4.4.2 Client Side Architecture

Before the server side returns any result, the welcome page `index.jsp` on the client side of the `HTMLHandler` is shown in Figure 4.23. Once a Q1, Q2 or Q3 search is performed by the user, the search results are displayed in the text field surrounded by the dashed line. On the server side, the servlet stores the search results in the session object. The client page

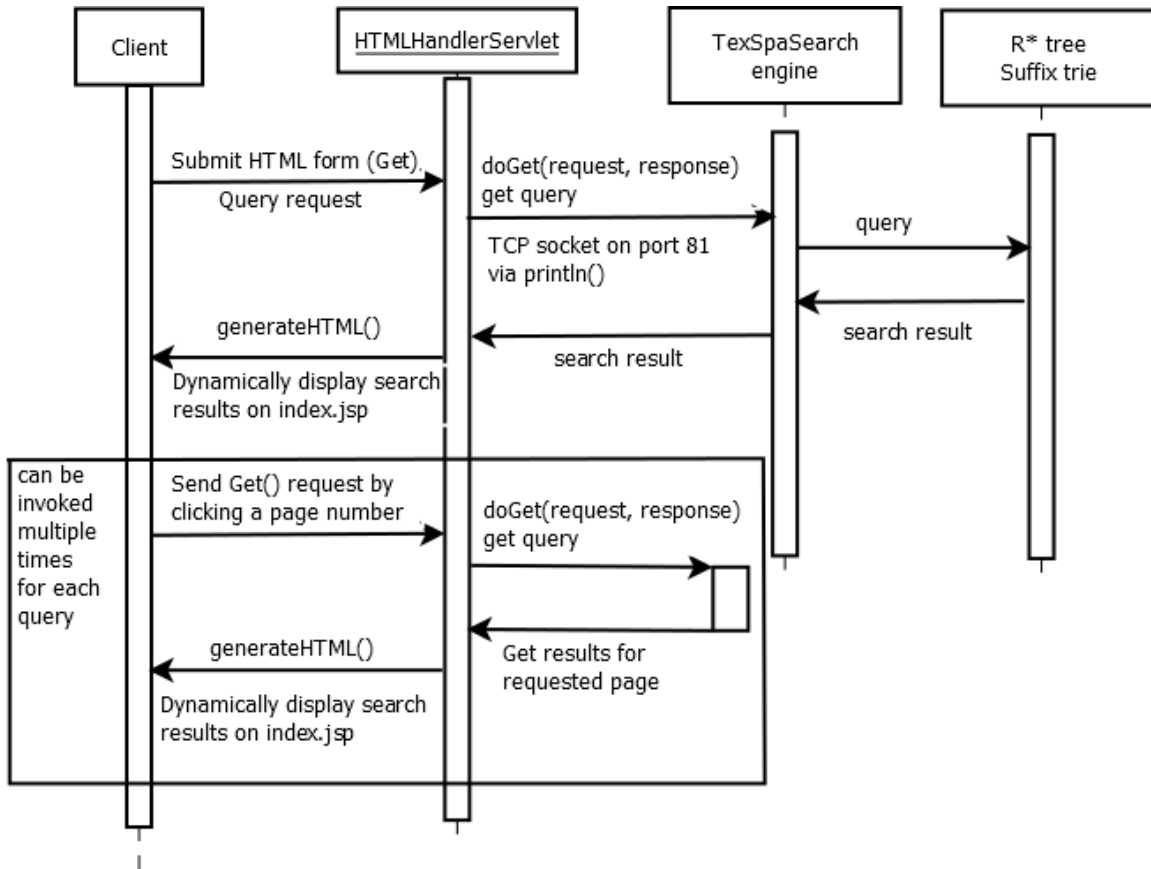


Figure 4.22: Sequence diagram of server and client interaction for the TexSpaSearch engine.

gets the result object `clientResult` from the session in the header of the `index.jsp` file. Method `updateMessage()` is the main method that controls the results displayed on the client side. In `updateMessage()` we get the query type and all the results to be displayed on the page corresponding to the query type through `clientResult`. There are three code blocks in the method `updateMessage()` on the client side, which are for Q1, Q2, and Q3 queries, respectively.

4.4.2.1 Display Q1 Results

If the query type is Q1, each `resultEntry` contains two parts: the URL link of the search result generated from the record's primary key, and the context of the corresponding record.

Text Search with Spatial Constraints Search Engine

Q1 Query String
 Q2 Query String Radius (m)
 Q3 Latitude Deg Min Seconds Longitude Deg Min Seconds Radius (m)

Figure 4.23: The welcome page of the HTMLHandler web application.

Method `upPageq1()` can display the search results in multiple pages. The number of search results displayed on each page is determined by variable `listNum`. An example of a Q1 search results display for the Q1 query string “Carex magellanica” is shown in Figure 4.24 with `listNum = 5`.

4.4.2.2 Display Q2 Results

For the Q2 point results, each element of `clientResult.pointCurrentResults` corresponds to a (ϕ, λ) pair. Each individual in `resultEntry` contains three fields: description of a returned point (a (ϕ, λ) pair), the URL of the nearest string search result, and all the URLs to the records having the same (ϕ, λ) pair associated, along with their context. The three fields are separated by symbol `&&&`. For each element in the `resultEntry`, we get

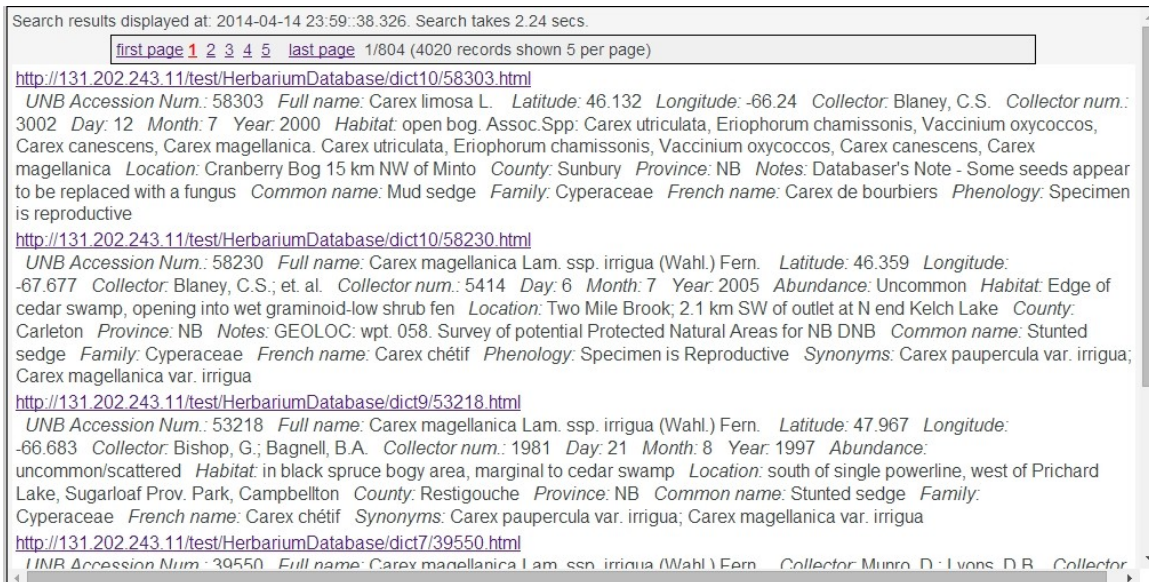


Figure 4.24: Q1 search result for string “Carex magellanica”, with listNum = 5.

a single field by splitting the individual element using separator `&&&`. Then, for the third field in the element, we can further obtain the single URL and context pairs of the records by splitting the field using separator `@@@`. At this point, the client side has all the readable information of Q2 point results. An example Q2 search showing point results as displayed on a web page is shown in Figure 4.26. The query used for Figure 4.26 is $Q2(t, r)$ where $t = \text{“crab apple”}$, $r = 5$ meters. In Figure 4.26, we labelled the corresponding attributes in `TexSpaPointResult` on the web page, the nearest string search result `texSearchPRK` of which is obtained from line 8 of Algorithm 4.2. For each point result in Q2 point results list R_1 , up to three records having exactly the same (ϕ, λ) are shown. If the number of records having exactly the same (ϕ, λ) is more than 3, we can click the link after the third result to display all the records, as shown in Figure 4.25.

Method `upPageq2point()` displays Q2 point results in multiple pages under the Q2 points tab. Figure 4.26 indicates 7 pages of Q2 point results.

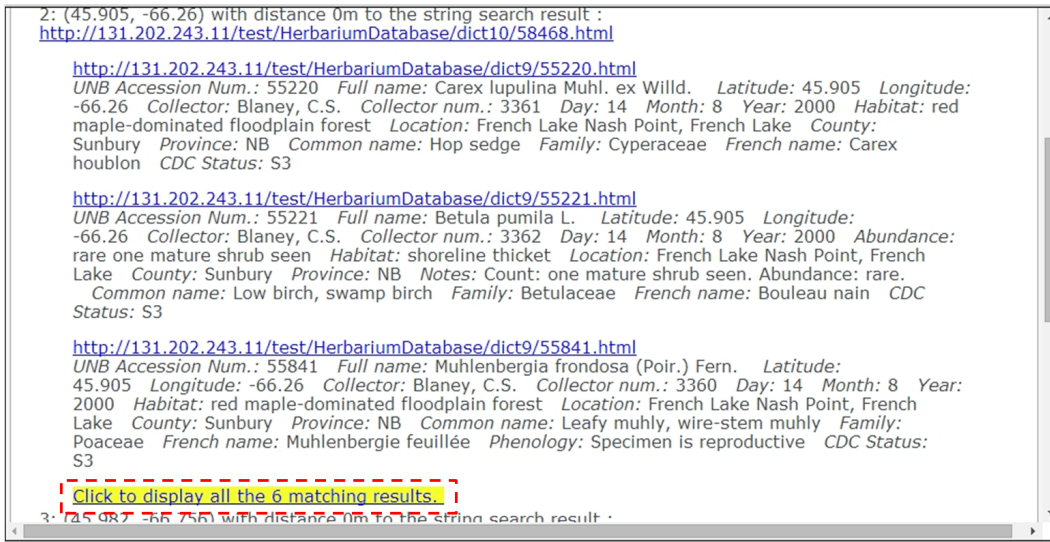


Figure 4.25: An example showing the link to display all 6 associated records for point 2: (45.905, -66.26) from a Q2 point result.

For the Q2 polygon results, all the individuals are stored in `resultclass.polygonCurrentResults`, each element of which corresponds to a county. Each `resultEntry` consists of three fields, as follows: (1) the description of a polygon result, i.e. the county name, (2) the URLs to the text search results intersecting the county along with their context, and (3) the URLs and their context of the records having no (ϕ, λ) pair but within this county. An example Q2 polygon result displayed on a webpage is shown in Figure 4.27. The contents of the third part of a Q2 polygon result are originally hidden. We can show the results associated with a county by clicking the link “Click to display all” on the bottom of the corresponding county result, as shown in Figure 4.28. Method `upPageq2polygon()` displays Q2 polygon results in multiple pages under the Q2 polygon tab.

4.4.2.3 Display Q3 Results

If the query type is Q3, we get the point and polygon results from `resultclass.pointCurrentResults` and `resultclass.polygonCurrentResults`. The

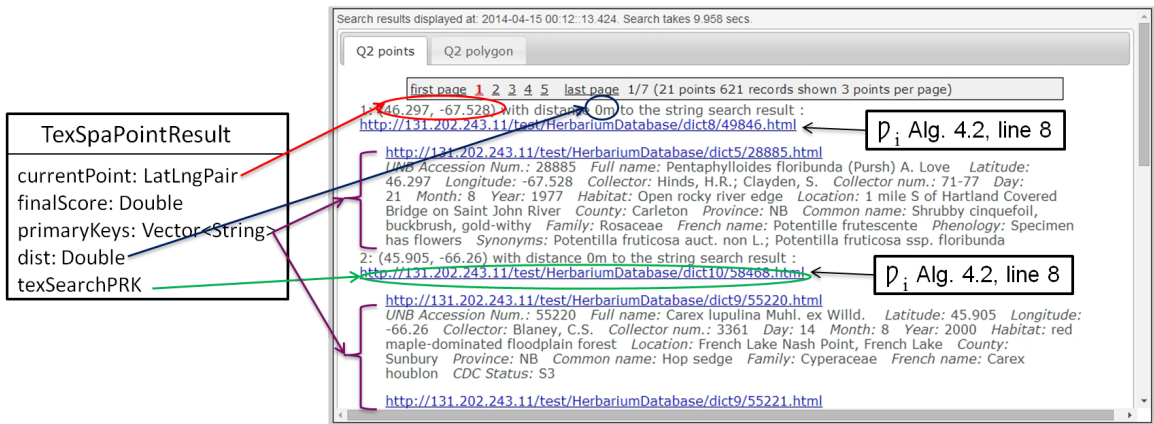


Figure 4.26: A sample of a Q2 point result showing two distinct points with distance 0 m to the Q2 string search results having primary keys 49846 and 58468, respectively.

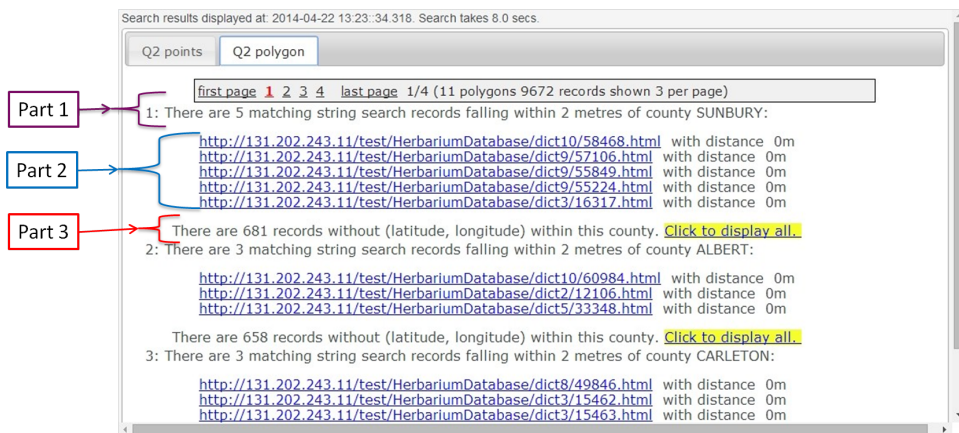


Figure 4.27: An example of Q2 polygon results for query string “Carex magellanica” and radius 2m displayed on the web page.

display panel for Q3 search results are also divided into two parts using two tabs. Method `upPageq3point()` displays Q3 point results in multiple pages under the Q3 points tab, and method `upPageq3polygon()` displays Q3 polygon results in multiple pages under the Q3 polygontab. Examples of Q3 point and Q3 polygon results are shown in Figures 4.29 and 4.30, respectively.

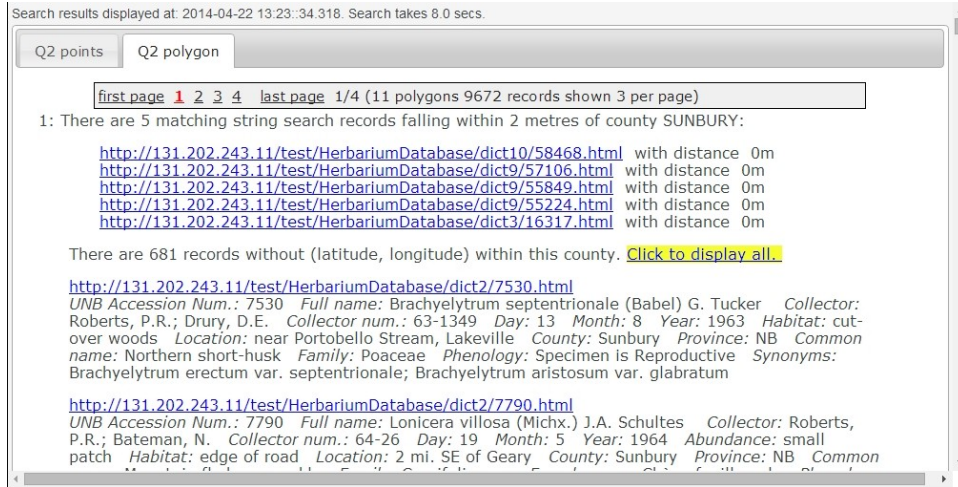


Figure 4.28: An example of linking to display all the associated records for a Q2 polygon result, in county Sunbury.

4.5 Source Code Summary

All the programs with their names, functionalities and number of lines are shown in Table 4.1. The total lines of code written is 8571.

4.6 Search Complexity

This section analyses the asymptotic search complexity of queries using the `TexSpaSearch` data structure. For a Q1 query $Q1(t)$, where t is the query string of length τ , we perform a suffix tree look up for each possible subphrase b of t . We represent the length of a subphrase b by $|b|$, so the algorithm searches for b in $O(|b|)$ time using the suffix tree as described in Section 1.1. For the query string t containing A single words, the time complexity $\mathcal{C}1$ can be computed as follows:

$$\mathcal{C}1 = \sum_{i=1}^{\frac{A(A+1)}{2}} O(|b_i|) \quad (4.12)$$

Since the longest subphrase of t has the length τ , so the upper bound of $\mathcal{C}1$ is $O(\frac{A(A+1)}{2}\tau)$.

Assume the average length of subphrases is $\overline{|b|}$, the average time complexity for a Q1 query

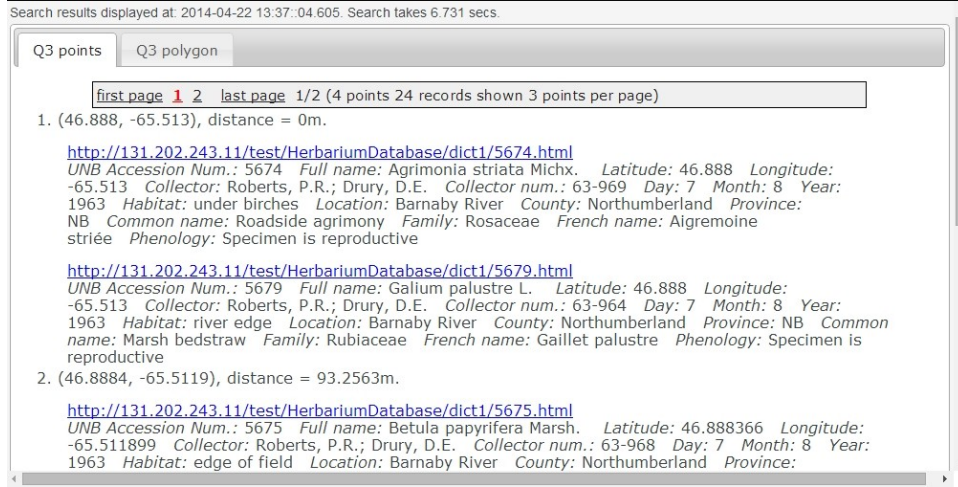


Figure 4.29: An example of Q3 point results for point (46.888, -65.513) and radius 5000m displayed on a web page.

$\overline{\mathcal{C}1}$ can be computed as:

$$\overline{\mathcal{C}1} = O\left(\frac{A(A+1)}{2}|\overline{b}|\right) = O(A^2|\overline{b}|) \quad (4.13)$$

For a Q3(p, r) query, assume there are \mathcal{D}_n data objects indexed in the R^* tree for n records, and the maximum number of entries of an internal node is \mathcal{M} , the query complexity $\mathcal{C}3$ for the Q3 search is $\mathcal{C}3 = O(\log_{\mathcal{M}} \mathcal{D}_n + y)$, where y is the number of leaf nodes found in range. In the worst case when all the bounding boxes in the R^* tree overlap the bounding square defined by p and r , the worst case query complexity is $O(\mathcal{D}_n)$. In the application, we use $\mathcal{M} = 8$.

For a Q2(t, r) query, the search engine first performs a Q1(t) query returning a list \mathcal{P} of text search results, which takes $\mathcal{C}1 = O(A^2|\overline{b}|)$ time. We then perform a Q3(p, r) query for each point record in \mathcal{P} . Assuming there are Z point records in \mathcal{P} , the time complexity $\mathcal{C}2$ for Q2 query can be computed as:

$$\mathcal{C}2 = O\left(\sum_{i=1}^{\frac{A(A+1)}{2}} |b_i|\right) + Z \log_{\mathcal{M}} \mathcal{D}_n + y = O(A^2|\overline{b}| + Z \log_{\mathcal{M}} \mathcal{D}_n + y) \quad (4.14)$$

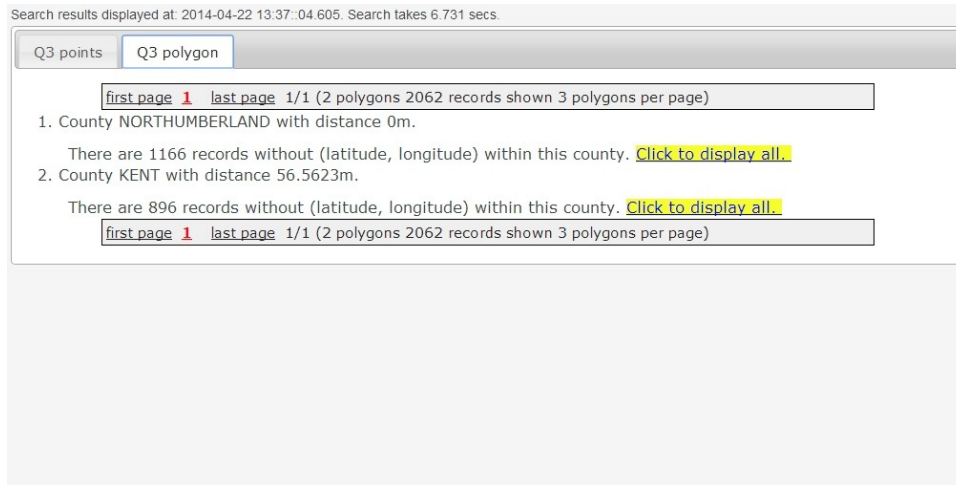


Figure 4.30: An example of Q3 polygon results for point (46.888, -65.513) and radius 5000m displayed on the web page.

on average. As we can see, the Q2 query time heavily depends on the number of point results returned by the list \mathcal{P} .

Algorithm 4.2: Q2Query ($t, r, \text{returnList}$)

Input:

Query string t , radius r ;

Output:

A long string `returnList` contains all the information of records intersecting a disk of radius r centered at points matching string t ;

```
1  $R_1$  is an empty list of TexSpaPointResult ;
2  $R_2$  is an empty list of TexSpaPolygonResult ;
3  $L_1, L_2, L_s \leftarrow$  empty lists ;
4  $t\_lower \leftarrow$  allToLower( $t$ ) ;
5  $t\_compact \leftarrow$  removePunct( $t\_lower$ ) ;
6  $\mathcal{P} \leftarrow$  Q1InternalSearch( $t\_compact$ ) ;
7 for  $int\ i \leftarrow 0$  to  $|\mathcal{P}| - 1$  do
8   if  $\mathcal{P}_i$  is a point record then
9     // not a polygon
10     $R_{td} \leftarrow \mathcal{P}_i.R_{td}$  // text score eq. (4.10);
11     $textPrk \leftarrow \mathcal{P}_i.primaryKey$  ;
12     $p \leftarrow r.star.tree.pointsRecordsHashTable[textPrk]$  //  $p = (\phi, \lambda)$  of  $\mathcal{P}_i$ ;
13    if  $p \notin L_s$  then
14      //  $p$  not already reported
15       $L_s \leftarrow L_s + p$  ;
16       $\mathcal{P}_i.L_1, \mathcal{P}_i.L_2 \leftarrow$  Q3InternalSearch( $p, r$ ) // see Alg. 4.1;
17      // two lists are returned,  $L_1$  for point results,  $L_2$  for
18      // polygon results
19      for  $int\ j \leftarrow 0$  to  $|L_1| - 1$  do
20        // point records
21         $dist \leftarrow L_1[j].distance$  // distance to  $p$ ;
22         $finalScore \leftarrow$  getCombinedScore( $R_{td}, dist$ ) // eq. (4.11);
23        TexSpaPointResult  $r_1 \leftarrow$  generateQ2Point( $\mathcal{P}_i, L_1[j], finalScore$ );
24         $R_1 = R_1 + r_1$  ;
25      for  $int\ k \leftarrow 0$  to  $|L_2| - 1$  do
26        // polygon records
27         $dist \leftarrow L_2[k].distance$  // distance to  $p$ ;
28         $finalScore \leftarrow$  getCombinedScore( $R_{td}, dist$ ) // eq. (4.11);
29        TexSpaPolygonResult  $r_2 \leftarrow$  generateQ2Polygon( $\mathcal{P}_i, L_2[k], finalScore$ )
30        ;
31         $R_2 = R_2 + r_2$  ;
32
33 25 Sort  $R_1$  based on finalScore ;
34 26 Sort  $R_2$  based on finalScore ;
35 27 return returnList  $\leftarrow$  generateReturnList( $R_1, R_2$ ) ;
```

Table 4.1: TexSpaSearch search engine source code summary.

Module	Program name	Lines of codes	Comments
R*tree	RStarBoundingBox.h	282	R* bounding box manipulations
	RStarVisitor.h	112	'acceptor' functions used for queries and removals
	RStarTree.h	726	R* tree library
Suffix tree	BuildRStarTree.h	644	R* tree constructor
	nPatriciaTrie.h	508	Suffix tree library
TexSpaSearchEngine	BuildSuffixTrie.h	620	Suffix tree constructor
	TexSpaSearch.h	1237	Functions for Q1, Q2 and Q3 queries
	ServerEndTest.h	818	Server end test
	Communicate.h	275	Communication with Java program
	RStarTest.cpp	52	Main method for integration test and server running
Java side	HTMLHandlerServlet.java	808	Java Servlet class
	ResultClass.java	286	The Java class for holding search results
	TimeBean.java	47	The Java class for recording time
Client side	index.jsp	736	client side .jsp page file
ShapeSimplify	EarthGeometry.java	149	The Java class for computing geometry parameters
	TrackPoint.java	31	The Java class defines points
	ShapeSimplify.java	278	Shape simplification
Data Preprocessing	DataPreprocessing.java	652	Data preprocessing for the source data getting from UNB Connell Memorial Herbarium database
	GenerateWebpage.java	310	Generate web pages from the preprocessed data

Chapter 5

Google Search Appliance (GSA)

Indexing

The Google Search Appliance is a rack-mounted device providing document indexing functionality that can be integrated into an intranet, document management system or web site using a Google search-like interface for end-user retrieval of results. The operating system is based on CentOS [9]. The GSA software that we use is from a GSA model GB-7007-1M running version 6.8.0.G.30. GSA provide us efficient ways of indexing web pages, performing text queries and ranking the query results. We first introduce the basic scoring algorithm that GSA uses to rank the search results and the Result Biasing Policy which is used for exerting influence on the ranking of the final results. We then describe how to configure the GSA for crawling and indexing.

5.1 PageRank Algorithm

PageRank is used by Google together with a number of different factors, including standard information retrieval (IR) measures, proximity, and anchor text (text of links

pointing to Web pages) in order to find most relevant answers to a given query [28]. Other than document collections, web pages on the web are hypertext and provide plenty of auxiliary information in the metadata of the web pages, such as link structure and link text [?]. The PageRank algorithm makes use of these features. The algorithm is based on the directed graph created by treating web pages as nodes and hyperlinks as edges [17]. Google’s PageRank algorithm assesses the importance of web pages without human evaluation of the content. Google claims, “the heart of our software is PageRank” [25].

The basic idea behind PageRank is that a page is ranked higher if there are more links to it. More specifically, PageRank is a probability distribution which is created to represent the likelihood that a person randomly clicking on links will arrive at any particular page [17]. A probability is expressed as a numeric value between 0 and 1. A PageRank of 0.5 means there is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank [25]. In this way, web pages with the highest PageRank value will appear on the top of the search results.

5.2 Result Biasing

The Google Search Appliance (GSA) provides multiple ways to exert influence on the search results. This includes Source Biasing, Date Biasing and Metadata Biasing. Metadata Biasing enables us to influence the order of documents based on metadata associated with a document. For example, we might use metadata biasing to increase the score of documents whose document type is “Article”. We can create a biasing scheme with the user interface depicted in Figure 5.1 [34].

To use Metadata Biasing, we can first change the influence setting from “No influence”

Metadata Biasing [\(Help\)](#)

Metadata biasing lets you increase or decrease a document's score when it contains a <meta> tag with one of the name:content pairs specified below.

How much influence should metadata biasing have?

No influence More influence

When the search appliance scores a document, it compares the values in the document's <meta> tags to the name:content pairs below. When a name:content pair below matches a name:content pair in a <meta> tag, the search appliance uses the Strength setting to adjust the document's score.

To remove an entry, make the name and the content fields blank.

Metadata Name	Metadata Content	Strength
<input type="text" value="DC.type"/>	<input type="text" value="Article"/>	<input type="button" value="Strong increase"/> <input type="button" value="⬆️"/>
<input type="text"/>	<input type="text"/>	<input type="button" value="Leave unchanged"/> <input type="button" value="⬆️"/>
<input type="text"/>	<input type="text"/>	<input type="button" value="Leave unchanged"/> <input type="button" value="⬆️"/>

Figure 5.1: The Metadata biasing user interface from a GSA model GB-7007-1M running version 6.8.0.G.30 of the GSA software.

to a stronger setting. We can make a specific adjustment by compiling a list of metadata tags. Documents are modified to include metadata tags corresponding to the metadata attributes described in the Metadata Biasing entries. As explained in [1], a metadata tag contains a name-value pair. An example name-value pair is as follows:

$$\langle \text{meta name}=\text{"DC.type"} \text{ content}=\text{"Article"} \text{ xml:lang}=\text{"en"} \rangle \quad (5.1)$$

We have seven choices on the influence strength of each tag: Strong decrease, Medium decrease, Weak decrease, Leave unchanged, Weak increase, Medium increase and Strong increase. We assume these tags have values $-e_3, -e_2, -e_1, 0, e_1, e_2, e_3$, respectively, where e_1, e_2 and e_3 are $\in \mathbf{R}^+$, and $e_1 < e_2 < e_3$. When the search appliance ranks search results, it compares metadata tags with each pattern in the list. For each document, the search appliance traverses the list in the order we specify the metadata tags from top to bottom, and compares the tags with the document's metadata. The search appliance makes only one score adjustment for each document. Once a tag matches a document, the score of

the document is modified, and the search appliance continues with the next document to be rescored, to see if the document matches any metadata tag [1].

Assume for a specific web page \mathcal{G} , the PageRank value is $W_{pr}(\mathcal{G})$. The general degree of influence that Metadata Biasing has is represented by f , which is one of 11 nonnegative numbers $\in [0, f_{max}]$. The score of the first matched tag in the document's metadata is represented by W_{mb} . As there are seven possible degrees of strength for each tag, the domain of W_{mb} should be seven numbers corresponding to seven degrees as described above. $W_{mb} = 0$ corresponds to the strength of a tag setting to "Leave unchanged". Based on the documentation we can find, we estimate that the final rank R_i of web page \mathcal{G}_i is as follows [34]:

$$R_i = W_{pr}(\mathcal{G}_i) + fW_{mb}(\mathcal{G}_i) \quad (5.2)$$

Assuming that we have a web of four pages, the PageRanks for page 1, 2, 3 and 4 are $W_{pr}(1) \approx 0.368$, $W_{pr}(2) \approx 0.142$, $W_{pr}(3) \approx 0.288$ and $W_{pr}(4) \approx 0.202$, respectively. The ranking of these pages is 1, 3, 4, 2 from top to bottom. Now assume we have the name:content pair in the Metadata Biasing list, which is meta name = "DC.type" and meta content = "Article". The strength is set to "Strong increase". In the four pages, we assume only pages 2 and 3 have metadata tags that agree with this name:content pattern. If our equation (3.2) is correct, the search appliance will make score adjustments for pages 2 and 3 as follows:

$$R_2 = W_{pr}(2) + fW_{mb}(2) = 0.142 + fW_{mb}(2)$$

$$R_3 = W_{pr}(3) + fW_{mb}(3) = 0.288 + fW_{mb}(3)$$

Assuming there is no other metadata tags in the list in Metadata Biasing scheme, then the search appliance will not make changes to the scores of pages 1 and 4. The final scores

for these two pages are as follows:

$$R_1 = W_{pr}(1) = 0.368$$

$$R_4 = W_{pr}(4) = 0.202$$

Consider a Metadata Biasing scheme in which the general degree of influence is set to the strongest degree. Assuming that $f \in [0, 1]$, then in this situation, $f = 1$. We assume the values $(e_1, e_2, e_3) = (0.05, 0.10, 0.15)$. With a W_{mb} value of “Strong increase” = e_3 for pages 2 and 4, we have the following four final ranks:

$$R_1 = 0.368$$

$$R_2 = 0.142 + 1 * 0.15 = 0.292$$

$$R_3 = 0.288 + 1 * 0.15 = 0.438$$

$$R_4 = 0.202$$

Thus, the final ranking of these four pages is 3, 1, 2, 4 from top to bottom. We can see that after Metadata Biasing, the scores of pages 2 and 3 are boosted, and the rank of every page has changed from that arising from the original rankings (i.e. rankings of 1, 3, 4, 2).

5.3 GSA operation

5.3.1 Configuring the GSA for Crawling and Indexing

We now have the web page collection in a form required for GSA crawling. Before crawling starts, we first configure the GSA for crawling the collection. We use the **Crawl and Index > Crawl URLs** page of the Admin Console of the GSA to configure a crawl of the URL patterns, as shown in Figure 5.2.

In the field “Start Crawling from the Following URLs”, we added the following line:

```
http://131.202.243.11/test/HerbariumDatabase/
```

This is the start URL which controls where the GSA begins crawling the content. In the

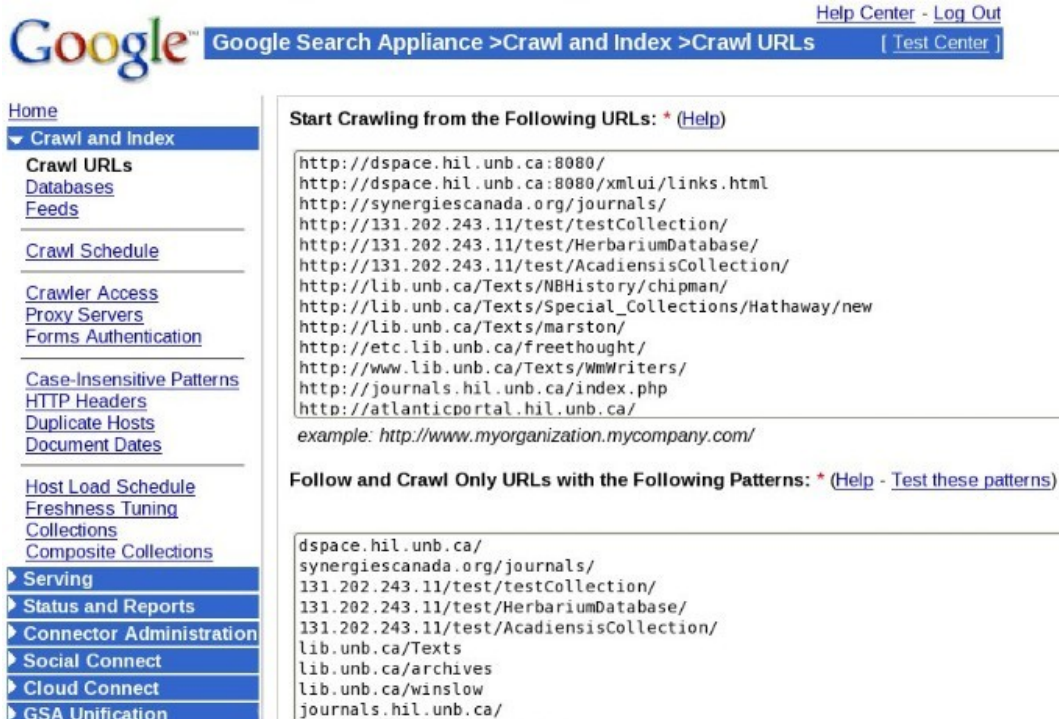


Figure 5.2: The page Crawl and Index > Crawl URLs in the Admin Console of GSA software.

field “Follow and Crawl Only URLs with the Following Patterns”, we added the following line:

```
131.202.243.11/test/HerbariumDatabase/
```

In this way, only URLs matching the patterns we specify in this field are followed and crawled. In the field “Do Not Crawl URLs with the Following Patterns”, we added the following lines:

```
# Test collection on ib214m20.cs.unb.ca herbarium database(Do not crawl
list)
regexp:http://131.202.243.11/test/HerbariumDatabase/.*statistics$
regexp:http://131.202.243.11/test/HerbariumDatabase/.*show=full$
regexp:http://131.202.243.11/test/HerbariumDatabase/.*browse?
regexp:http://131.202.243.11/test/HerbariumDatabase/.*advanced-search$
```

These are URL patterns for specific file types, directories, or other sets of pages that we do not want crawled in this collection [10]. The \$ specifies the end of a string and forces matching only one web page. The “.*” matches any number of characters. To test URL patterns, click a “Test these patterns” link to open the Pattern Tester Utility. We can specify a list of URLs on the left and a set of patterns on the right. It notifies you if each URL is matched by one of the patterns in the set.

After configuring the GSA crawler, we can build a new collection in the GSA for the web pages transformed from the Herbarium database records. The user interface for managing collections is shown in Figure 5.3.

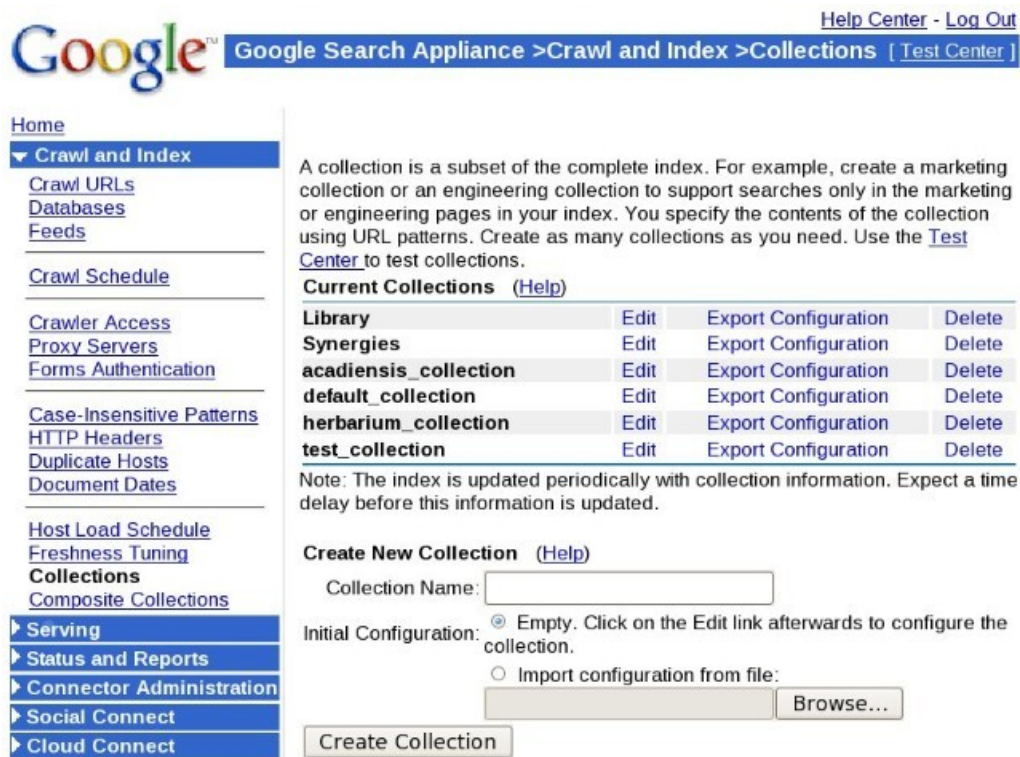


Figure 5.3: The page Crawl and Index > Collections in the Admin Console of GSA software.

We created a collection called “herbarium.collection” by using the Crawl and Index >

Collections page in the Admin Console. In the field “Include Content Matching the Following Patterns”, we entered the pattern as follows:

```
131.202.243.11/test/HerbariumDatabase/
```

Clicking the button “Save Collection Definition” creates the `herbarium_collection`.

In the next step, we associate the `herbarium_collection` with a front end generated using GSA software. We used the `test` front end to perform search of `herbarium_collection`. The `test` front end is the front end we built initially for exploring how GSA ranks search results and comparing Google search with Synergies search. Three collection (e.g. Synergies, `default_collection`, `test_collection`) are already associated with the “test” front end. The test front end can be accessed by the url `http://gsa1.lib.unb.ca/` as shown in Figure 5.4.

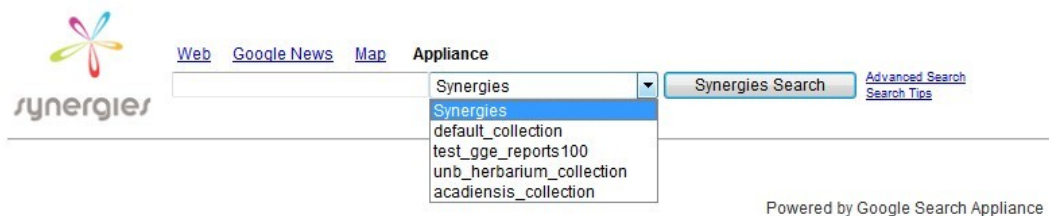


Figure 5.4: The XSLT code which is used to associate `herbarium_collection` to the `test` front end.

To associate the `herbarium_collection` to the `test` front end, we choose “Edit” for the `test` front end in the `Serving > Front Ends` page in Admin Console. Then we can view the XSLT stylesheet code of the `test` front end. Using `<Ctrl>F` in the browser, we found the section “Collection menu beside the search box” in the code, and added a paragraph as shown in Figure 5.5.

After adding this XSLT “choose” item, we clicked the button “Save XSLT Code” to

```

<xsl:choose>
  <xsl:when test="PARAM[(@name='site') and (@value='herbarium_collection')]">
    <option value="herbarium_collection"
selected="selected">unb_herbarium_collection</option>
  </xsl:when>
  <xsl:otherwise>
    <option value="herbarium_collection">unb_herbarium_collection</option>
  </xsl:otherwise>
</xsl:choose>

```

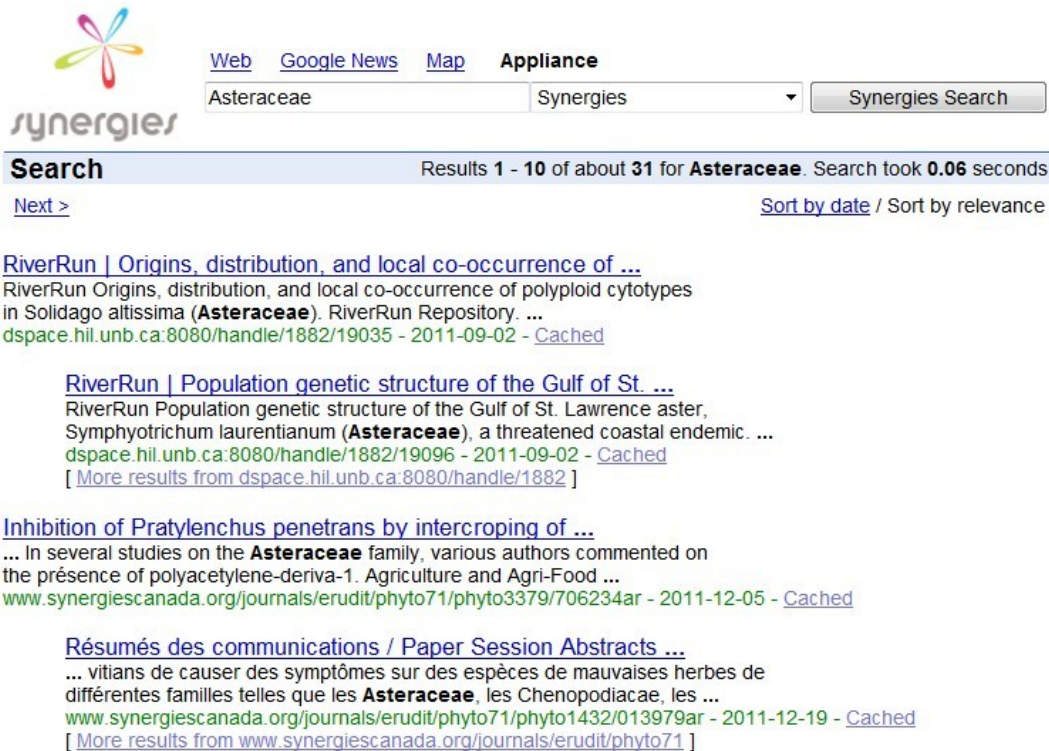
Figure 5.5: The XSLT code which is used to associate `herbarium_collection` to the `test` front end.

save the update. We can then use the `test` front end to perform text search under the `herbarium_collection`. We first open the web page <http://gsa1.lib.unb.ca/>. In the drop-down menu beside the search box, we choose the item `unb_herbarium_collection` and input a search string in the text field. Clicking the “Synergies Search” button, the corresponding search results are displayed. An example query with the query string “Asteraceae” is shown in Figure 5.6.

5.3.2 Build Result Biasing Policy

We can use the `Serving > Result Biasing` page to create or edit a result biasing policy. The search appliance ranks the documents that it finds in response to a user search query by calculating the PageRank score for each document [34]. The score reflects the probable relevance of the document content and determines the order in which results appear on the search results page . To influence search appliance rankings, use a result biasing policy [1].

To create a new result biasing policy, under `Serving > Result Biasing` subdirectory, we



The screenshot shows the Synergies search interface. At the top left is the Synergies logo, a stylized flower with five petals in red, orange, yellow, green, and blue. To the right of the logo are navigation links: [Web](#), [Google News](#), [Map](#), and **Appliance**. Below these is a search bar containing the text "Asteraceae" and a dropdown menu set to "Synergies". A "Synergies Search" button is to the right of the search bar. Below the search bar is a blue header bar with the word "Search" on the left and "Results 1 - 10 of about 31 for Asteraceae. Search took 0.06 seconds." on the right. Below the header bar are two links: "Next >" on the left and "Sort by date / Sort by relevance" on the right. The main content area displays five search results, each with a blue hyperlink for the title, a brief description, and a green URL with a "Cached" label. The results are:

- [RiverRun | Origins, distribution, and local co-occurrence of ...](#)
RiverRun Origins, distribution, and local co-occurrence of polyploid cytotypes in *Solidago altissima* (**Asteraceae**). RiverRun Repository. ...
dspace.hil.unb.ca:8080/handle/1882/19035 - 2011-09-02 - [Cached](#)
- [RiverRun | Population genetic structure of the Gulf of St. ...](#)
RiverRun Population genetic structure of the Gulf of St. Lawrence aster, *Symphytotrichum laurentianum* (**Asteraceae**), a threatened coastal endemic. ...
dspace.hil.unb.ca:8080/handle/1882/19096 - 2011-09-02 - [Cached](#)
[[More results from dspace.hil.unb.ca:8080/handle/1882](#)]
- [Inhibition of *Pratylenchus penetrans* by intercropping of ...](#)
... in several studies on the **Asteraceae** family, various authors commented on the presence of polyacetylene-deriva-1. Agriculture and Agri-Food ...
www.synergiescanada.org/journals/erudit/phyto71/phyto3379/706234ar - 2011-12-05 - [Cached](#)
- [Résumés des communications / Paper Session Abstracts ...](#)
... vitians de causer des symptômes sur des espèces de mauvaises herbes de différentes familles telles que les **Asteraceae**, les **Chenopodiaceae**, les ...
www.synergiescanada.org/journals/erudit/phyto71/phyto1432/013979ar - 2011-12-19 - [Cached](#)
[[More results from www.synergiescanada.org/journals/erudit/phyto71](#)]

Figure 5.6: An example query result from the `unb_herbarium_collection` with the query string “Asteraceae” using the `test` front end. Only the first five results are shown.

enter a name of the new result biasing policy in the `Result Biasing Name` text box, then we can edit the created policy by clicking the hyperlink “edit”. The GSA provides us 3 ways to exert influence on the documents’ scores. They are Source Biasing, Date Biasing and Metadata Biasing. The user interface for configuring the Metadata Biasing is shown in Figure 5.1. We have already talked about how to use the Metadata Biasing in Section 5.2.

The Source Biasing enables us to increase or decrease a document’s score when it belongs to a specified collection or its URL matches a specified pattern. The user interface for Source Biasing is shown in Figure 5.7. To configure a Source Biasing, we first set the strength of general influence. Similar to the Metadata Biasing, there are 11 different general degrees of influence, which can be considered as 11 nonnegative numbers [34].

If we want to configure Source Biasing by Collection, we select **Collection** from the pull-down menu and select a collection name of which we want to exert influence. Then we specify the strength of this adjustment. Similar to the Metadata Biasing, we have seven choices on the influence strength of each adjustment: Strong decrease, Medium decrease, Weak decrease, Leave unchanged, Weak increase, Medium increase and Strong increase. If we want to configure Source Biasing by URL Pattern, we select **URL Pattern** from the pull-down menu. For each URL we want to affect, we enter a pattern that the URL matches. Then we specify the strength of each pattern we entered.

Source Biasing [\(Help\)](#)
 Source biasing lets you increase or decrease a document's score when its URL matches one of the patterns or belongs to a collection specified below.

How much influence should source biasing have?

No influence

 More influence

The search appliance tries to match each URL in a result set to a collection or a URL pattern, starting from the top of the pattern list. When a URL matches, the search appliance applies the specified tuning and then tries to match the next URL. Only the first applicable biasing rule is applied to any given URL.

To remove an entry, change it to blank URL pattern and save.

URL Pattern or Collection	Strength
URL Pattern ▾	Leave unchanged ▾
URL Pattern ▾	Leave unchanged ▾

Figure 5.7: The user interface for configuring Source Biasing.

The Date Biasing enables us to increase the score of more recent documents relative to older documents [1]. The user interface for Data Biasing is shown in Figure 5.8. To use the Data Biasing, we first specify how much we want our adjustment to influence the scoring calculation . Optionally, to specify a time period for considering documents moderately old, click the check box and choose a time period from the list [1]. Finally, click **Save Settings** to save the configuration for Result Biasing.

Date Biasing [\(Help\)](#)
Date biasing lets you increase the score of more recent documents relative to older documents.
How much influence should date biasing have?

No influence More influence

Consider documents moderately old after

Figure 5.8: The user interface for configuring Data Biasing.

Chapter 6

Test Results

6.1 R* Tree and Suffix Tree Construction

In the R* tree construction, there are 28,435 records having (ϕ, λ) pairs associated, 11,909 records having county names but no (ϕ, λ) pairs associated, and 447 records having no related spatial information ((ϕ, λ) pair or county name). The total number of records in the Herbarium database we used (circa 2012) is 40,791, in which 40,344 records are indexed by the R* tree.

While inserting point records into the R* tree, there are 8291 distinct (ϕ, λ) pairs obtained, each associated with one or more records represented by their primary keys. For each (ϕ, λ) pair, the average number of keys indexed c is 3.46726. In the `packPoints(allPoints)` process in the R* tree construction (see Section 4.2.2), we pack up to B neighbouring points together to make the generated `RStarDataObject` (see Section 4.2.1) approximately fit on one disk block. As shown in Figure 4.9, for the point data, each `RStarDataObject` contains a Boolean flag `isPolygon` and a list of B `DataPoints` consisting of a (ϕ, λ) pair and a primary key list. The average number of keys per point is $c = 3.46726$, and the maximum

length of the primary key is $l_p = 10$ characters, so the value of B can be computed as follows:

$$\text{sizeof}(\text{boolean}) + B(2 * \text{sizeof}(\text{double}) + l_p * c * \text{sizeof}(\text{char})) < \text{Block_size} \quad (6.1)$$

or:

$$B < (\text{Block_size} - \text{sizeof}(\text{boolean})) / (2 * \text{sizeof}(\text{double}) + l_p * c * \text{sizeof}(\text{char})) \quad (6.2)$$

We have $B = 19$ for $\text{Block_size} = 1024$, $c = 3.46726$, $l_p = 10$, $\text{sizeof}(\text{boolean}) = 1$, $\text{sizeof}(\text{double}) = 8$ and $\text{sizeof}(\text{char}) = 1$.

The time for constructing the R* tree (the entire process shown in Figure 4.10) is 249.213 seconds. The time for constructing the suffix tree (the entire process shown in Figure 4.4) is 142.157 seconds. There are several key points in the entire query process, as shown in Figure 6.1.

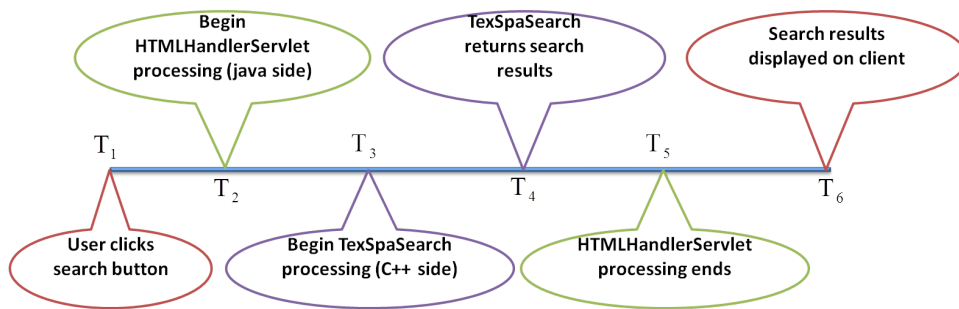


Figure 6.1: Several key points in the entire query process.

The key points for the subsequent queries that are sent by clicking a page number are shown in Figure 6.2.

For measuring the timing in the tests, we define the following:

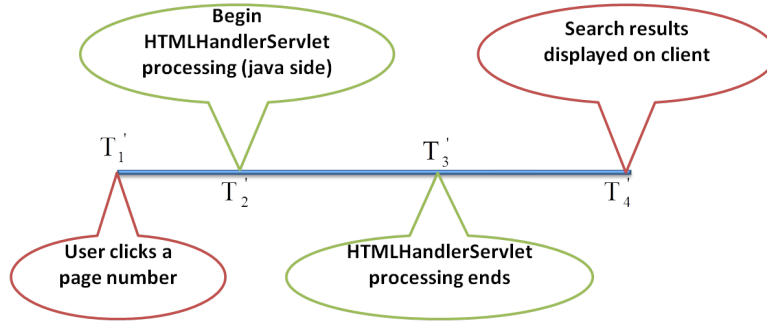


Figure 6.2: Several key points in the query process for subsequent queries sent by clicking a page number.

1. r : radius for Q2 and Q3 query in meters.
2. N_r : is the total number of search results returned by a Q1 query.
3. N_{pt} : the total number of (ϕ, λ) pairs returned for Q2 and Q3 point results.
4. R_{pt} : the total number of records returned for Q2 and Q3 point results.
5. N_{pl} : the total number of counties returned for Q2 and Q3 polygon results.
6. R_{pl} : the total number of records returned for Q2 and Q3 polygon results.
7. $T_c = T_4 - T_3$, which is the C++ side processing time.
8. $T_s = T_5 - T_2$, which is the server side (including C++ and Java) processing time.
9. $T_e = T_6 - T_1$, which is the total query time from the user clicking the search button to the search results displayed on the web page. So T_e is the total time for the **Text Search with Spatial Constraints Search Engine**.
10. $T_p = T'_4 - T'_1$, where queries exclude C++ processing as shown in Figure 6.2. T_p is the page query timing, which is the timing for subsequent requests sent by clicking a page number. Note that the first query response is longer, especially for Q2 and Q3 query.

11. n_p : the number of subsequent page queries we performed for a Q1 query. When there are more than 10 pages of search results, we only test the page query timing for the first 10 pages; otherwise, we perform page query for all the existing pages.
12. n_{pt} : the number of subsequent page queries we performed for a Q2 or Q3 point query. When there are more than 5 pages of results returned, we perform page query for the first 5 pages in the point results; otherwise, we perform page query for all the existing pages.
13. n_{pl} : the number of subsequent page queries we performed for a Q2 or Q3 polygon query. When there are more than 5 pages of results returned, we perform page query for the first 5 pages in the polygon results; otherwise, we perform page query for all the existing pages.

The `TexSpaSearch` testing environment had the web server and web browser running on the same workstation in the ITB214 Communication and Networking Laboratory.

6.2 Q1 Test Results

We choose 20 sample query strings for Q1 test as shown in Table 6.1.

The line graph in Figure 6.3 shows the changes of the T_s and T_e measured in ms versus N_r . As we can see, T_s and T_e rise proportionately to the number of results returned. Approximately 500 ms offset between the two curves is constant for most values of N_r , and is due to the overhead of sending the query request and response from and to the client, and dynamically generating the displayed results. Since each page can only display limited number of records, so data contained in the generated web page is approximately constant, which leads to a constant value of $T_e - T_s$, the average of which is 507.2. To return certain

Table 6.1: Q1 test results for the 20 sample queries. All times are shown in ms.

Query string	N_r	T_c	T_s	T_e	T_p		
					n_p	Avg.	St. dev.
apple	60	5	20	524	10	515.1	4.8408
crab	10	3	7	516	2	513.5	9.1924
crab apple	69	10	14	520	10	512.8	5.2451
Red spruce	2706	915	916	1424	10	513.9	4.9318
Hieracium pilosella	361	122	144	646	10	517.5	7.5755
Rosa virginiana	661	213	221	725	10	518	11.5758
seaside arrow grass	3759	1361	1366	1854	10	511.1	3.984693
Red pitcher plant	2149	744	750	1263	10	510.8	3.5214
pitcher plant	413	136	144	659	10	514.7	5.2079
Eupatorium perfoliatum	142	38	41	547	10	517.3	5.4375
Bromus Inermis Leyss	151	44	49	555	10	513.9	5.5867
Amelanchier laevis wieg	1244	449	452	955	10	518.3	15.8539
Vesce des haies	1959	698	700	1214	10	516.5	6.0782
Poison ivy	70	14	17	525	10	513.5	4.0346
Lilac	15	13	15	519	3	513.3	5.8595
Lady slipper	267	60	61	575	10	514.1	4.5570
lady's slipper	200	16	18	524	10	518.3	9.6500
Herbe aux écrevisses	1054	481	487	986	10	516.6	9.8229
Verge d'or des bois	2654	943	946	1450	10	520.4	13.2262
vanilla	43	14	28	535	9	514.7778	11.6809

amount of results, the time cost for the entire search process (T_e) is significantly higher than the server side (including C++ and Java) processing time T_s . The processing time T_s is slightly higher than T_c as shown in Table 6.1, which accounts for the extra time (5.65ms, on average) for Java to reformat the search results for web display. For the Q1 test, the average costs for T_c , T_s and T_e are 315.05ms, 323.95ms and 825.8ms, respectively. The average number of records returned is 899.35.

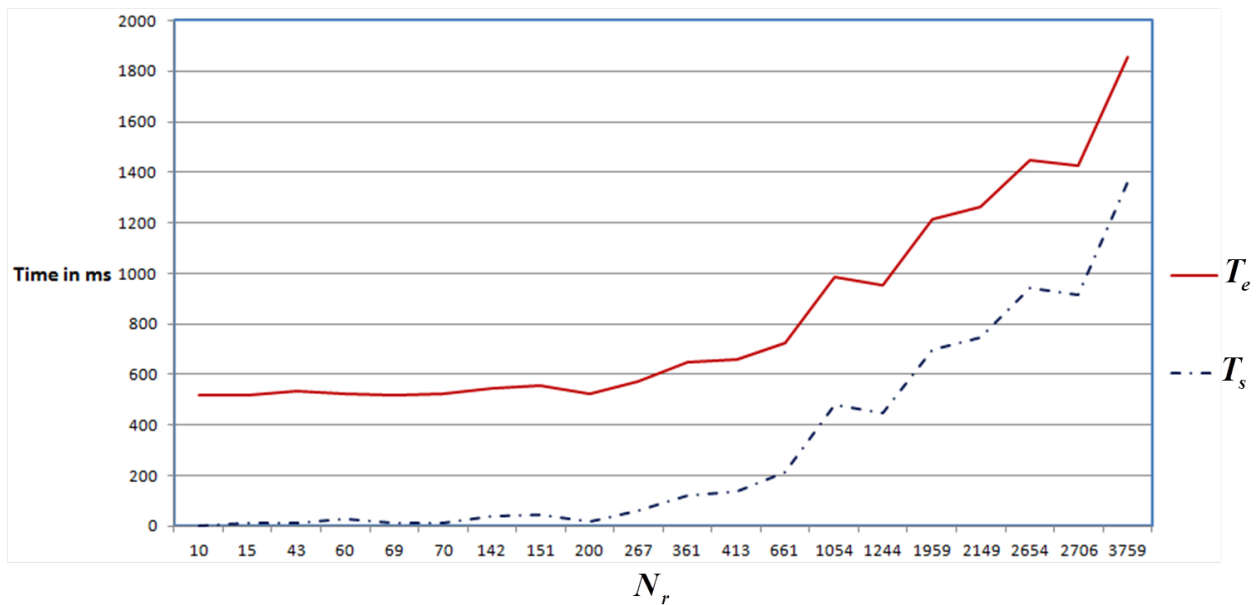


Figure 6.3: Search engine server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number N_r of returned search results for Q1 queries.

6.2.1 Comparing with GSA test results

Let N_g stand for the total number of search results returned by a GSA query, and T_g stand for the corresponding total query time. The comparison of the **TexSpaSearch** engine and the **Google Search Appliance (GSA)** on the 20 sample queries is shown in Table 6.2. Table 6.3 shows the time (in ms) per returned record for Q1 search results of **TexSpaSearch** and the **GSA**. Notice that the T_g/N_g value of Red pitcher plant is not counted for calculating

the average value of T_g/N_g .

Table 6.2: The comparison of the TexSpaSearch engine and the Google Search Appliance (GSA) on the 20 sample queries. All times are shown in ms.

Query string	TexSpaSearch			GSA		T_e/T_g	T_c/T_g
	N_r	T_e	T_c	N_g	T_g		
apple	60	524	27	59	20	26.2	1.35
crab	10	516	3	10	20	25.8	0.15
crab apple	69	10	520	1	10	52	1
Red spruce	2706	1424	915	290	30	47.7	30.5
Hieracium pilosella	361	646	122	40	10	64.6	12.2
Rosa virginiana	661	725	213	72	20	36.25	10.65
seaside arrow grass	3759	1854	1361	47	20	92.7	68.05
Red pitcher plant	2149	1263	744	0	20	63.15	37.2
pitcher plant	413	659	136	3	30	22.0	4.53
Eupatorium perfoliatum	142	547	38	32	20	27.4	1.9
Bromus Inermis Leyss	151	555	44	28	20	27.8	2.2
Amelanchier laevis wieg	1244	955	449	188	20	47.8	22.45
Vesce des haies	1959	1214	698	30	10	121.4	69.8
Poison ivy	70	525	14	40	20	26.3	0.7
Lilac	15	519	13	15	20	26.0	0.65
Lady slipper	267	575	60	119	20	28.75	3
lady's slipper	200	524	16	119	20	26.2	0.8
Herbe aux écrevisses	1054	986	481	54	20	49.3	24.05
Verge d'or des bois	2654	1450	943	34	20	72.5	47.15
vanilla	43	535	14	43	20	26.8	0.7
Average						45.5	16.95

The times for GSA queries were obtained by manually recording the search time and count shown in the upper right corner of the test front end (see e.g. Figure 5.6). The GSA was residing in the UNB server facility located adjacent to the building where the work station running the browser was located. From Tables 6.2 and 6.3, and the search results, we make the following observations:

1. The query time for GSA is significantly lower than that for TexSpaSearch. If we assume that the GSA reported search time T_g includes the time to display the search

Table 6.3: Time (in ms) per returned record for Q1 search results of TexSpaSearch and the GSA.

Query string	TexSpaSearch			GSA	
	N_r	T_e/N_r	T_c/N_r	N_g	T_g/N_g
apple	60	8.73	0.45	59	0.34
crab	10	51.6	0.3	10	2
crab apple	69	7.54	0.14	1	10
Red spruce	2706	0.53	0.34	290	0.10
Hieracium pilosella	361	1.79	0.34	40	0.25
Rosa virginiana	661	1.10	0.32	72	0.28
seaside arrow grass	3759	0.49	0.36	47	0.43
Red pitcher plant	2149	0.59	0.35	0	N.A.
pitcher plant	413	1.60	0.33	3	10
Eupatorium perfoliatum	142	3.85	0.27	32	0.63
Bromus Inermis Leyss	151	3.68	0.29	28	0.71
Amelanchier laevis wieg	1244	0.77	0.36	188	0.11
Vesce des haies	1959	0.62	0.36	30	0.33
Poison ivy	70	7.5	0.2	40	0.5
Lilac	15	34.6	0.87	15	1.33
Lady slipper	267	2.15	0.22	119	0.17
lady's slipper	200	2.62	0.08	119	0.17
Herbe aux écrevisses	1054	0.94	0.46	54	0.37
Verge d'or des bois	2654	0.55	0.36	34	0.59
vanilla	43	12.44	0.33	43	0.47
Average	899.35	7.18	0.34	61.2	1.51

results on the screen, then the GSA is 45.5 times faster (on average) than TexSpaSearch. If the GSA is reporting only the search engine search time, then TexSpaSearch is 16.95 times slower than the GSA.

2. For some of the sample queries, TexSpaSearch returns more results than GSA. The reason is that GSA only returns records that contain all of the single words in the query string t , while TexSpaSearch returns records containing one or more subphrases of t . So we have: $N_g \leq N_r$.
3. Both TexSpaSearch and GSA index French characters well.
4. For the query strings like `Lady slipper` and `lady's slipper`, GSA treats them as the same phrase, while TexSpaSearch regards them as different phrases.
5. We checked 7 of the 20 samples (crab, crab apple, pitcher plant, Eupatorium perfoliatum, Bromus Inermis Leyss, Lilac, Verge d'or des bois), and the top N_g records in the TexSpaSearch Q1 results are exactly the same as the GSA search results. For 6 of the 7 search results, the ranking within those top N_g results is different. This is because GSA only returns results containing all the single words in the query string t . In TexSpaSearch Q1 query results, the records containing exactly t or all the single words in t usually rank higher than other records.
6. From Table 6.3 we can see that, although the efficiency for Q1 text search compared to the GSA is low (45.5 times lower for T_e and 16.95 times lower for T_c), the theoretical analysis does show a highly efficient search cost on the time per returned record when the number of results returned is large. For the sample query strings Red spruce, seaside arrow grass, Red pitcher plant, Amelanchier laevis wieg, Vesce des haies, Herbe aux écrevisses and Verge d'or des bois whose numbers of records N_r returned by TexSpaSearch are greater than 1000, the values of T_e/N_r are 0.53, 0.49, 0.59, 0.77,

0.62, 0.94 and 0.55, respectively, and the values of T_c/N_r are 0.34, 0.36, 0.35, 0.36, 0.36, 0.46 and 0.36, respectively, which are lower than the corresponding average value of TexSpaSearch and the average T_g/N_g of GSA.

6.3 Q2 Test Results

We tested Q2 text + spatial search using the 20 sample query strings with radius 2m, 20m and 200m respectively.

Table 6.4: Q2 test results for the 20 sample queries with radius 2m, 20m and 200m, respectively. All times are shown in ms.

Query string	r (m)	T_c	T_s	T_e	Point results				Polygon results					
					N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.	N_{pl}	R_{pl}	n_{pl}	Avg.	St.dev.
apple	5	2050	6973	7707	17	580	5	781.8	145.989	11	9672	4	7406.75	3616.092
	50	2088	7063	7692	18	581	5	817.6	281.078	12	10242	4	7547.75	3617.731
	500	2335	12949	14564	41	721	5	726.6	69.561	13	11138	5	6543.2	4180.488
crab	5	1105	6235	6941	5	42	2	666	38.184	6	5858	2	9783	5669.582
	50	1027	6203	6799	5	42	2	654.5	34.648	6	5848	2	9846.5	5811.711
	500	1249	6393	7067	26	369	5	630.8	51.480	7	6723	3	7296	5792.561
crab apple	5	2364	7277	7874	21	621	5	646	50.418	13	10849	5	6504	2875.790
	50	2727	7665	8356	22	622	5	710.8	180.131	14	11419	5	6649.8	2980.260
	500	2738	15168	15929	63	1085	5	732.6	82.754	14	11419	5	6578.8	4410.003
Red spruce	5	22778	36257	37018	594	8522	5	879.4	180.3846	15	11909	5	6830.8	4890.398
	50	23319	36987	37797	605	8593	5	808.2	158.823	15	11909	5	6847.4	4869.357
	500	28146	41758	42508	1482	10973	5	864	140.760	15	11909	5	6710.8	4368.432
Hieracium pilosella	5	6003	18093	18876	139	4189	5	810.2	188.205	15	11909	5	6644	4841.308
	50	6105	18179	18934	141	4201	5	821.6	177.503	15	11909	5	6279.6	3865.257
	500	7147	16881	17569	409	5203	5	771.8	195.580	15	11909	5	6719.8	4567.485
Rosa virginiana	5	11333	15080	15675	298	4562	5	618.4	33.938	15	11909	5	6763.4	3781.263
	50	11797	15606	16312	306	4624	5	710.6	132.709	15	11909	5	6809.6	3803.953

Continued on next page

Table 6.4 – continued from previous page

Query string	r (m)	T_c	T_s	T_e	Point results				Polygon results					
					N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.	N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.
seaside arrow grass	500	13895	20354	21012	774	6499	5	652.2	40.795	15	11909	5	6760.2	2920.250
	5	23276	25958	26630	752	10503	5	613	36.062	15	11909	5	6788.2	3736.452
	50	24068	26744	27340	759	10526	5	585.4	24.583	15	11909	5	6779.8	3817.644
	500	29512	32269	32937	1755	12878	5	630	39.326	15	11909	5	6804.2	3829.932
Red pitcher plant	5	20525	25206	25922	573	8650	5	625	27.102	15	11909	5	6744	2695.906
	50	20827	25577	26298	583	8679	5	643.8	32.576	15	11909	5	6885	2909.520
	500	25463	29567	30605	1442	11030	5	676.4	25.235	15	11909	5	6859.4	2981.659
pitcher plant	5	6944	14030	14778	143	2186	5	932	303.200	15	11909	5	6692.4	3124.997
	50	6617	13528	14255	145	2194	5	815.8	314.621	15	11909	5	6847.2	3212.638
	500	7949	14826	15455	467	3373	5	781.8	137.414	15	11909	5	6833.8	3234.025
Eupatorium perfoliatum	5	3874	10915	11633	75	2607	5	736.8	191.486	13	10355	5	6182.8	5146.128
	50	3975	11120	11843	75	2607	5	838.2	344.983	14	11013	5	6384.2	4937.478
	500	4555	11719	12440	189	2981	5	748	170.206	15	11909	5	6833.4	3453.863
Bromus Inermis Leyss	5	4517	6321	6918	92	3173	5	606.8	42.038	14	11481	5	6836.8	5141.141
	50	4478	6270	6974	93	3183	5	608.2	36.355	14	11481	5	6767.8	5077.076
	500	5271	7088	7669	303	3977	5	577	12.806	15	11909	5	6981.2	5229.476
Amelanchier laevis wieg	5	15377	19675	20389	385	4376	5	683.4	18.078	15	11909	5	7107.4	3407.886
	50	15726	20066	20799	391	4434	5	721.2	148.382	15	11909	5	6978.6	3039.904

Continued on next page

Table 6.4 – continued from previous page

Query string	r (m)	T_c	T_s	T_e	Point results				Polygon results					
					N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.	N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.
	500	18757	23051	23756	919	6381	5	660.4	29.501	15	11909	5	7067.4	3266.374
Vesce des haies	5	21069	37521	38388	670	9868	5	851	141.262	15	11909	5	6957.4	6043.914
	50	21622	38099	38880	682	9899	5	854.8	120.350	15	11909	5	7082.4	6327.991
	500	26267	42972	43806	1650	12338	5	757.2	34.230	15	11909	5	7068.2	5992.005
Poison ivy	5	3245	7465	8179	36	985	5	791.2	182.546	15	11909	5	6972.2	3405.361
	50	3165	6970	7561	37	986	5	859.6	304.341	15	11909	5	6855	3233.792
	500	3455	7239	7880	111	1307	5	671.6	42.465	15	11909	5	7019	6700.929
Lilac	5	446	6712	7379	3	96	1	695	N.A.	3	2596	1	683	N.A.
	50	725	7208	7728	3	96	1	708	N.A.	4	4401	2	8238	1804.537
	500	1037	7570	8276	5	98	2	862	226.274	6	6137	2	10872	5529.575
Lady slipper	5	4981	17184	17979	93	3279	5	783.4	51.457	15	11909	5	5123	3263.330
	50	4822	17157	17983	95	3285	5	778.6	66.455	15	11909	5	7092.2	4637.325
	500	5663	17794	18606	245	3841	5	746	91.829	15	11909	5	6989.2	4585.910
lady's slipper	5	2114	11264	12023	20	780	5	715.4	23.448	10	8958	4	7316.75	3126.277
	50	2236	11395	12128	20	780	5	777.2	127.270	11	9448	4	7392.25	3396.576
	500	2476	11642	12374	93	1100	5	760.8	164.185	11	9448	4	7282.25	3323.772
Herbe aux écrevisses	5	13238	22586	23346	405	8154	5	747.6	158.248	15	11909	5	6825.6	3462.329
	50	13442	22651	23388	412	8173	5	784.6	188.190	15	11909	5	7255.6	3284.714

Continued on next page

Table 6.4 – continued from previous page

Query string	r (m)	T_c	T_s	T_e	Point results				Polygon results					
					N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.	N_{pl}	R_{pl}	n_{pl}	Avg.	St.dev.
	500	16209	25432	26150	952	9726	5	807	176.229	15	11909	5	7143	3904.790
Verge d'or des bois	5	27905	37217	38156	843	11539	5	650.8	47.631	15	11909	5	7639.8	3660.444
	50	26969	36360	37086	860	11600	5	692.2	54.861	15	11909	5	7458	3622.664
	500	33047	42483	43220	1975	14223	5	719.2	68.126	15	11909	5	7886	3707.809
vanilla	5	1795	10925	11552	13	229	5	685.4	51.714	10	8493	4	7549.5	3365.583
	50	1843	10982	11747	14	239	5	653.2	44.556	10	8493	4	7389.25	3473.257
	500	1503	10561	11301	36	400	5	698.2	25.371	11	9211	4	7696.5	3789.028

Compared to Q1 results, Q2 test results shown in Table 6.4 take 22.8 more time. A main reason is the search complexity on the C++ side is higher than that of Q1. The line graph in Figure 6.4 shows the changes of the T_c , T_s and T_e measured in ms with increasing $R_{pt} + R_{pl}$. As we can see, T_c , T_s and T_e rise proportionately to the total number of point results and polygon results returned. Note that the line of T_c fluctuates more frequently than that in Q1. A possible reason leading to the fluctuation is that the time for a Q2 query heavily depends on the number of text search results returned, which is an uncertain factor. Similar to Q1 results, the constant value of $T_e - T_s$ still holds, the average of which is 732.45 ms. To return certain amount of results, the time cost for the entire search process (T_e) and the server side (including C++ and Java) processing time T_s are significantly higher than the C++ processing time T_c . In Table 6.4, the average value of $T_s - T_c$ is 7619.32. The value of $T_s - T_c$ is considerably higher than that in Q1, because the format of `returnList` returned by C++ of Q2 is more complex than Q1, which leads to a higher overhead on analysing and interpreting the `returnList` to generate the `ResultClass` object. The total number of results returned by Q2 queries being higher than that of Q1 queries also gives rise to the larger $T_s - T_c$ values. For the Q2 test, the average T_c , T_s and T_e are 10488ms, 18107.3ms and 18839.8ms, respectively, and the average value of $R_{pt} + R_{pl}$ is 15433.6.

6.4 Q3 Test Results

We tested Q3 point + radius search using 15 sample query points spread in the 15 counties in New Brunswick, Canada with radius 5m, 50m and 500m and 5000m, respectively.

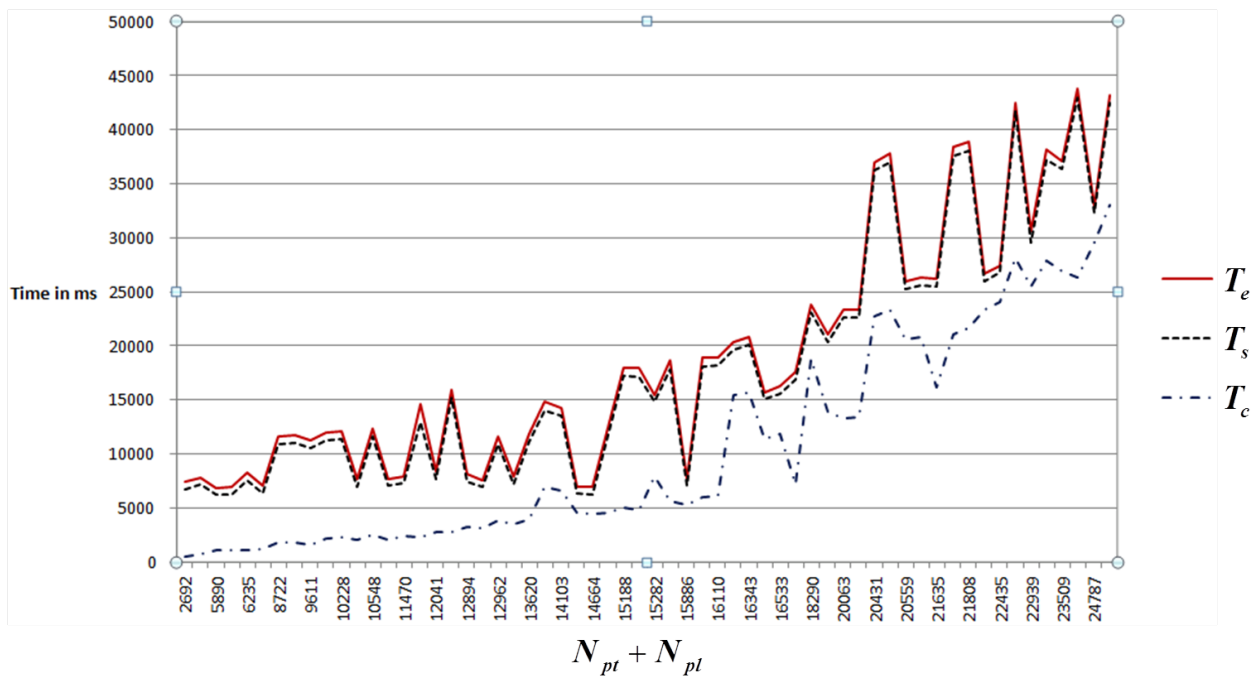


Figure 6.4: Search engine C++ processing time T_c , server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number of returned search results for Q2 queries $R_{pt} + R_{pl}$.

Table 6.5: Q3 test results for the 15 sample query points with radius 5m, 50m, 500m and 5000m, respectively. All times are shown in ms.

Query point	r (m)	T_c	T_s	T_e	Point results					Polygon results				
					N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.	N_{pt}	R_{pt}	n_{pt}	Avg.	St.dev.
(46.888 , -65.513)	5	163	3733	4265	1	2	1	530	N.A.	1	1166	1	573	N.A.
	50	156	3718	4287	1	2	1	537	N.A.	1	1166	1	549	N.A.
	500	298	6071	6655	4	24	2	695.5	3.536	2	2062	1	617	N.A.
	5000	294	6024	6725	4	24	2	698.5	10.607	2	2062	1	679	N.A.
(45.827 , -67.549)	5	307	7538	8445	1	14	1	685	N.A.	1	1805	1	634	N.A.
	50	234	7502	8092	1	14	1	569	N.A.	1	1805	1	651	N.A.
	500	226	7404	8022	2	19	1	581	N.A.	1	1805	1	599	N.A.
	5000	294	7519	8156	11	78	4	578.25	29.250	1	1805	1	553	N.A.
(47.892 , -66.954)	5	153	1932	2560	1	55	1	575	N.A.	1	875	1	523	N.A.
	50	152	1888	2426	1	55	1	540	N.A.	1	875	1	635	N.A.
	500	144	1831	2381	5	60	2	558.5	28.991	1	875	1	565	N.A.
	5000	219	2056	2596	49	146	5	557.4	33.687	1	875	1	532	N.A.
(47.05 , -67.736)	5	58	455	962	1	2	1	529	N.A.	1	443	1	510	N.A.
	50	117	722	1279	1	2	1	558	N.A.	2	724	1	531	N.A.
	500	123	710	1272	11	56	4	569.5	38.267	2	724	1	540	N.A.
	5000	132	731	1259	27	86	5	564.4	21.847	2	724	1	534	N.A.
(46.55 , -65.116)	5	177	2348	2893	1	5	1	539	N.A.	1	896	1	539	N.A.

Continued on next page

Table 6.5 – continued from previous page

Query point	$r(m)$	T_c	T_s	T_e	Point results					Polygon results				
					N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$	N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$
	50	121	2264	2822	1	5	1	552	N.A.	1	896	1	536	N.A.
	500	258	5925	6589	2	8	1	693	N.A.	2	2062	1	704	N.A.
	5000	261	5965	6541	24	85	5	673.4	49.334	2	2062	1	700	N.A.
(47.646 , -65.677)	5	127	1138	1722	1	7	1	573	N.A.	1	718	1	557	N.A.
	50	98	1106	1643	1	7	1	542	N.A.	1	718	1	555	N.A.
	500	107	1128	1663	4	16	2	535.5	3.536	1	718	1	539	N.A.
	5000	130	1176	1712	21	87	5	538.2	11.432	1	718	1	549	N.A.
(45.935 , -66.067)	5	101	406	914	1	27	1	512	N.A.	1	359	1	517	N.A.
	50	61	382	888	1	27	1	512	N.A.	1	359	1	512	N.A.
	500	72	374	880	4	45	2	522.5	6.364	1	359	1	509	N.A.
	5000	436	10037	10668	45	177	5	682	44.480	3	2845	1	623	N.A.
(46.067 , -65.092)	5	77	765	1271	1	1	1	525	N.A.	1	570	1	507	N.A.
	50	191	1805	2445	1	1	1	582	N.A.	2	1228	1	556	N.A.
	500	179	1753	2285	2	2	1	538	N.A.	2	1228	1	573	N.A.
	5000	248	2387	2983	10	24	4	591.5	41.621	3	1718	1	647	N.A.
(45.9 , -64.583)	5	74	783	1291	1	2	1	532	N.A.	1	570	1	545	N.A.
	50	159	1767	2319	1	2	1	541	N.A.	2	1228	1	546	N.A.
	500	163	1753	2293	1	2	1	537	N.A.	2	1228	1	548	N.A.
	5000	209	1787	2342	7	15	3	535	8.718	2	1228	1	621	N.A.

Continued on next page

Table 6.5 – continued from previous page

Query point	$r(m)$	T_c	T_s	T_e	Point results					Polygon results				
					N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$	N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$
(47.664 , -68.315)	5	46	260	767	1	11	1	516	N.A.	1	281	1	509	N.A.
	50	43	247	755	1	11	1	518	N.A.	1	281	1	506	N.A.
	500	164	2215	2775	2	12	1	544	N.A.	2	1156	1	572	N.A.
	5000	188	2208	2738	22	61	5	553	20.652	2	1156	1	585	N.A.
(46.152 , -67.598)	5	143	2742	3277	1	31	1	550	N.A.	1	1019	1	574	N.A.
	50	145	2765	3306	1	31	1	554	N.A.	1	1019	1	545	N.A.
	500	363	10891	11592	5	39	2	733	45.255	2	2824	1	730	N.A.
	5000	470	11095	11840	43	155	5	687.4	45.092	2	2824	1	735	N.A.
(45.18 , -67.296)	5	179	6180	6991	1	3	1	573	N.A.	1	1520	1	601	N.A.
	50	215	6127	6793	1	3	1	676	N.A.	1	1520	1	660	N.A.
	500	208	6130	6738	6	54	2	575.5	2.121	1	1520	1	663	N.A.
	5000	227	6191	6804	33	170	5	601	43.174	1	1520	1	596	N.A.
(45.872 , -66.445)	5	140	1389	1924	1	6	1	561	N.A.	1	681	1	544	N.A.
	50	106	1440	1968	2	9	1	555	N.A.	1	681	1	572	N.A.
	500	272	9784	10363	4	11	2	673	4.243	2	2486	1	676	N.A.
	5000	488	10723	11495	27	49	5	706.8	18.254	4	3335	2	6525	8051.118
(45.583 , -64.961)	5	96	1121	1628	1	1	1	545	N.A.	1	658	1	509	N.A.
	50	102	1081	1588	1	1	1	528	N.A.	1	658	1	525	N.A.
	500	433	10113	10726	9	39	3	694.667	23.438	3	2891	1	1006	N.A.

Continued on next page

Table 6.5 – continued from previous page

Query point	$r(\mathbf{m})$	T_c	T_s	T_e	Point results					Polygon results				
					N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$	N_{pt}	R_{pt}	n_{pt}	$Avg.$	$St.dev.$
	5000	793	15934	16681	53	142	5	764.8	28.350	5	4901	2	10049.5	11901.314
(45.259 , -66.088)	5	60	486	993	1	5	1	531	N.A.	1	428	1	535	N.A.
	50	61	481	990	1	5	1	518	N.A.	1	428	1	507	N.A.
	500	58	489	995	2	6	1	518	N.A.	1	428	1	513	N.A.
	5000	173	1215	1758	29	130	5	555.4	22.131	2	918	1	561	N.A.

The line graph in Figure 6.5 illustrates the changes of the T_c , T_s and T_e measured in ms with increasing $R_{pt} + R_{pl}$. As we can see, T_s and T_e show an increase in proportion to the total number of point results and polygon results returned; T_c remains relatively steady for most values of $R_{pt} + R_{pl}$ at around 200 ms. The steady performance of T_c is consistent with the constant average query complexity $O(\log_{\mathcal{M}} \mathcal{D}_n + y)$ of the R* tree, although $O(D_n)$ is the worst case time complexity for a rectangle intersecting a set of n records, where \mathcal{D}_n is the number of objects indexed in the R* tree, and y is the number of the records found in range (see section 4.6). Similar to Q1 and Q2, a constant value of $T_e - T_s$ still holds, with an average of 580.68 ms. The time cost for the entire search process (T_e) and the server side (including C++ and Java) processing time T_s are significantly higher than the C++ processing time T_c . In Table 6.5, the average value of $T_s - T_c$ is 3411.63. The value of $T_s - T_c$ is considerably higher than that for Q1, but lower than that for Q2. A main reason for the lower value is due to the size of the results returned by Q3, which is less than that of Q2, but generally more than that of Q1. The time spent on analysing and interpreting the `returnList` gives rise to these differences. For the Q3 test, the average T_c , T_s and T_e are 191.5ms, 3603.2ms and 4183.9ms, respectively, and the average value of $R_{pt} + R_{pl}$ is 1313.4.

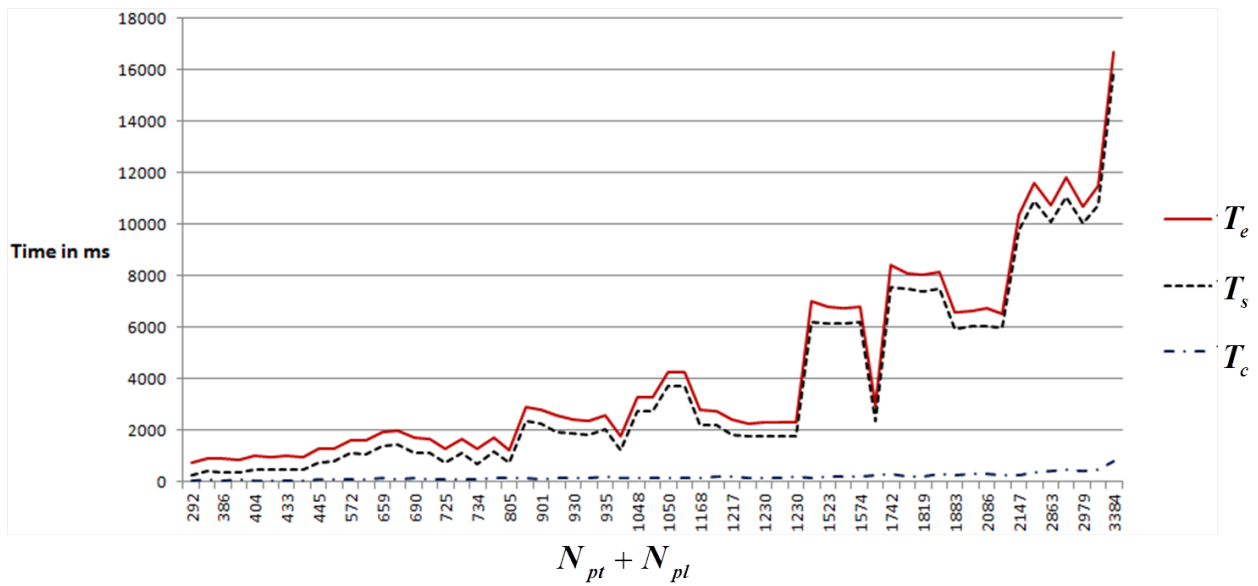


Figure 6.5: Search engine C++ processing time T_c , server side (including C++ and Java) processing time T_s and total query time T_e plotted versus the number of returned search results for Q3 queries $R_{pt} + R_{pl}$.

Chapter 7

Summary and Conclusions

7.1 Summary

A search engine system called `TexSpaSearch` supporting text with spatial constraints searching is presented in this thesis. A polygon simplification algorithm `PackPolygon` is proposed in the data preprocessing phase, which is a variant of the RDP algorithm. We indexed the UNB Connell Memorial Herbarium database of 40,791 records using the R* tree and suffix tree simultaneously, and designed data structures for efficiently combining them to realize the $Q1(t)$, $Q2(t, r)$ and $Q3(p, r)$ queries. A Lucene scoring algorithm is implemented to rank the text search results.

A Java-based web application was implemented to provide the web user interface for our search engine. It also handles sending and receiving requests and responses between the clients and the server, communicating with the C++ back end program through TCP sockets, analysing the results returned by the C++ program, and interpreting the results to generate the corresponding web pages. We use JSP pages to generate search results dynamically. The Java server transmits messages to the JSP pages using session objects.

7.1.1 Contributions

This thesis presents an innovative $Q2(t,r)$ query definition and text + spatial data structures supporting $Q2$ as well as $Q1(t)$ and $Q3(p,r)$ search simultaneously. To our knowledge, this is the first data structure supporting all three query types. Our data structure can also return points or polygons describing spatial objects resulting from $Q2$ or $Q3$ queries. Our R^* -tree data structure is, to our knowledge, the first one to efficiently pack simplified polygons defined by up to w_B points into the leaf nodes. Leaf nodes can also contain up to B points. A modified Lucene scoring algorithm was designed and implemented. This modified Lucene scoring uses words and subphrases instead of fields and multiword terms to provide a better way to index and search free-form-text. We also present a novel scheme for ranking combined text and spatial search results. To improve the efficiency of returning ranked text search results, our suffix tree data structure contains precomputed components (e.g. term frequency, document frequency) needed by the modified Lucene scoring algorithm. A unique scheme for displaying the $Q1$, $Q2$ and $Q3$ search results was designed and implemented. Although the efficiency for $Q1$ text search compared to the GSA is low, our theoretical analysis does show a highly efficient search cost when the number of results returned is large.

7.2 Conclusions

The `TexSpaSearch` engine can perform $Q1(t)$, $Q2(t,r)$ and $Q3(p,r)$ queries successfully, and can rank the search results reasonably. The experimental results on the 20 sample query strings for the $Q1$ text only query indicate an average 45.5 times slower search time for T_e (total query time) and 16.95 times slower search time for T_c (C++ side processing time) compared with a Google Search Appliance, but returns a wider range of results (records containing any subphrase of the query string) than the GSA. The average query time for

Q1 is 825.8ms, with on average 899.35 results returned, while the average theoretical query time for Q1 is $O(A^2\overline{|b|})$. For a Q2 query, the average query time is 18839.8ms to return an average number of records is 15433.6, and the average theoretical query time for Q2 is $O(A^2\overline{|b|} + Z \log_{\mathcal{M}} \mathcal{D}_n + y)$. Q3 test gives an average query time of 4183.9ms for an average 1313.4 returned records, when the average theoretical query time for Q3 is $O(\log_{\mathcal{M}} \mathcal{D}_n + y)$. A constant value of $T_e - T_s$, where T_e stands for the total query time and T_s is the server side (including C++ and Java) processing time, holds for all the three query types due to the data contained in the generated web page being approximately constant (the number of results displayed on each page is limited to e.g. 5 for Q1, and 3 for Q2 and Q3).

7.3 Future Work

Future work on improving and testing the `TexSpaSearch` engine might include the following topics:

1. Time did not permit us to complete the nearest neighbour filtering of GSA ranked results as planned. Nearest neighbour filtering of GSA search results might be added in the future.
2. We implement a Q2 text + spatial query by performing the text search first. Can the Q2 query be achieved in the inverse order (perform spatial query first)?
3. It would be useful to test the performance of `TexSpaSearch` on different operating systems, web browsers and on a much larger database.
4. Evaluation of the search engine by those with domain knowledge (e.g. biologists for the herbarium database) would be valuable to determine how the `TexSpaSearch` engine could be improved.

5. How can support for approximate spatial search be incorporated? For example an approximate query might be “find pitcher plants near Fredericton”.

References

- [1] <http://code.google.com/apis/searchappliance/documentation/610/>, accessed May 22, 2011.
- [2] *Apache http server project*, <http://httpd.apache.org/>, accessed November 4, 2011.
- [3] *Apache lucene*, <http://lucene.apache.org>, accessed November 10, 2013.
- [4] *Arcgis*, <http://en.wikipedia.org/wiki/ArcGIS>, accessed December 4, 2012.
- [5] *Code page 437*, http://en.wikipedia.org/wiki/Code_page_437, accessed November 4, 2011.
- [6] *English stopwords*, <http://www.ranks.nl/resources/stopwords.html>, accessed November 1, 2013.
- [7] *Fichier: Douglas peucker.png*, http://fr.wikipedia.org/wiki/Fichier:Douglas_Peucker.png, accessed December 6, 2012.
- [8] *The google geocoding api*, <https://developers.google.com/maps/documentation/geocoding/#GeocodingRequests>, accessed December 3, 2012.
- [9] *Google Search Appliance*, Available at http://en.wikipedia.org/wiki/Google_Search_Appliance.
- [10] *Google search appliance help center*, https://gsa1.lib.unb.ca:8443/EnterpriseController/crawl_urls.html, accessed November 7, 2011.
- [11] *Great-circle distance*, http://en.wikipedia.org/wiki/Great-circle_distance, accessed December 16, 2012.
- [12] *Interface servlet*, <http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/Servlet.html>, accessed May 12, 2014.
- [13] *Iso/iec 8859-1*, http://en.wikipedia.org/wiki/ISO/IEC_8859-1, accessed November 12, 2011.
- [14] *Lucene*, <http://en.wikipedia.org/wiki/Lucene>, accessed November 10, 2013.

- [15] *Lucene as a ranking engine.*, <http://www.wortcook.com/pdf/lucene-ranking.pdf>, accessed November 10, 2013.
- [16] *Lucene java doc, class similarity.*, http://lucene.apache.org/java/2_4_1/api/org/apache/lucene/search/Similarity.html, accessed November 10, 2013.
- [17] *Pagerank*, <http://en.wikipedia.org/wiki/PageRank>, accessed May 22, 2011.
- [18] *Shapefile c library*, <http://shapelib.maptools.org/>, accessed December 1, 2012.
- [19] *Snb geographic data & maps products & services*, http://www.snb.ca/gdam-igec/e/2900e_1.asp, accessed December 1, 2012.
- [20] *Spatial search*, <http://wiki.apache.org/solr/SpatialSearch>, accessed December 4, 2011.
- [21] *Specimen Label Data for the Connell Memorial Herbarium*, Available at http://herbarium.biology.unb.ca/fmi/iwp/res/iwp_auth.html.
- [22] *Suffix tree*, http://en.wikipedia.org/wiki/Suffix_tree, accessed June 23, 2011.
- [23] *Trie*, <http://en.wikipedia.org/wiki/Trie>, accessed December 4, 2011.
- [24] *World geodetic system 1984 - background*, <http://www.dqts.net/wgs84.htm>, accessed December 15, 2012.
- [25] David Austin, *How google finds your needle in the web's haystack*, <http://www.ams.org/samplings/feature-column/fcarc-pagerank>, accessed May 22, 2011.
- [26] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The r^* -tree: An efficient and robust access method for points and rectangles*, SIGMOD Conference, 1990, pp. 322–331.
- [27] Jon Louis Bentley, *Multidimensional binary search trees used for associative searching*, Commun. ACM **18** (1975), no. 9, 509–517.
- [28] Monica Bianchini, Marco Gori, and Franco Scarselli, *Inside pagerank*, ACM Trans. Internet Technology **5** (2005), no. 1, 92–128.
- [29] David H. Douglas and Thomas K. Peucker, *Algorithms for the reduction of the number of points required to represent a digitized line or its caricature*, Cartographica: The International Journal for Geographic Information and Geovisualization **10** (1973), no. 2, 112–122.
- [30] Martin Farach, *Optimal suffix tree construction with large alphabets*, FOCS, 1997, pp. 137–143.

- [31] Radu Gruian, *Patricia trie template class*, <http://www.codeproject.com/Articles/9497/Patricia-Trie-Template-Class>, July 2007, accessed January 1, 2013.
- [32] Dan Gusfield, *Algorithms on strings, trees, and sequences - computer science and computational biology*, Cambridge University Press, 1997.
- [33] Antonin Guttman, *R-trees: A dynamic index structure for spatial searching*, SIGMOD Conference, 1984, pp. 47–57.
- [34] Dan Han and Bradford G. Nickerson, *Comparison of text search ranking algorithms*, Tech. report, TR11-209, Faculty of Computer Science, University of New Brunswick, August 2011.
- [35] Jan T. Heuer and Sören Dupke, *Towards a Spatial Search Engine Using Geotags*, GI-Days 2007 - Young Researchers Forum (Florian Probst and Carsten Keßler, eds.), IfGIprints, 2007.
- [36] Robert W. Irving and Lorna Love, *Suffix binary search trees and suffix arrays*, Tech. report, TR-2001-82, Computing Science Department Research Report, University of Glasgow, March 2001.
- [37] ———, *The suffix binary search tree and suffix avl tree*, J. Discrete Algorithms **1** (2003), no. 5-6, 387–408.
- [38] Christopher B. Jones, Alia I. Abdelmoty, David Finch, Gaihua Fu, and Subodh Vaid, *The spirit spatial search engine: Architecture, ontologies and spatial indexing*, GI-Science, 2004, pp. 125–139.
- [39] Udi Manber and Eugene W. Myers, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput. **22** (1993), no. 5, 935–948.
- [40] Edward M. McCreight, *A space-economical suffix tree construction algorithm*, J. ACM **23** (1976), no. 2, 262–272.
- [41] Gahyun Park and Wojciech Szpankowski, *Towards a complete characterization of tries*, Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), SODA '05, Society for Industrial and Applied Mathematics, 2005, pp. 33–42.
- [42] A Rajaraman and J. D. Ullman, *Mining of massive datasets*, (2011), 1–C17.
- [43] Urs Ramer, *An Iterative Procedure for the Polygonal Approximation of Plane Curves*, Computer Graphics and Image Processing **1** (1972), 244–256+.
- [44] H. Samet, *Foundations of multidimensional and metric data structures*, Morgan Kaufmann, 2006.

- [45] Qingxiu Shi, *Data structures for efficient search in high-dimensional spaces*, Ph.D. thesis, University of New Brunswick, 2002.
- [46] Dustin Spicuzza, *A header-only c++ r* tree implementation.*, <http://www.virtualroadside.com/blog/index.php/2008/10/04/r-tree-implementation-for-cpp/>, accessed November 14, 2013.
- [47] Esko Ukkonen, *On-line construction of suffix trees*, *Algorithmica* **14** (1995), no. 3, 249–260.
- [48] Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual Symposium on Switching and Automata Theory, SWAT 1973, Washington, DC, USA, IEEE Computer Society, 1973, pp. 1–11.

Appendix A

Stopwords list

a	about	above	after	again	against
all	am	an	and	any	are
aren't	as	at	be	because	been
before	being	below	between	both	but
by	can't	cannot	could	couldn't	did
didn't	do	does	doesn't	doing	don't
down	during	each	few	for	from
further	had	hadn't	has	hasn't	have
haven't	having	he	he'd	he'll	he's
her	here	here's	hers	herself	him
himself	his	how	how's	i	i'd
i'll	i'm	i've	if	in	into
is	isn't	it	it's	its	itself
let's	me	more	most	mustn't	my
myself	no	nor	not	of	off
on	once	only	or	other	ought
our	ours	ourselves	out	over	own
same	shan't	she	she'd	she'll	she's
should	shouldn't	so	some	such	than
that	that's	the	their	theirs	them
themselves	then	there	there's	these	they
they'd	they'll	they're	they've	this	those
through	to	too	under	until	up
very	was	wasn't	we	we'd	we'll
we're	we've	were	weren't	what	what's
when	when's	where	where's	which	while

who	who's	whom	why	why's	with
won't	would	wouldn't	you	you'd	you'll
you're	you've	your	yours	yourself	yourselves

Vita

Candidate's full name: Dan(Amber) Han

University attended:

September 2010 - May 2014
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

September 2006 - June 2010
Bachelor of Engineering
Shool of Information and Communication Engineering
Beijing University of Post and Telecommunications
Beijing, China

Technical Report:

Dan Han and Bradford G. Nickerson, "Comparison of Text Search Ranking Algorithms", UNB Faculty of Computer Science, TR11-209, August 2011, Fredericton, N.B., Canada.

Poster:

Dan Han and Bradford G. Nickerson, "Poster: Text Search with Spatial Constraints", 10th UNB Computer Science Research Exposition, April 12, 2013, Fredericton, N.B., Canada.

Publications:

Conference Presentations: