IMPROVED DATA STRUCTURES FOR DYNAMIC AND MASSIVE DATA

A Thesis

Submitted in fulfilment of the requirement of the degree of

Doctor of Philosophy

by

Sunita

Reg. No. 951003005

Dr. Deepak Garg, Supervisor

Professor & Head, Computer Science Engineering, Bennett University. Noida

&

Dr. Maninder Singh, Administrative Supervisor

Professor & Head, CSED, Thapar Institute of Engineering & Technology, Patiala

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY, PATIALA

February, 2018

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY

Computer Science and Engineering Department
Thapar Institute of Engineering &Technology, Patiala

CERTIFICATE

This is to certify that the thesis entitled "**Improved Data Structures for Dynamic and Massive Data**" being submitted by **Sunita (Reg. No. 951003005)** in fulfilment of the requirement for the award of the degree of **Doctor of Philosophy** is candidate's own work done under my supervision.

The thesis has not been submitted in part or in full to any other Institute/University for the award of any other degree.

Dr. Deepak Garg,
Supervisor
Professor & Head,
Computer Science Engineering,
Bennett University, Noida

Dr. Maninder Singh,
Administrative Supervisor
Professor & Head, CSED,
Thapar Institute of Engineering
& Technology, Patiala

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY

Computer Science and Engineering Department
Thapar Institute of Engineering &Technology, Patiala

CANDIDATE'S DECLARATION

I hereby certify that the work which presented in this thesis entitled "**Improved Data Structures for Dynamic and Massive Data**" is being submitted to the Computer Science and Engineering Department of the University in fulfilment of the requirement for the award of the degree of **Doctor of Philosophy**. The work submitted in this thesis is my own work done under the supervision of **Dr. Deepak Garg, Professor & Head, Computer Science Engineering, Bennett University, Noida.**

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other Institute/University.

Sunita

Sunita

Reg. No. 951003005

ACKNOWLEDGEMENT

I thank God, the almighty for giving me strength, wisdom and peace.

I consider myself privileged to have Dr. Deepak Garg as my supervisor. I thank him for constant guidance, motivation and encouragement throughout this research work. His comments and suggestions made all the difference. With sincerity and humility I thank him for especially believing in me, his sound advice over the years and for devoting his valuable time to this work without which this work would not have been possible in time.

I would like to thank Dr. Maninder Singh Professor and Head, Computer Science and Engineering Department for the administrative support and suggestions at various occasions. I would like to thank Dr. Seema Bawa, Professor for her important suggestions time to time. I am extremely grateful to the members of Doctoral Committee Dr. Maninder Singh (Chairman), Dr. Anil Kumar Verma, Dr. V. P. Singh and Dr. Rajesh Sharma for their encouragement, insightful comments and valuable suggestions. I owe my thanks to the faculty members of the department for their suggestions.

I would like to extend my gratitude to all those reviewers as well as editors who gave their valuable comments that really enhanced the quality of my research work. I thank my parents, brother, sisters and relatives for their support, encouragement and understanding during the research work. They always pray to God to make me successful in life. A special thanks to my brother Mr. Harvinder Bajaj for all his help and support. I wish to dedicate this thesis to my husband Dr. Anshu Parashar who always supported me, encouraged me in my hard times. I would like to extend my love to my son Ritvik for his smile and understanding**.** Many thanks are also due to my colleagues and friends for their kind cooperation. Many apologies are extended to those who have had to put up with me during this process.

Sunita

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ST	Suffix Tree
SA	Suffix Array
SACA	Suffix Array Construction Algorithms
LCP	Longest Common Prefix
BWT	Burrows Wheeler Transform
EM	External Memory
SSSP	Single Source Shortest Path
DSSSP	Dynamic Single Source Shortest Path
RPQ	Retroactive Priority Queue
PST	Priority Search Tree
R&R	Ramalingam and Reps

LIST OF PUBLICATIONS

1 **"A Retroactive approach for dynamic shortest path problem",** Journal of National Academy Science Letters (in Press, First Online: 06 October 2018), published by Springer and indexed by SCI.

Cite this article as: *Sunita & Garg, D. Natl. Acad. Sci. Lett. (2018). https://doi.org/10.1007/s40009-018-0674-6*

2 "**Extended suffix array construction using lyndon factors**" SADHANA, The Indian Academy of Sciences Journal (43:133, 2018), published by Springer and indexed by SCI.

Cite this article as: *Sunita & Garg, D. Sādhanā (2018) 43:133. https://doi.org/10.1007/s12046-018-0832-z*

3 **"Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue"** Journal of King Saud University - Computer and Information Sciences (in Press, First online: 6 March 2018) , published by Elsevier.

Cite this article as: *Sunita, Garg, D. Dynamizing Dijkstra: A solution to dynamic shortest path problem through retroactive priority queue, Journal of King Saud University – Computer and Information Sciences (2018). https://doi.org/10.1016/j.jksuci.2018.03.003*

ABSTRACT

For many real world problems solved using various algorithmic techniques, traditional algorithms are theoretically optimal but as soon as we move on to real world data, based on the nature of data the algorithm efficiency degrades drastically. For massive data which is stored in external memory, algorithms designed for internal memory which are otherwise efficient will fail due to I/O bottleneck. Moreover in most real world scenarios data is dynamic i.e. changes with time. Novel and very different design techniques are needed to manage this dynamic and massive real world data. This thesis contributes to the vast research of such results. In this era of information explosion, the flood of data emerging on daily basis has made data handling and management a very crucial task. In this study, we are looking in to this data from two different perspectives: massive data and dynamic data.

For dynamic data, we are considering the implementation of Retroactive data structures specific to some application and then using the proposed implementation to analyse the performance. Dynamic shortest path algorithms modify the existing shortest paths by considering the changes in the underlying graph configuration. Basic approach used in literature to solve this shortest path problem is to dynamize the Dijkstra algorithm: widely used algorithm for the static case. Different solutions already exist for the problem but we are using the concept of retroactive data structures for incorporating the dynamic changes into the solution. In this work algorithms for the different operations of retroactive priority queue are given. Also an algorithm has been proposed for the solution of dynamic shortest path problem as dynamic Dijkstra which uses the retroactive priority queue for its dynamization. The proposed approach is a suitable solution for the dynamic shortest path problem.

For massive data, we are extending the novel approach of incremental suffix array construction using Lyndon factorization to the construction of extended suffix array.

Extended suffix array is the suffix array along with the corresponding longest common prefix (LCP) array. Main motive behind the incremental and simultaneous construction of suffix array and LCP array is that both involve calculating the order information by considering the common prefixes of the suffixes. Using Lyndon factorization for suffix array construction reduces the merging overhead in terms of time as well as space because to extend the sorting from local suffixes of Lyndon factors to the global suffixes in merged Lyndon factors need no additional information. Also, the two sorted orders coincide thus making the merging of Lyndon factors a simple merging of two sorted lists of suffixes. Also, incremental LCP construction simultaneously saves a lot of computation and hence time. The proposed approach has quadratic run time and the disk working space requirement is O(n). Experiments also show the performance gain of our approach in terms of time over the existing method of incremental construction.

CHAPTER-1

Introduction

The advancement in computer and technology has flooded us with data. It has been estimated that the amount of information is almost doubled every 20 months [6, 67]. So, it becomes vital to store this large amount of data (massive data) in such a way that the storage as well as searching for data for relevant information can be done efficiently. Also, this data from widely varying sources like biological experiments, Internet routing information, sensor data, data available through search engines and audio/video devices require new methods for managing data [66] as it is dynamic in nature(dynamic data). Almost all of this data needs to be interpreted in one way or the other like interpretation of biological data, online cross community research group data [55], Dynamic analysis of software engineering data [68] etc. So, managing such a massive and dynamic data needs to consider all the issues like storage, searching and retrieval along with maintaining the dynamism of the data. In the initial years of its development, maintaining dynamism involved updating the existing approaches of data storage and retrieval explicitly to accommodate the dynamic changes in the data. But in the year 2004, with the introduction of a new data structuring paradigm called as retroactive data structures [30], it has become quite easy to develop dynamic algorithms from the existing static ones.

To efficiently search/access the data, the whole searching procedure is divided into two different phases: indexing and searching. The indexing phase scans the whole data and creates a list of all possible items to be searched. Then the search phase simply uses the index created in the indexing phase to search for the required data. So, index is once created and thereafter used for all the search operations reducing the total search time significantly. According to the type of data and the type of search operations to be implemented, there are two main index types: Inverted-indexes and Full-text indexes. Inverted indexes are keyword or term-based indexes

while Full-text indexes are substring based indexes which help to search for any substring of the given data. In the thesis we are focussing on the second type of indexes i.e. Full-text indexes.

Further in this chapter, an introduction of the basic concepts is given which will be used throughout this thesis.

1.1 Retroactive Data Structures

Retroactive data structures are the data structures that along with maintaining the data also maintain the time for the data. These timeline data structures help to maintain the dynamic data. Dynamic data refers to the data in which updating and querying can be at any time i.e. present or past. The main idea behind the retroactive data structures is that data can be modified at any point of time i.e. past as well as present and the effects of that modification on the present state of data are propagated accordingly [31]. So, Retroactive data structures store data in the data structure in terms of sequence of operations along with the corresponding time value. A user is able to go back in time, perform an operation in the past and then look at the effects of that change on the current state of the data. The time line thus maintained in the data structure is linear. Operations that are allowed on a retroactive data structures are:

Insert(t, x) : Retroactively insert operation x at time t

Delete(t) : Retroactively delete operation at time t

Query(t, x) : Do query x at time t

The main difficulty in implementing the retroactivity is that simply adding time as a dimension in the data structure is not sufficient as the operations performed on the data structures are interdependent, so that when any changes are done in the past these need to be accommodated in the present state of the data structure also. Thus, traditional high dimensional data structures are not suitable for the implementation of most of the retroactive data structures. These need to be adapted in some way (according to the need) to be used for implementing the

retroactive data structures. Also, to create efficient retroactive data structures some techniques must be introduced so that storing every state of the structure explicitly is not required like Chan's convex hull algorithm [22], which dynamically creates the hull but does not explicitly store it.

1.2 Models of Retroactive Data Structures

There are two different models of retroactive data structures: Partial Retroactive and fully Retroactive Data Structure. The state of a retroactive data structure is defined in terms of a sequence of queries and updates made on the data structure over time. The sequence of updates applied on the data structure over a period of time t (starting at t_1 and ending at t_n) be represented as $U = (u_{t1}, u_{t2}, u_{t3}, \ldots, u_{tn})$, where u_{ti} be the update performed on the data structure at time t_i , and $t_1<t_2<t_3\ldots\ldots<t_n$.

1.2.1 Partial Retroactive Data Structures

These are the data structures which allow query operations to be applied only to the current state of the data structure. The update operations can be applied at any point of time i.e. present as well as past. So, formally, partial retroactive data structures are defined [31] as those which in addition to allowing update and query operations to be performed in the present time of the data structure, also allow performing insertion and deletion of the update operations at past times as well. The insertion and deletion operations of the data structure will be defined w.r.t. time parameter while query operation is defined without time, as we can query only at the present time. So, the retroactive operations for this variant of the data structure are defined as:

- Insert (t, x): inserts into update sequence U, a new update operation x at time t, where operation x can be either insert operation or delete operation and can be represented as:
    - Insert(t, insert(v)): inserts value v at time t, if t is present time then it simply adds a new update operation to update sequence U, else it adds the update

operation in past and propagates the change in the data structure till the present time.

- Insert(t, delete(x)): deletes the value v at time t, if t is present time then it simply adds a new update operation to update sequence U, else it adds the update operation in past and modifies the current state of the data structure according to the operation.

- Delete (t): deletes the update operation u_t performed at time t from the sequence of updates U and also modifies the current state of the data structure accordingly.

These retroactive operations of the data structure possibly affect almost all the operations that have been performed between the time of retroactive operation and the present time. In general, accommodating a change in the past in the retroactive data structure leads to a completely new state of the data structure. The efficient implementation of these retroactive data structures requires representing all these operation sequences and corresponding information implicitly.

1.2.2 Fully Retroactive Data Structures

These are the data structures which allow both the update as well as query operations to be applied at any point of time: past or present. So, these data structures give the user flexibility to modify the past as well as observe it. The update operations for both the partial and fully retroactive data structures are same. But the query operation of fully retroactive data structures is also defined with a time parameter and is represented as Query(t,"search(x)"), it searches for element x in the state of data structure at time t and returns yes if element x was there in the data structure on or before time t else it returns no. Complexity analysis of retroactive data structures [30] is done in terms of the maximum size of the data structure at any time say *n*, the number of update operations performed in the structure (performed in past or present time) say *m*, the

number of update operations performed before which the retroactive operation is to be performed say *r* (i.e., $t_{m-r} < t \leq t_{m-r+1}$).

1.2.3 Non-oblivious Retroactive Data Structures

A slight modification in above models of retroactive data structures has been proposed by *Acar et al.* [3] known as non-oblivious retroactive (active) data structures. In the retroactive data structures, the changes arising in the data structure due to some modification in the past are automatically accommodated in the present but in non-oblivious retroactivity, the changes are propagated one by one. The user is informed of the first operation that is affected due to the operation performed in the past i.e. an operation whose return value has changed due to the operation performed in the past. In this way the user can easily identify the affected operation and he can accordingly perform further modifications which will make the Update sequence and hence the state of data structure consistent. Instead of automatically accommodating the changes due to current retroactive operation i.e. invoking or revoking of an operation to the present state of the data structure, the user is informed of the first operation that becomes inconsistent, i.e., an operation whose value has changed from before. This gives the user flexibility to choose the revisions to be performed on the Update sequence to bring the data structure back to the consistent state. For example, if some wrong information is logged into a database erroneously, then the operation that entered that wrong information into the database can be revoked by applying a retroactive operation. As a result of the retroactive operation, the user will be notified of the first operation in the Update sequence that becomes inconsistent due to this retroactive operation. Thus instead of predefining the corrective actions within the retroactive operation, the user has the flexibility to take corrective actions as per the requirements. Based on this idea propagating the changes one by one, these data structures can be used effectively to dynamize static algorithms.

1.3 Applications of Retroactive Data Structures

Most of the real world data is dynamic in nature like data from various automated devices, sensor data, transaction log data of various database systems e.g. banking transactional data, software engineering data, version control data etc. For almost all the types of dynamic data, the

data needs to be manipulated at any point of time past or present to maintain its consistency and integrity. In all these cases, retroactive data structures are very useful for maintaining the consistency even in the case of manipulation of data in the past. One more area of applicability of these data structures is the dynamization of the static algorithms. Use of retroactive data structures for dynamization of static algorithms is more efficient as compared to their implementations which use simple dynamic data structures for holding dynamic data.

1.4 Suffix Tree

It is the Patricia trie data structure [95] which is used to index all the suffixes of a given string. It is an ordered data structure as all the leaves are ordered lexicographically. A path from the root to leaf *i* corresponds to the i^{th} suffix of the string. A suffix tree (ST) for a string of length *n* will have *n* leaf nodes, numbered from *1* to *n*. The suffix tree, like the trie data structure, enables quick string as well as prefix searching. Time complexity for searching of a substring in this data structure is linear i.e. $O(m)$ where *m* is the substring length. Space required for the data structure is $\theta(nlogn)$ (where *n* is the size of original string) which is much larger than that required for the original string. These are the important data structures for string processing applications like Computational Biology, Information Retrieval etc. Let text *T* be "ababaabcd$". Suffix tree for the example text is shown in figure 1.1.

1.5 Suffix Arrays

Suffix arrays (SA) were introduced by *Manber & Myers* [91] to overcome the huge space requirement of suffix trees: the full-text indexes. The same data structure was also discovered independently by *Gaston Gonnet* in 1987 with a different name *PAT array* [59].

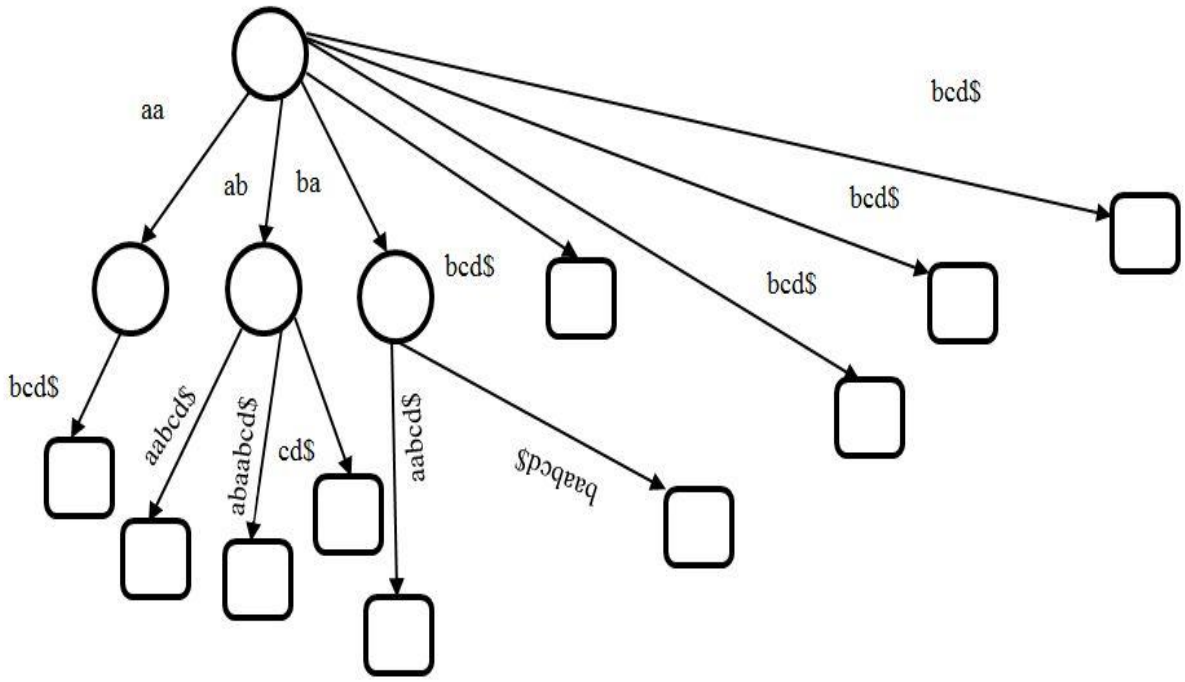

Figure 1.1 Suffix Tree for text T = "ababaabcd$"

Suffix array is defined as a sorted array of all the suffixes of the given data. Suffix array is the basis of almost all full-text index structures. It is created for a given text *T* by assuming that the whole text is a single string. All the substrings of text *T* starting at index *i* and extending up to the last index of *T* are said to be the suffixes of *T*. Let *T* be a text of length *n* represented as *T[0...n-1]* then a suffix of text *T* is given as $Suf_i = T[i.....n-1]$. Also, a text of length *n* will have *n* suffixes. So, the size of corresponding suffix array is also *n*. To simplify the construction of suffix array for this text, we append the text by a character '$' which is smallest of all the characters of the text and also it does not exist anywhere in the text. The text to be indexed (*T*) is shown in table 1.1, all the suffixes of the text *T* are shown in table 1.2. Table 1.3 shows all the suffixes in sorted order.

Table 1.1 Text *T* "ababaabcd$" to be indexed

i	0	1	2	3	4	5	6	7	8	9
T[i]	a	b	A	b	a	a	b	c	d	$

Table 1.2 Suffixes of text *T*

I	Suffix
0	ababaabcd$
1	babaabcd$
2	abaabcd$
3	baabcd$
4	aabcd$
5	abcd$
6	bcd$
7	cd$
8	d$
9	$

Table 1.3 Sorted Suffixes of text *T*

I	Suffix
9	$
4	aabcd$
2	abaabcd$
0	ababaabcd$
5	abcd$
3	baabcd$
1	babaabcd$
6	bcd$
7	cd$
8	d$

Table 1.4 Suffix Array (SA) of text *T*

i	0	1	2	3	4	5	6	7	8	9
SA[i]	9	4	2	0	5	3	1	6	7	8

Originally, suffix array was introduced as an index structure for substring searches as suffix array stores all the suffixes of a given text so, it can perform substring searches very quickly. Apart from being used as full-text index, this data structure has applications in many other fields like string matching [60, 92], data compression algorithms [19, 25, 46] and within the fields of computational biology and bioinformatics like genome analysis [1, 2, 50, 69]. Some work has also been done for using these indexes in information retrieval [28, 110].

1.6 Important Concepts

1.6.1 Space Complexity

As shown in table 1.4, suffix array for a text of size *n* stores *n* integers. Suppose an integer takes *4* bytes in storage, suffix array requires *4n* bytes in total which is much less than the memory requirements of the corresponding suffix tree. However, in certain applications where the text is over a large alphabet, the space requirement of suffix array is much greater than the size of the corresponding text. So, even with these space requirements, suffix arrays were not the suitable data structures for massive data, leading to a new trend involving indexing as well as compression. Many authors have worked on this new trend designing various compressed suffix arrays and Burrows-Wheeler Transform (BWT) –based compressed indexes. But SA or BWT have been the basic components in all these variants. A very good review of all these compressed data structures has been given by *Navarro and Makinen* [102]. All these data structures besides being compressed are also self-indexes i.e. there is no need to store the indexed text as it can be reconstructed efficiently from the index itself.

1.6.2 Construction Algorithms

The first step for any index structure is its construction. Although construction of suffix array is linear theoretically but the same is not true in practice. All the algorithms use some computing resources out of which time and memory space are of prime concern. Memory space is represented in terms of working space which is the amount of memory required by the construction algorithm and is usually much more than the memory requirement of the index itself.

The major step during the SA construction is the sorting of the suffixes. So, the simplest approach for the construction of SA is to use any of the existing sorting algorithms for sorting the suffixes. The lower bound of any comparison based sorting algorithm is $\Omega(n \log n)$, which is the minimum number of suffix comparisons required to do the sorting. Also a suffix of average size n takes $O(n)$ time, so the complexity of this approach is $O(n^2 \log n)$. But as we know all the suffixes to be sorted are related to each other and this fact led to many algorithms with lesser time complexity as compared to the above approach.

In the initial stages of its development, SA construction was done by first constructing the corresponding suffix tree [69]. After some time many authors came up with the direct suffix array construction algorithms (SACA). A very extensive survey of existing suffix array construction algorithms had been done by *Puglisi* et al. [112]. They have given the following classification for the SA construction algorithms:

- **Internal Memory Algorithms:**

  In which the whole text during the construction is in the internal memory of the system.

- **External Memory Algorithms:**

  In which the text during the construction is in the external memory of the system. A portion of the text is stored in internal memory and using that text, partial SA is constructed and the procedure is repeated for the remaining text. All the partial SA's are merged suitably to get the SA for the whole text.

- **Lightweight Algorithms:**

  Construction algorithms which optimize the working space in internal memory [94, 105].

- **Constant Alphabet Algorithms:**

  Algorithms which are used for the text whose alphabet size is bounded by a constant.

- **Integer Alphabet Algorithms:**

  Algorithms where characters of the text are integers in a range depending on *n* size of the text [18].

- **General Alphabet Algorithms:**

  Algorithms which are used for text where only character comparisons are allowed [18].

Also, some efficiency parameters for SA construction algorithms have been given by *Puglisi et al.* [112]:

- Optimal asymptotic complexity
- lightweight in space, i.e. working memory requirements are very less (excluding the memory requires for the text and the suffix array itself)
- optimal runtime

In the recent years, research in suffix array construction algorithms has moved to Dynamic SA Construction Algorithms. Authors *Salson et al.* [120] have proposed an algorithm for updating the suffix array of a text that changes with time. The worst case time complexity of the algorithm is *O(nlogn).*

1.6.2.1 Internal Memory Suffix Array Construction Algorithms

Suffix array construction algorithms in which the whole text during the construction is in the internal memory of the system are known as internal memory SACA's. All the suffixes of the text to be sorted are related to each other and authors have utilized this fact to make the construction algorithms efficient. Almost all the internal memory SA constructions algorithms are based on one of the mentioned techniques: Prefix doubling [86, 91, 92, 117], Recursive [76, 82, 84], induced copying [18, 74, 94, 123, 125].

1.6.2.2 External Memory Suffix Array Construction Algorithms

As the size of data to be indexed grows, it makes the existing internal memory construction impossible as the working space required for the construction is far beyond the memory capacity of the system. So, the attention moved towards devising external memory SA construction algorithms. Most of the external memory SA construction algorithms are based on incremental approach. In the incremental approach, the whole text to be indexed is divided into blocks. The text is stored in external memory. A block of text is transferred from external memory to internal memory and its partial SA is constructed. Now, that block of text is removed and next block is transferred from external memory. SA for that text block is constructed and merged with the previous block's SA. And the procedure is repeated for all the remaining blocks. This incremental approach significantly reduces the working space. But the merging step of the approach is a big hindrance to the scalability of these algorithms. Many authors have worked on reducing the merging time [13, 33, 77]. A completely different perspective was introduced by the work of *Bonomo et al.* in 2013 [14], who have used Lyndon Factorization of text for suffix array construction. Their approach significantly reduces the merging time.

Some more important concepts related to efficient use of SA is the Longest Common Prefix (LCP) Array and Burrows-Wheeler Transform (BWT).

1.6.3 Longest Common Prefix Array

An array representing the longest common prefixes of all pairs of adjacent suffixes in SA is called as longest common prefix (LCP) array. LCP array for text T is shown in table 1.5. Using LCP array as an auxiliary data structure with the SA speeds up querying into the text. LCP array is an array of size n whose entries are defined as:

$$LCP[i] = -1 \quad \text{if } i=0$$

$$lcp(suf_{SA[i-1]}, suf_{SA[i]}) \quad 0< i <= n$$

For the example text given in section 1.2.1, the LCP array will be:

Table 1.5 Longest Common Prefix Array for text *T*

i	0	1	2	3	4	5	6	7	8	9
LCP[i]	-1	0	1	3	2	0	2	1	0	0

Querying the text using only SA takes *O(m log n)* time where *m* is the size of the query text. This time can be reduced to *O(m+log n)* by using the LCP array [92]. Suffix array saves space by discarding the structure information of the suffix tree and keeping only the order information. So, applications where along with order information topological information is also required suffix array cannot efficiently replace the suffix tree. This problem can be remedied by having some additional information- Longest Common Prefix (LCP) with the suffix array. In a suffix tree, each internal node represents a group of suffixes having the same LCP. So, LCP encodes some of the topological information of suffix tree, thus it helps to effectively replace a suffix tree with suffix array and LCP even in the applications which need some topological information of the suffix tree.

1.6.4 Burrows-Wheeler Transform

Burrows-Wheeler Transform (BWT) was introduced in the year 1994 [19] as a way to compress large texts without losing their information content. It is basically a reversible text permutation which is highly compressible. BWT transforms a text *T* of *n* characters by first getting all the

cyclic shifts of *T*, then sorting these cyclic shifts lexicographically, and finally extracting the last character of each of the rotations in sorted order (also called array L). The i^{th} character of BWT is the last character of the i^{th} sorted cyclic rotation. The transform thus calculated is easily reversible if we also have the first character of every cyclic rotation taken in sorted order, represented as array F. Sorted cyclic rotations of a text appended with the smallest character (not existing in the text) also give the sorted order of the suffixes i.e. suffix array of that text. So, the first column (F) of the sorted cyclic shifts represents the first character of the sorted suffixes and the last column (L) corresponds to BWT. There is a function LF () which gives the correspondence between the characters of the columns L and F. This LF () function helps to reconstruct the text from its BWT. For the example text given in section 1.2.1, sorted cyclic shifts and the BWT array are shown in table 1.6 and table 1.7 respectively.

Table 1.6 Sorted Cyclic Shift of text *T*

Suffix	**i**
$ababaabcd	9
aabcd$abab	4
abaabcd$ab	2
ababaabcd$	0
abcd$ababa	5
baabcd$aba	3
babaabcd$a	1
bcd$ababaa	6
cd$ababaab	7
d$ababaabc	8

Since its inception, BWT has been the most widely used data compression method in one form or the other and with the introduction of FM-index in the year 2000 [44], this has become the key player in a new branch of indexing structures known as the compressed indexes [64].

Table 1.7 Burrows-Wheeler Transform for text *T*

i	0	1	2	3	4	5	6	7	8	9
BWT[i]	d	b	b	$	a	a	a	a	b	c

Authors for the first time have exploited the relationship between compression and indexing of data collections. Constructing indexes for the text using Burrows-Wheeler transform helps to index the text without much affecting its usage (not increasing the search and query time of the index) and also makes it a self-index.

1.7 Motivation and Problem Statement

With the ever-increasing size of the data in the present scenario, research has directed more towards full-text index structures in external memory. The full-text indexes for internal memory extensively use compression techniques to have efficient implementation. Compression is not a necessity in case of external memory full-text indexes due to availability of sufficient and cheap external memory. Moreover, accessing internal memory is much faster than external memory access, so how the data structure is stored in memory and how it is accessed from the memory directly impacts the complexity of corresponding operations in external memory as compared to internal memory. Another major bottleneck in the use of full-text index is the initial construction phase in which the most basic step is suffix array construction. Many construction algorithms for suffix array require more space than the final index itself. So even if the final index (suffix array) can fit in internal memory, it's not always possible to construct it in internal memory. So, many authors have worked on finding space-efficient algorithms by using compressed data structures or by finding a way to tradeoff runtime or by using external memory (disk), or some combination of any of these methods. Also, in case of massive data, the working space required by the internal memory construction algorithms is much higher than the available working memory space as well as memory required for the final index structure. So for massive data, external memory construction algorithms were devised. Related to external memory index construction

algorithms, the major bottlenecks are limited working space available as internal memory, cache misses and a large number of random accesses to secondary memory and research in this direction is directed towards resolving these issues.

Apart from considering the size of data present today, there is one more perspective that is to be looked into: we have data arising from various sources which are dynamic in nature. Data is said to be dynamic when the data to be stored changes with time and hence the ultimate user of that data wants privileges to view and/or modify it at various points in the past, in addition to its present state. The data structures that are used to store such dynamic data are known as temporal data structures. There are two primary models of temporal data structures. First is *persistent data structuring* paradigm, in which going back in time and making changes there creates a new branch of the data structure that is different from the original one. Second is *Retroactive Data Structures,* in which going back in time and making changes their returns the effects of that change in the present state of the data structure. So, retroactive data structures efficiently manage the dynamic data as well as help to modify static algorithm into its dynamic counterpart by allowing operation sequence to be defined with respect to time parameter.

Based on the above observation, we have directed our focus towards finding how to efficiently implement retroactive data structures for specific applications. The application that we have considered for our purpose is the Dynamic single source shortest path (DSSSP) problem.

Also, we have considered the issue of reducing the run-time of external memory construction by handling the major bottlenecks of the approach like cache misses and a high number of random accesses to secondary memory. Based on the above facts following research objectives have been formulated:

1.8 Research Objectives

- Implementing Retroactive Priority queue for the solution of DSSSP Problem.
- Efficient External memory construction of full-text index for massive data.
- Analysis of the proposed approaches and their suitability for the respective data.

1.9 Thesis Layout

This thesis consists of five chapters.

It begins with **Chapter 1** that describes fundamental concepts of Retroactive data structures, different models of these data structures, Suffix array and the important concepts related to suffix array like Construction algorithms, Longest common prefix array and Burrows-Wheeler Transform etc. The motivation for the work is given along with the research objectives. Subsequent chapters of the thesis are organized in the following manner:

Chapter 2 reviews existing work relevant to the research objectives. Literature has been reviewed focussing on two different aspects of data i.e dynamic data and massive data. Technological advances made in the field of managing and handling dynamic data have been reviewed then narrowing the review to the domain of retroactive data structures. Moreover, for selecting a suitable application for the performance evaluation of proposed retroactive solutions, we have also reviewed a class of graph problems as graphs are most commonly used to model the dynamic data arising in varying domains. Then moving on to the domain of massive data, we have reviewed the literature corresponding to full-text indexes which are used to efficiently store and retrieve the massive data. In the end, we have reviewed different types of construction algorithms for one of the most basic indexing structure i.e. suffix array.

Chapter 3 presents the proposed approach for implementing a retroactive priority queue using priority search trees. Then the proposed definition of retroactive priority queue has been used for the solution of Dynamic Single Source Shortest Path problem. The chapter, first of all, gives an overview of the shortest path problem and some of its variants. Then the idea of solving the Dynamic single source shortest path problem using retroactive data structures has been given,

starting with problem specification and terminology used and then briefing the idea of implementing the retroactive priority queue using priority search trees. The proposed approach for the solution of the problem along with the algorithm has been given. The proposed approach has been analyzed both in terms of correctness and complexity. Experimental evaluation of the approach has also been done comparing it with one of the existing approaches for the problem under consideration.

Chapter 4 is an extension of the work presented in chapter 3. On the lines of chapter 3, a definition of retroactive priority using height balanced search trees has been given. Then the definition has been used for the dynamization of well-known Dijkstra algorithm. Dynamized algorithm has been presented and analysed in terms of complexity.

Chapter 5 presents a method that constructs Extended Suffix Array that is the basic component of almost all the full-text indexes. Extended suffix array has been used as it can be used to index massive data effectively. Lyndon factorization of the text has been used for the construction. This approach is based on the combinatorial relation between the construction of suffix array and Burrows-Wheeler Transform established based on the Lyndon factorization of text. First of all an overview of extended suffix array, its construction methods and Lyndon factorization is given. After that, we move on to the approach used for constructing extended suffix array using Lyndon factorization. Then, the proposed approach for extended suffix array construction has been explained and analysed/compared experimentally with one of the existing approaches for suffix array construction.

Finally, **Chapter 6** concludes the research work presented in this thesis, highlighting the major contributions and the future research directions in this area.

CHAPTER-2

Literature Review

In this era of information explosion, the flood of data emerging on a day to day basis has made data handling and management a very active area of research. In this study, we are looking into this data from two different perspectives: massive data and dynamic data. No doubt the two types of data are interrelated, but for the purpose of our study, we are considering them as two different research directions. We have studied the literature for them separately. In section 2.1, various techniques for handling dynamic data have been reviewed. One such technique referred to as retroactive data structures in literature has been reviewed in detail as this is one of the research objectives of this thesis. In section 2.3, we review the related work corresponding to the application we are considering for analysing our work based on retroactive data structures. In section 2.4, we start with the most prominent technique related to storage and effective retrieval of the data known as full-text indexes. After an introductory review of basic techniques, we move on to the study of the construction of these indexes which is also the motivation of our proposed approach for handling massive data. Section 2.5 reviews various external memory construction algorithms for full-text indexes.

2.1 Dynamic Data

Dynamic data is data that changes in time, such as the land-use patterns of New Delhi or weather report of Chandigarh on May 20, 2016. So, the dynamic data is completely specified by value as well as the time at which this data value exists. Dynamic data is collected to analyse weather reports, monitor traffic conditions, study business as well as research trends, and so on. This data comes from many sources ranging from manual data entry to data collected using observational sensor networks [75, 126], opportunistic networks [87] or generated from simulation models [132]. Different techniques are required to handle the dynamic data arising from such widely

varying sources like: dynamic data in case of opportunistic networks is the details of its current and potential helpers and this data is analysed periodically by the network for its possible expansion or contraction [87, 88]. Programs that solve problems related to this dynamic data also need to be dynamic. Making the program dynamic is difficult due to the following reasons: First of all, information may not be instantly available. Also when it is available it may refer to some condition in the past. As delays are random, information may reach out of sequence. So, we need a way to update the past also and propagate it's after effects to the present. Many researchers have given different ways to achieve dynamism in algorithms.

Work on managing dynamic data has two different directions. One way is to update the underlying data structure to handle the dynamic data and another one is to modify the corresponding algorithm to manage it. Early work on updating the underlying data structure started with making data structures persistent. There are some applications, such as computational geometry [23, 26, 39, 106, 107, 121, 130], text and file editing [115] and implementation of very high level programming languages which require that with changing data, all the values i.e. previous value as well as modified value should be maintained. The data structures devised to handle such data were called persistent data structures. There are two variants of persistent data structures [48]: partially persistent if modification is supported only in the latest version but access can be performed on all versions and fully persistent if modification and updation are supported for all versions of the data structure.

Different authors have worked on devising partially or fully persistent forms of various data structures, like stacks [99], queues [73], search trees [98, 100, 107, 115, 130], and related structures [23, 26, 39]. But most of this work uses no general technique for making the data structure persistent. *Overmars* [106] systematically arrived at three different ways to achieve partial persistence using a generalized method.

Another class of work is to make the underlying algorithm dynamic. This approach of modifying the existing static algorithms to obtain the corresponding dynamic algorithm is known

as dynamization. A class of search problems was dynamized by *Bentley and Saxe* [11] by using an incremental approach. In this approach, the whole search space can be searched for information but if update operation is to be performed, it is performed only on a part of the search space. This approach trades off search-time for update-time. *Overmars* [108, 109] dynamized a class of C(n)-order decomposable problems by using the incremental approach of *Bentley and Saxe* [11]. *Mulmuley* and independently *Schwarzkopf* proposed to maintain the history of operations and dependences among them [97, 124] using a dynamic shuffling approach. The approach allows insertions/deletions to be applied at any point of time in "history". Updations done to the history are also propagated forward in time. But the limitation of the approach is that it is problem specific as maintaining history, the dependences, and propagation are to be defined according to the problem to be dynamized.

Later based on the same idea *Basch, Guibas, and Herschberger* [9] devised a new class of data structures called Kinetic Data Structures. These data structures can be used for continuously moving data rather than the data that is changing at discrete intervals of time. These data structures can also support discrete modifications like insertion/deletion with little extra effort. This approach is also problem specific. None of these approaches address the issue of automating the process of dynamization. Later this idea of automatic dynamization was introduced by *Demers et al.* [34]. They proposed to maintain the dependence graph of a computation so that whenever a change arises it can be easily accommodated by using the dependence graph. The major drawback of the approach is that to use this approach the dependence graph should be static (irrespective of the input). *Pugh and Teitelbaum* [111], used memoization for dynamization of some divide-and-conquer algorithms. The approach is similar to the C(n)-decomposable framework of *Overmars* [108], with the difference that it can be applied to a class of problems instead of a specific problem and also it works automatically. Some of the above techniques can be used for the solution of various problems based on dynamic data.

In 2004, the idea of retroactive data structures was introduced by *Demaine el al.* in 2004 [30]. These data structures help to efficiently handle the dynamic data using data structuring paradigm. There are so many applications of these data structures ranging from handling security breaches to dynamization of static algorithms. But creating these data structures is somewhat complicated because representing dynamic data simply in terms of time is not sufficient. Complicated dependencies exist among data at different points of time, so there is no general method which can be used to modify a data structure into its (efficient) retroactive counterpart. Authors have shown how to construct retroactive data structures by storing the sequence of update operations with respect to a time parameter. They have also given some general transformations to make data structures partially or fully retroactive for some problems. Finally, they gave definitions for various retroactive data structures like queues, double-ended queues, priority queues, union-find etc.

Tangwongsan in *2006* [131] proposed a novel data-structuring paradigm, called active data structures. Active data structures allow operations to be performed on the data structure at any point of time – present or past. Unlike most time machines, where changes to the past are accommodated and propagated automatically by magic, active data structures systematically communicate with the outside world, prompting them to take appropriate actions. Author has given an efficient implementation of three active data structures: (monotone) priority queue, dictionary, and compare-and-swap. Both retroactive data structures and active data structures are similar in the sense that these allow operations to be performed in the present as well as the past. The main difference between the two is that in a retroactive data structure, changes are accommodated automatically without the user intervention but in active data structures, these are reflected to the users also.

In 2007, *Acar et al.* [3] proposed another model of retroactive data structures and called it as non-oblivious retroactivity. This model stores both the update operations as well as query operations (with respect to a time parameter) as part of the operation sequence. When any

operation is invoked or revoked by the user, the user is informed of the first inconsistent operation, i.e., an operation whose value has changed after invoking or revoking the operation. This way, the user according to his requirements can accommodate the effects of retroactive operation in to the current state of the data structure. For example, if any wrong information is logged into a database, then the erroneous operation can be revoked when identified. Performing the revoke operation will return the first inconsistent operation. The user can then take corrective actions according to his requirements.

Dickerson et al. [37] addressed the problem of replicating a Voronoi diagram V(S) of a planar point set S by making proximity queries. Their methods to get such a cloning provide one of the first natural algorithmic applications of retroactive data structures, as several of those methods critically rely on the use of such structures.

Demaine et al. [32] introduce the approach of "hierarchical check-pointing" for further improving the transformation process of partial retroactive data to fully retroactive data structures. Hierarchical check-pointing is a logarithmic transformation which provides a generalized framework to transform the partial retroactive data structure to full retroactive structure. This transformation can be applied only to those Partial data structures which are "time-fusible" means that multiple structures with disjoint timespans can be fused into a timeline supporting queries of the present. Using this transformation authors have shown the transformation of a partial retroactive priority queue to a fully retroactive structure. Resultant priority queue has $O(log^2 m)$ amortized time per update for the update operations i.e. inserting an element and deleting the minimum element. The query operation i.e finding the minimum element at any point in time is $O(log^2 m log\ logm)$ per query. The space requirement of the structure is $O(m\ logm)$.

Authors have given an implementation of the fully retroactive priority queue using the scapegoat tree. Scapegoat tree supports the basic operations in $O(logn)$ amortized time where n is the number of nodes in the tree. Scapegoat trees [57] are the balanced binary trees which

preserve height balance by guaranteeing weight balance at each node with the help of a parameter associated with it. This parameter helps to keep a record of the weight balance at each node. As soon as a node violates the weight balance, the subtree rooted at that node is reorganized to reach the weight balance value again. To implement the fully retroactive priority queue, the leaves of the scapegoat tree store all updates to the queue across time, with the leaves sorted from left to right according to time. Each non-leaf node x, stores a single partially retroactive priority queue that tracks all updates to the subtree rooted at x. Update time for the structure is $O(log^2 m)$ and query time is $O(log^2 m log\ log m)$.

Graphs, or networks, are among the most basic and powerful objects to model relations and processes in sociology, economy, physics, chemistry, computer science, biology, engineering, and anthropology, as well as many other fields. The idea of using graphs/networks was around long before the modern computers entered the scene. But tremendous growth in the availability of data and in the size of networks is driving an increasing need to devise new computational methods to represent, store, and process large graph/network data in a cost-effective and timely manner for the solution of related problems. One of the most widely studied graph problems is the shortest path problem in static as well as dynamic case. This is a generic problem with applications in many different fields such as Operation Research, Management Systems, Computer Science and Artificial Intelligence. Main reason for its use in such diverse fields is that any combinatorial optimization problem can be expressed as a shortest path problem. This is a very large class of problems and includes numerous practical problems that are least related to the shortest path problems [129] and we have chosen it for applying the idea of retroactive data structures for dynamization purpose.

Different variants of the dynamic shortest paths problem exist like semi-dynamic or fully dynamic. Semi-dynamic can be either incremental or decremental. Incremental means only insertions/ edge weight decreases are allowed and decremental means only deletions/edge weight increases are allowed. Many approaches have been used by authors for the solution of the

different variants of the problem like *Gallo* [56] and *Fujishige* [54] gave solution for the semi-dynamic decremental problem and *Even and Shiloach* [41] gave solution for the incremental case. The widely accepted algorithm for the problem was given by *Ramalingam and Reps* [113]. The main advantage of their algorithm is that it updates the shortest-path graph, rather than a shortest-path tree. Also, the algorithm is for the case that has only positive weight cycles even in the presence of dynamic updates. The algorithm has been widely used by different authors as it has good performance in most cases. Also, a variation of the algorithm has been proposed by *Demetrescu et al.* [35], which allow updating the shortest path tree rather than the shortest path graph. Many authors have provided solutions only for the case which considers a single update at a time [16, 53, 136]. A different approach has been proposed by *Narviaez et al.* [101], in which the batch updation is performed. *Edward Chan and Yaja Yang* [22] have also proposed a dynamic version of Dijkstra algorithm DynDijkstra. DynDijkstra are two semi-dynamic algorithms that can be used for edge weight increases and decreases in a batch respectively. Also, they have corrected and extended the work of Narviaez *et al* [101] and *Ramalingam and Reps* [113].

Buriol et al. [17] have used an altogether different approach to improve the efficiency of the dynamic shortest path problem. They have given a method to reduce the size of the priority queue used in the shortest path problems. Instead of placing all the affected nodes in the queue only a few selected nodes are placed, which help to reduce the queue size which in turn positively affects the computational time of the solution.

2.2 Massive Data

In the information age, one of the fastest growing databases are the textual databases- like news, Genome databases, Books collections, Digital libraries etc. Many techniques have been developed to process this massive data in a fast and efficient way like: Hadoop MapReduce framework [29]. Their ultimate impact heavily depends on the ability to store and search the information present in these. With the ever decreasing cost of external storage devices, storage of

such massive data is not a big issue, but efficiently extracting the relevant information [61] from this data stored on external storage is an issue due to the slow access speed of these devices. In order to handle this issue, specialized indexing structures and searching tools have been devised. The main idea behind indexing structures is to focus the search of a pattern string to a small portion of the text collection. Different types of indexes were introduced to handle the widely varying type of data available like term-based indexes (generally used in search engines [127]) and full- text indexes. One such type of indexing structure is full-text index which can deal with arbitrary texts and general queries. One more advantage of these indexes is that these are self-indexes means, once the index is created, the corresponding text is not required afterward, as the same can be recreated from the index itself. These indexes have been successfully applied to string matching problem, genetic sequences, and text compression and for indexing special linguistic text. The most common complexity measures for evaluating their efficiency are time and extra space required to construct the index itself, the time required to evaluate any query and the space required to store the index. In this research work, we are concentrating on the first efficiency measure i.e. the efficiency of the index construction process for massive data.

Starting from the suffix tree data structure, there have been many breakthroughs in the full-text index structures like suffix array by *Manber and Myers* in 1990 [91], enhanced suffix array [2] and extended suffix array [120]. Further improving the capabilities of suffix array data structure led to a new class of data structure- abstract data structures like compressed suffix array [44, 45, 63, 72, 89] and then compressed suffix trees [20, 49, 118, 133] with many variations in their implementations. The research in this class of data structure has been oriented towards finding an optimal space/time trade-off in their implementation either during the construction or in their use as an indexing data structure. Space/time trade-off is a major concern during the construction phase, as the size of the text which needs to be indexed is increasing day by day far exceeding the memory capacity of today's computers. At the core of all these data structures is the suffix array which is invariably used in one form or the other in all of these variants.

Research for devising efficient construction algorithm for Suffix array started with its introduction in the year 1990 [91]. Authors gave an *O(nlogn)* time algorithm to directly construct suffix arrays, where *n* is the text length. The algorithm uses the doubling technique given by *Karp et al.* [79]. Most of the algorithms developed for the problem in late 90's were based on traversing the underlying suffix tree. These algorithms are linear time algorithms but their space requirements were very high. The space requirement of the even the space efficient implementation by *Kurtz* [85] is between *8n* and *14n* bytes, for a text of size *n*. Also, the running times of linear time suffix tree construction algorithms do not match the theoretical bounds because these do not take into consideration the memory hierarchy in their implementation. The reason being the high cache miss rate when the suffix tree grows over a specific size. Since then many algorithms were devised but first linear time construction algorithm was simultaneously given by three different groups of researchers *Ko and Aluru* [84], *Karkkainen and Sanders* [76] and *Kim et al.* [82]. Shortly after, *Hon et al.* [71] gave a linear time algorithm with working space requirement of *O(n)* bits. Some researchers simultaneously gave practical construction algorithms. *Larsson and Sadakane* [86] have proposed an *O(nlogn)* algorithm, called qsufsort, with *8n* bytes of working space requirements. Their algorithm is also based on the doubling technique. *Kim et al.* [82] proposed an algorithm based on divide-and-conquer approach, which is an extension of their previous algorithm [81] and has *O(nlog logn)* worst-case time complexity. The algorithm proposed also has faster practical running times than the previous linear time algorithms. Both algorithms of *Kim et al.* [81, 82] use the odd-even scheme for suffix tree construction given by *Farach* [42].

A class of construction algorithms is mainly concerned about space. They are called lightweight algorithms because they require small working space (these trade-off time over space). *Itoh and Tanaka* [74], as well as *Seward* [125] proposed algorithms using only *5n* bytes and theoretically these, have worst-case time complexity $\Omega(n^2)$. However, these are fast in practice provided average LCP of the corresponding text is small. *Manzini and Ferragina* [94]

have given an algorithm called deep-shallow suffix sorting. They have used the idea of sorting the suffixes by using corresponding LCP information. The algorithm can be used for texts with high average LCP. Space demands of the algorithm are low as it uses less than *n* bytes of additional working space but it's worst-case complexity is as high as $O(n^2 \log n)$. Another lightweight algorithm was developed by *Burkhardt and Karkkainen* [18] called as the difference-cover algorithm. Its worst-case running time is $O(n \log n)$ by using $O(n/\sqrt{\log n})$extra space. Theoretically, resource bounds are small but practically algorithm is on average slower than deep-shallow suffix sorting for real-life data.

In 2009, *Nong et al.* [104] engineered an extremely elegant linear time algorithm called SAIS that was fast in theory as well as practice and is based on the induced sorting principle [74]. Working space requirements for the algorithm are so small that it is generally said to be an in-place algorithm. The algorithm is one of the fastest known suffix array construction algorithms. A very efficient implementation has been done by *Yuta Mori* [96], which is practically better than almost all the previous linear construction approaches. All the internal memory algorithms have been divided into three different categories based on the approach used

- ***Prefix doubling*** algorithms are based on the strategy of *Karp et al.* [79]. In this strategy, the prefixes which are in the correct lexicographic order in the text are found. The prefixes identified in the previous step are doubled in length in successive iterations of the algorithm.
- ***Recursive*** algorithms are based on the approach of the suffix tree construction algorithm by *Farach* [42]. A subset of suffixes is sorted and this sorted subset is used to recursively sort the remaining set of suffixes. These sorted suffixes of the two sets are then merged to compute the suffix array of the whole text.
- ***Induced copying*** algorithms are similar to recursive algorithms only difference being the use of iteration for inferring the lexicographic order of suffixes of the text from the already sorted order of a subset of suffixes.

2.3 External Memory Construction Algorithms

In traditional algorithm design, the internal/main memory is assumed to have infinite size. Also, it is assumed that it allows random uniform access to all its locations, such that accessing data from any of the locations takes constant time. These assumptions help the designer to assume that all the data fits in the internal memory and also memory access time is constant and hence need not be considered for predicting the runtime of the algorithm. These algorithms are also called as internal memory algorithms. Based on all these assumptions, the performance of an algorithm is decided by the number of instructions executed, and therefore, the algorithm optimization is achieved by minimizing the number of instruction executions. But these assumptions are not valid while handling massive data because the bulk of data that doesn't fit in main memory is stored in cheap but slow external/secondary memory. Accessing data from external memory is much more expensive as compared to that of internal memory. So, the number of instructions executed does not provide a proper prediction of algorithm runtime. A better predictor will be the time taken for transferring data to and from the external memory also called as input/output (I/O) communication. Due to the slow access speed of secondary memories, I/O communication is also very slow. We now describe some external memory models of computations that have been given for representing the performance of external memory algorithms.

2.3.1 The External Memory Model

In this memory model, introduced by *Aggarwal and Vitter* [5], primary assumption is that most of the data is stored in the secondary/external memory. The secondary memory is divided into blocks. The transfer of a block of data from or to the secondary memory is called as input/output (I/O). The processor's clock period and the main memory access time are negligible as compared to the secondary memory access time. The number of I/O's performed by the algorithm, give the performance of the algorithm. Algorithms designed on this model are referred to as external memory algorithms. The model defines the following parameters: the size of the problem input

(*N*), the size of the main memory (*M*), and the size of a disk block (*B*). It has been shown that, on this model, the number of I/O's needed to read (write) *N* contiguous items from (to) the disk is Scan(N) = Θ(N/B), and that the number of I/Os required to sort *N* items is Sort(N) = $\Theta((N/B)\log_{M/B}(N/B))$ [5]. For all realistic values of *N*, *B*, and *M*, Scan (N) < Sort(N) << N.

2.3.2 External Memory Algorithms for Suffix Array Construction

As seen in section 2.3, a considerable amount of research for internal memory suffix array construction has been directed towards engineering lightweight algorithms [16, 104]. But as we move on to large text collections like web crawls, Wikipedia or genomic databases which are impossible to hold in internal memory, all the internal memory space efficient algorithms fail. This is an indication that there is a need to move from internal memory construction to external memory construction which will provide a scalable solution to this problem. For external memory construction algorithms space efficiency is not of much concern as secondary memory is much cheaper than the primary memory. But at the same time access speed of these devices is 10^5 to 10^6 times slower than that of internal memory [134]. So, running time of the algorithms utilizing external memory is dominated by the time to transfer data to and fro from the disk. Modern operating systems use caching and prefetching heuristics to take advantage of locality of reference to overcome this I/O bottleneck in case of data stored on external memory. But these are designed to be general-purpose, hence cannot efficiently utilize the locality present in every computation. To get substantial gains in performance locality should be incorporated directly into the algorithm design according to the application under consideration. These algorithms and data structures that explicitly manage data placement and movement are known as external memory (or EM) algorithms and data structures. These are sometimes also referred to as I/O algorithms and data structures. The performance of external memory construction algorithms is also dependent on the underlying application.

The first I/O optimal algorithm for suffix array construction was given by *Bender et al.* in 2000 [10]. This algorithm is based on suffix tree construction and uses divide and conquer

approach for suffix sorting. However, the algorithm is very complex and adds high constant factors to the complexity making it impractical. An extensive implementation study for external suffix array construction had been done by *Crauser and Ferragina* [27]. They implemented several non-pipelined variants of the doubling algorithm [8]. According to the survey algorithm by *Gonnet et al.* [60] takes *O(NMscan(n))* I/Os. But the algorithm is quite expensive due to significant internal work, leading to high running time. But in case of small text sizes or fast machines, this algorithm might be suitable.

There are many external memory suffix array construction algorithms which are theoretically optimal with internal work *O(nlogM/B(n/B))* and I/O complexity*((n/B) logM/B(n/B))* [13, 77], where *M* is the size of the RAM and *B* is the disk block size. All these algorithms also have been implemented [13, 33]. General-purpose EM string sorting routines have been described by *Arge et al.* [8]. EM methods have also been developed for the construction of related text indexes like the Burrows-Wheeler transform [19].

CHAPTER-3

Proposed Approach to Dynamic Shortest Path Problem using Retroactive Data Structures

In this modern era of technological overgrowth, the tremendous increase in the availability of data and in the size of networks leads to the design of new computational methods to represent, store and process this data efficiently both in terms of time and space. Moreover, the data from widely varying sources like biological experiments, Internet routing information, sensor data, and data available through search engines and audio/video devices is ever changing and requires new methods for managing data [66] to solve related problems. So, managing such a massive and dynamic data needs to consider all the issues like storage, searching, retrieval etc. along with maintaining the dynamism of the data. In the initial years of its development, maintaining dynamism involved updating the existing approaches of data storage and retrieval explicitly to accommodate the dynamic changes in the data. But in the year 2004, with the introduction of a new data structuring paradigm called as retroactive data structures [30], it has become quite easy to develop dynamic algorithms from the existing static ones.

In this chapter, we are considering the dynamization of static algorithms using the innovative concept of retroactive data structures. Graphs, or networks, are the most basic and powerful objects to model relations and processes in widely varying fields like sociology, economy, physics, chemistry, computer science, biology, engineering, and anthropology etc. Also well-defined problems on graphs like shortest path, vertex cover, edge cover etc. help to solve the real world problems in these varying domains. So, we are taking one such graph problem i.e. the shortest path problem which we are going to dynamize. In section 3.1, we give an overview of the shortest path problem and some of its variants. Then in section 3.2, we explain the idea of solving the Dynamic single source shortest path problem using retroactive data structures starting with problem specification and terminology used and then briefing the idea of

implementing the retroactive priority queue using priority search trees in section 3.3. Section 3.4 gives the proposed approach for the solution of the problem along with the algorithm. Section 3.5 explains in detail the analysis of the proposed algorithm. The analysis has been done both in terms of correctness and complexity. The section also details the experimental work done, corresponding results and their discussion. Finally, section 3.6 concludes the chapter.

3.1 Single Source Shortest Path Problem

Given a directed weighted graph *G(V, E)* and a source vertex s, Single Source Shortest Path (SSSP) problem is the problem of determining the shortest paths from the source vertex s to all the other vertices of the graph. SSSP problem [119] is one of the fundamental graph problems. This problem has applications in diverse fields like network routing, geographic information systems, artificial intelligence, operation research etc. Two variants of this problem exist: static and dynamic. In the static variant, given a graph, shortest paths for all the vertices of the graph from the specified source vertex are calculated and the solution is complete. Most frequently used solution for the static version of the SSSP problem on graphs with non-negative weights is Dijkstra's algorithm [38].

3.2 Dynamic Single Source Shortest Path (DSSSP) Problem

In the dynamic variant of SSSP problem called as Dynamic Single Source Shortest Path (DSSSP) problem, the underlying graph may change dynamically i.e. either the weight of edges may change (*increase or decrease*) or some edges may be inserted or deleted from the graph. So, for the solution of DSSSP problem, we need to update the existing solution in case the underlying graph is changed. We can formally state the DSSSP problem as : let *T* be a Shortest Path Tree (SPT) for vertex *s* in graph *G* and some dynamic change arises for the graph, and graph *G'* is obtained by applying the given change on *G*. Then, the goal of DSSSP problem is to find a new SPT *T'* for *G'* using *T* with minimum number of re-computations and without computing *T'* from Scratch.

Dynamic changes that can arise in a graph are edge weight changes or topological changes(means edges or vertices are added or deleted from the graph). On top of this, these changes can occur either in batches or as singular changes. According to the type of dynamic changes arising in the graph, we are considering the DSSSP problem as increase only in which only edge weight increases can occur; decrease only in which edge weight decreases can occur and mixed in which both types of changes can happen. The edge insertions can be handled by edge weight decreases as we assume that for the edge inserted, weight has decreased from ∞ to some positive value. Similarly, edge deletions can be handled as edge weight increases from a value to ∞.

Much work has been done to avoid calculating the solution for DSSSP problem from scratch, each time a change arises. Most of this work is based on identifying the set of vertices that have been affected by the given dynamic change and calculating the correct values of shortest paths for these vertices. Almost all the previous approaches on this problem vary in how these identify the affected vertices and how these calculate the correct values of the affected vertices [16, 22, 35, 53, 101, 136]. Further, an altogether different approach for the solution of the problem has been suggested by *Acar et al.* [3]. They have given a generalized idea of how static algorithms can be dynamized by using non-oblivious retroactive data structures. Non-oblivious retroactive data structures are a variant of retroactive data structures introduced by *Demaine et al.* [30, 31].

3.3 A Retroactive Approach towards the Solution of DSSSP Problem

Retroactive data structures are those in which history of all the operations performed on the data structure is maintained. In the retroactive data structures, the changes arising in the data structure due to some modification in the past are automatically accommodated in the present. But in the non-oblivious retroactivity, the changes are propagated one by one i.e. the user is informed of the first operation that is affected due to the modification done in the past and so on. We are basically using this non-oblivious variant of retroactive data structures for the solution of our problem.

3.3.1 Defining Retroactive Priority Queue for Dynamization

3.3.1.1 Retroactive Priority Queue

A priority queue data structure is an extension of queue data structure, in which all the elements have a priority value associated. The deletion of values from the priority queue is according to the associated priority value. Whenever deletion is applied, element existing in the data structure with the highest priority value is deleted. So, the operations of the Priority queue are defined as:

- **Insert(x):** insert an element with key x into the priority queue.
- **Delete ():** deletes an element with the highest priority from the priority queue.
- **Find:** Return the element in the priority queue with the highest priority.

This priority queue data structure can be converted to its retroactive counterpart, by allowing the operations of the data structure to be performed along the timeline. A retroactive priority queue (RPQ) data structure supports the operations as:

- **Invoke (Op, t):** invokes/inserts a priority queue operation *Op* into the retroactive priority queue's timeline at time *t*. Invoke operation can invoke either an insert operation or a delete operation.
- **Revoke (Op, t):** revokes/deletes a priority queue operation *Op* from the retroactive priority queue's timeline at time *t*.
- **Find-Min (t):** returns the element existing in the priority queue with the smallest key at time *t* in the timeline.

But simply adding time parameter as a dimension is not enough as complicated interdependencies exist among operation sequences. Retroactive priority queues are more challenging to implement due to major nonlocal effects caused by a minor modification to the past in the priority queue. The lifetime of almost all the operations is effected by a single operation performed in the past. So, we need a succinct representation for efficiently

representing and managing these cascading effects to avoid the cost incurred in explicit maintenance of element lifetimes.

3.3.1.2 Defining Retroactive Priority Queue using Priority Search Trees

Retroactive data structures define all the operations with respect to a time parameter and also provide to invoke and revoke the basic operations defined for the underlying data structure. All the operations performed in the past are termed as the retroactive operations and each retroactive operation returns the first operation (also called as retroactive successor) that becomes inconsistent due to this retroactive operation. Retroactive successors for the retroactive operations are defined based on the underlying problem also.

To implement the Retroactive Priority Queue (RPQ) for our purpose, we use Priority Search Tree (PST) given by *McCreight* [95] as the underlying data structure. A priority search tree is a combination of a heap and a balanced search tree. It was discovered to be used for range queries where at least one of the sides of the range is unbounded. The PST is a search tree on one dimension and is a priority queue on the second dimension. In our case, we use insertion time of a vertex in the RPQ as the first dimension and its predicted distance value as the second dimension. So, each node of the RPQ stores the *ins_time* (insertion time) of the vertex as well as *d* (predicted distance value). Some auxiliary information is also stored in each node as the corresponding vertex (*v*) as well as the *del_time* (deletion time) of that vertex from the priority queue. Each node also maintains the insertion time corresponding to a vertex in its sub-tree which has not been deleted yet and which has minimum distance value existing. A priority search tree representing a set of *N* values occupies *O (N)* space. So, the space-bound of our RPQ is also linear in the number of vertices $|V(G)|$ in the graph, as each vertex represents a value that needs to be stored in the RPQ. Each vertex *v* with distance value *d* and insertion time *ins_time* is stored at the unique node *x* in priority search tree such that all the nodes in the left subtree of *x* have insertion time less than or equal to *v's* insertion time and in its right subtree have insertion time greater than *v's* insertion time, also distance *d* of *v* is less than the *d* value of all of its

descendants. Whenever we need to insert a vertex *v* with distance value *d* in the priority queue at present time *t*, we insert *(t, d, v)* in the priority queue if vertex *v* does not exist already in the priority queue. If it exists already, then the existing *d* value and current *d* values are compared. If the existing value is less than the current value then the existing value is retained and nothing is inserted else the existing value is deleted and new value is inserted. As insertion or deletion from the underlying priority search tree takes *O(log n)* time, thus insertion time of our RPQ is *O(log n)*.

Similarly, whenever deletion is to be performed at the present time, we have to search for the minimum distance value such that the value has not been deleted yet. The root node of the RPQ gives the corresponding insertion time, which can then be searched in *O(log n)* time in the priority queue and successively deleted from it with total time complexity *O(log n)*. Now, whenever we need to accommodate a weight change in the SPT, we get the *ins_time(t)*, *del_time(t')* and d value of the vertex *v* whose shortest path has changed. We query the RPQ for the value *(t, d, v)* and delete that value. This deletion of value in the past returns the values inserted at time *t'* as the inconsistent operations. The query for the inconsistent operation identification can be formulated as:

***R_Succ_Del*:** Given *x0* and *x1* where *x0* is the *ins_time* for the deleted value and *x1* is its *del_time*, we need to find all nodes *x* in priority queue such that *x.ins_time = del_time*.

The above query is a simple search query of a binary search tree, which can be performed in *O(log n + k)* time where *k* is the number of values returned.

This deletion of value in the past is succeeded by insertion of a value with the same *ins_time* but with distance value d_{new}. This insertion operation in the past returns the value with the minimum value greater than the d_{new}, as the inconsistent operation. The query for the inconsistent operation identification can be formulated as:

R_Succ_Ins: Given *x0, d*, where *x0* is the *ins_time* for the inserted value and *d* is its distance value, we need to find node *x* such that $x.d >_{min} d$.

The above query is similar to the query of priority search tree which identifies minimal *y* value in given *x* range. The only difference is that we have added a condition with the *y* dimension. The query in priority search tree takes *O(log n)* time, giving same bounds for our query also.

Thus, we can say that the RPQ has *O(n)* space bound with all operations performed in *O(log n)* time.

3.4 Proposed Approach

Our approach to solve the DSSSP problem is to dynamize the Dijkstra algorithm using the retroactive priority queue data structure defined in section 3.3. This data structure helps to propagate the changes made to the data structure in the past to be propagated to the present by suitably and efficiently identifying the affected operations in the operation sequence of the data structure. Below we give some of the notations to be used further as well as some of the implementation issues of the approach.

3.4.1 Terminology Used

A *directed weighted graph G = (V(G), E(G),w)* consists of a set of *vertices V*(*G*) , a set of *edges E*(*G*) and weight function *w(u, v).* An edge directed from *u* to *v is denoted* by *(u, v)*. If (*u, v)* is an edge in G, then *u* is the tail vertex of the edge and is represented as e_t and *v* is the head vertex and is represented as e_h. Let *out(v)* be defined as the set of all vertices such that *out(v) = {m | m ϵ V(G) and (v, m) ϵ E(G)}*. Similarly *in(v)* can be defined as the set of all vertices such that *in(v) = {m | m ϵ V(G) and (m, v) ϵ E(G)}*. Let *P(u, v)* be a path from *u* to *v* in *G*; then *v* is said to be reachable from *u*. All vertices reachable from *u* in *G* including itself are called as *u*'s successors and are denoted as *succ(u*). A path *P(u, v)* is said to be a *shortest path*, denoted as *SP(u, v)*, if it is shorter than all the possible paths *P(u, v)* in *G*. The shortest distance from *u* to *v* in *G* is denoted as *d(u, v)*. For the given graph *G*, an SPT (shortest path tree) rooted at a vertex s (source vertex), denoted as *T*, is a tree with root *s* and descendants v, and *T* contains an *SP(s, v),* $^{\forall v \in succ(u)}$, $v \neq s$.

Also $E(T) \subset E(G)$, i.e. the edge set of *T* is a subset of the edge set of corresponding graph *G*. In *T*, *d(s, v)* is denoted as *d(v)* i.e. *d(v)* is the shortest distance of vertex v from source vertex *s*.

3.4.2 Implementation Issues

Our algorithm maintains a RPQ (Retroactive Priority Queue) data structure which helps to generate *T* in the static case and also helps to generate *T'* in dynamic case by propagating the required changes step by step in to the existing *T*. This section gives the details of the graph representation that we are going to use for our purpose.

3.4.2.1 Graph Representation

We are using a graph representation similar to the one used by [17]. The advantage of using the representation is that the *in(v)* or *out(v)* sets for any vertex can be calculated efficiently, without looking in to all the vertices of the graph. The input graph is represented using five arrays. For each edge $e \varepsilon E(G)$, array *w[e]* denotes the weight of edge *e*. Two arrays *Arc_Forward* and *Arc_Reverse* are used to explicitly store the arcs and have the size $|E(G)|$ and another two arrays *Findex* and *Rindex* are used as indexes in to the previous two arrays respectively and their size is $|V(G)|+1$. The array *Arc_Forward* stores the arcs of the input graph, where each arc is represented as its head vertex and is pointed to by the i^{th} index (where *i* is the tail vertex of the arc) of the *Findex* array. So, the i^{th} position of the *Findex* array indicates the initial position in *Arc_Forward* of the list of outgoing arcs from vertex *i and* the last position in *Arc_Forward* of the list of outgoing arcs from vertex *i* is $Findex[i+1]-1$. In array *Arc_Forward*, the arcs are ordered by their tails and arcs corresponding to the same tail are ordered by their head vertices. Also, this array can be used effectively to identify all the outgoing arcs from a given vertex.

Similarly, the arrays *Arc_Reverse* and *Rindex* are defined. The *Arc_Reverse* array reverse stores the arcs, where the arcs are sorted by their heads, with ties broken by their tails. The i^{th} position of array *Rindex* points to the initial position in *Arc_Reverse* of the list of incoming arcs into vertex *i*, and the last position in *Arc _Reverse* of the list of incoming arcs into vertex *i* is

Rindex[i+1]−1. Array *Arc_Reverse* can be effectively used to identify all the arcs incoming into the vertex *i*.

3.4.3 Proposed Algorithm

The underlying algorithm for our solution is the Dijkstra algorithm. To compute the shortest paths efficiently, the algorithm maintains the computed shortest paths as the shortest path tree *T*. All the edges corresponding to a vertex that are included in *T* are the children of that vertex in the tree denoted by *succ(v, T)*. Each node of the tree contains auxiliary information field, the distance value denoted by *d(v)* which gives the shortest path of *v* from source vertex *s* in *T*. This tree structure changes progressively during the execution of the algorithm (in both static as well as dynamic changes). When the execution of the algorithm is complete the data structure gives us the solution for the dynamic single source shortest path problem from a specified source vertex.

In addition, the algorithm also maintains a mapping from vertices to corresponding insertion time as well as deletion time in the RPQ and a heap that stores the inconsistent operations (ordered by time) returned by the retroactive operations in the RPQ due to a dynamic change. Whenever a dynamic change comes, we backtrack in the underlying RPQ to a point where the vertex related to the dynamic change had been inserted. This RPQ helps us to do minimal backtracking required to accommodate the dynamic change. All the operations performed after the deletion of that vertex from the RPQ may be affected by this change. The operations that are affected by this change are identified and corrected which in turn create inconsistency at the next level. So, level by level the inconsistent operations are identified and corrected. The algorithm consists of two procedures; one is *D_Dijk_incr* that handles the dynamic changes involving the edge weight increases. Another one is *D_Dijk_Dec* which handles the cases in which edge weights are decreased. In the first step of both the algorithms, the underlying graph is updated according to the edge weight change. The heap data structure is used to maintain the inconsistencies arising during a dynamic update, these inconsistencies are ordered by time and in

case the time is same for two inconsistencies, the tie is broken by their *d* values. This heap data structure has entries of the form *(ins_time, d, v, del_time)*, where *ins_time* is the insertion time of the vertex, *d* is the distance value, *v* is the corresponding vertex and *del_time* is the deletion time of the vertex.

3.4.3.1 Algorithm for Dynamic Dijkstra when Weight of an Edge is Increased.

D_Dijk_incr (u, v, δ)

Step 1: Update the graph.

//(u, v) is the edge whose weight has been increased by amount δ

W'(u, v) = w(u, v) + δ

Step 2: Accommodate change in SPT

//*Update the distance of tail vertex of changed edge i.e. d(v).*

$D(v) = d(v) + \delta$

for each $m \in V(G)$ s. t. $(m,v) \in E(G)$

if ($d(m) + w(m,v) < D(v)$)

$D(v) = d(m) + w(m,v)$; P(v) = m

if ($D(v) > d(v)$)

$d(v) = D(v)$

$p(v) = m$

else

Quit (no change in SPT)

Step 3: Propagate the change in the shortest path further in SPT.

$t \leftarrow ins\_time(v)$

$t' \leftarrow del\_time(v)$

Revoke(*Insert*(t))

// *revokes the entry corresponding to vertex v in RPQ and Returns the inconsistent operation and adds this to heap H.*

Invoke(*Insert*(t, d(v),v))

// *inserts vertex v with new distance value at time t in Q.*

While *non_empty*(H)

do $k \leftarrow \min\_heap()$

// *Remove 1st inconsistent operation from H*

if ($k \in succ(v)$ in SPT)

go to step 2.

else

Increment the deletion time of vertex k by 1 in RPQ.

3.4.3.2 Algorithm for Dynamic Dijkstra when Weight of an Edge is Decreased

D_Dijk_Dec(u, v, δ)

Step 1: Update the graph.

//(u, v) is the edge whose weight has been increased by amount δ

w'(u, v) = w(u, v) + δ

Step 2: Accommodate change in SPT

//*Update the distance of tail vertex of the changed edge.*

if($v \in succ(u)$ in SPT)

$D(v) = d(v) - \delta$, $P(v) = p(v)$

else $D(v) = d(u) + w'(u,v)$ **;** $P(v) = u$

$\Delta = d(v) - D(v)$

if ($\Delta > 0$)

$d(v) = D(v)$

$p(v) = P(v)$

else

Quit (no change in SPT)

Step 3: Propagate the change in the shortest path further in SPT.

$t \leftarrow ins\_time(v)$

$t' \leftarrow del\_time(v)$

Revoke(*Insert*(t))

//*revokes the entry corresponding to vertex v in RPQ and returns retroactive successor and adds this to heap H.*

Invoke(*Insert*(t, d(v),v))

// *inserts vertex v with new distance value at time t in Q.*

While *non_empty*(H) **do**

$k \leftarrow \min\_heap()$ // *Remove 1st inconsistent operation from H*

Case 1:

if ($k \in succ(v)$ in SPT)

$d(k) = d(k) - \Delta$; $p(k) = v$

Goto Step 3

Case 2:

if ($(v,k) \in E(G)$ and $d(k) > d(v) + w(v,k)$)

$d(k) = d(v) + w(v,k)$; $p(k) = v$

Goto Step 3

else

Increment the deletion time of vertex k by 1 in RPQ.

3.5 Analysis of the Algorithm

3.5.1 Correctness Analysis

In this section, we outline the proof of correctness of the proposed algorithm and give some of its properties.

The objective of the algorithm is to update the shortest path tree after the weight of an edge has been changed in the underlying graph. Let *G* be the original graph, *(u, v)* be the modified edge of *G* and *T* be the shortest path tree before the edge weight change. The input to the algorithm is the graph *G*, shortest path tree *T* and the modified edge *(u, v)*. To establish the correctness of the algorithm, we need to prove that following properties are maintained by the algorithm.

- *P1: For each vertex* $v \in V(G)$ *s.t.* v *is affected by given change, the priority queue returns* v *as an inconsistent vertex.*
- *P2: For each vertex* $v \in V(G)$ *s.t.* v *is affected by given change, the distance value calculated for the vertex is correct.*

Lemma 1:

In Dijkstra algorithm, if a vertex u is deleted from the priority queue before vertex say v, then in no case vertex v can be a predecessor of vertex u in the shortest path of vertex u from the source vertex s.

Proof:

As we know, if a value *x* is deleted from priority queue before some other value *y* and then from the property of a priority queue, *x* should be less than *y*. So, in our case, if vertex *u* is deleted before vertex *v*, this means $d(u) \le d(v)$. Now, to prove the property we assume that *v* be a predecessor of *u* in its shortest path from the vertex *s*.

$$\Rightarrow d(u) = d(v) + w(u,v)$$

$$\text{Thus } d(u) \ge d(v)$$

This contradicts the above inequality.

Lemma 2:

In the retroactive priority queue, if we insert a new distance value for a vertex in the past, then the deletion time of this vertex separates the remaining vertices i.e. $\forall v \in V(G), v \neq k$ *, into two sets. One set is the set of all vertices whose deletion time is less than the deletion time of k and is denoted by* S_{NE} *and the other one denoted as* S_E *of remaining vertices i.e. vertices whose deletion time is more than the deletion time of k. All the vertices in set* S_{NE} *are not affected by the given change but the vertices in set* S_E *may or may not be affected.*

Proof:

We will prove this property by contradiction. Let us assume that *u* be a vertex in the set S_{NE} and it is affected by the given change i.e. *k* possibly can be a predecessor of *u*.

As $u \in S_{NE}$

$$\Rightarrow Del\_Time(u) < Del\_Time(k)$$

According to Lemma 1, if above condition holds then k can never be a predecessor of u, hence violating our assumption.

Above lemma 2 proves that property P1 holds. Now to prove the property P2, we develop on the notation of inconsistent vertices as given by *Ramalingam and Reps* [114]. According to their convention, inconsistent vertices can be of two types either overestimated or underestimated. Let *D(v)* be the actual value that a vertex should have and *d(v)* is the current value it is holding. Then, for overestimated vertices

$$d(v) > D(v)$$

and for underestimated vertices

$$d(v) < D(v)$$

Now, we give some properties of these inconsistent vertices.

Lemma 3:

Whenever we decrease the shortest path of a vertex by decreasing the weight of the corresponding edge, then all the successors of this vertex are overestimated.

Proof:

Let (u, v) be an edge whose weight has been decreased.

$$w'(u,v) = w(u,v) - \delta \qquad (1)$$

Now, if (u, v) be an edge in the SPT then,

$$d(v) = d(u) + w(u,v)$$

But after edge weight change,

$$D(v) = d(u) + w'(u,v)$$

From eq. (1)

$$w'(u,v) < w(u,v)$$

$$\Rightarrow d(u) + w(u,v) > d(u) + w'(u,v)$$

$$\Rightarrow d(v) > D(v)$$

Lemma 4:

Whenever we increase the shortest path of a vertex by decreasing the weight of the corresponding edge, then the successors of this vertex are underestimated.

Proof:

Analogous to that of property 3.

Lemma 5:

All the overestimated vertices can be made consistent without considering adjacent vertices.

Proof:

As by Lemma 3, for an overestimated vertex

$$d(v) > D(v)$$

Also by the property of shortest path tree, shortest path of a vertex is defined as

$$\forall m \in in(v)$$

$$d(v) = \min d(m) + w(m,v)$$

$$\Rightarrow D(v) < \min d(m) + w(m,v)$$

So, $d(v)_{new} = D(v)$

Thus, no edges incoming in vertex *v* can lead to a shorter path than *D(v)* which itself is less than *d(v)*. So, we need not consider the adjacent vertices while making an overestimated vertex consistent.

Lemma 6:

To make the underestimated vertices consistent, the adjacent vertices also need to be considered.

Proof:

For an underestimated vertex,

$$d(v) < D(v)$$

By the property of shortest path tree, shortest path of a vertex is defined as

$$\forall m \in in(v)$$

$$d(v) = \min d(m) + w(m,v)$$

$$\Rightarrow D(v) > \min d(m) + w(m,v)$$

So, to calculate the shortest path of v from s we need to find minimum value i.e.

$$d(v)_{new} = \min(D(v), d(m) + w(m,v))$$

Above equation shows that all the adjacent vertices also need to be considered for calculating the correct distance value.

3.5.2 Complexity Analysis

In this section, we theoretically explain the time bounds (upper) for the proposed algorithms. The time bounds of the proposed algorithm can be measured in terms of following parameters:

- Size of underlying retroactive priority queue data structure.
- Number of times the loop in the algorithm is executed, which in turn depends upon the number of operations performed in the RPQ after the given change.

In both cases of edge weight change i.e. increase or decrease, the inconsistent vertices arising are handled in two ways. Either the shortest path of the vertex needs to be updated or vertex needs to be updated only in terms of its insertion and deletion time in the RPQ. In first case, we

have to calculate the updated shortest path, delete the previous one from the RPQ and insert the new one. As obvious from lemma 5 and 6, calculating the new shortest path in any case takes constant time. Also, insertion and deletion in the RPQ can be done in *O(log n)* time. Let *m* be the number of operations performed in the RPQ after time *t* at which a change has come and then while backtracking in the RPQ, maximum *m* operations can be returned as the inconsistent operations. So, loop in both the cases will be executed *m* times in the worst-case. Thus the total time complexity comes out to be *O(m log n)* in worst case.

3.5.3 Result and its Discussion

Apart from theoretical correctness (Lemma 1 to Lemma 6), we also validate our results experimentally. Table 3.1describes the details of graphs used for experimentation.

Table 3.1 Experimental Setup

Graph	XML File	Number of Vertices |v|	Number of Edges |E|
Graph-1	NH.xml	197	564
Graph-2	MA.xml	381	1156
Graph-3	WA.xml	496	1354
Graph-4	OK.xml	576	1762
Graph-5	AR.xml	615	1826
Graph-6	SC.xml	737	2344
Graph-7	AL.xml	784	2372
Graph-8	MO.xml	809	2510
Graph-9	VA.xml	890	2716

We have simulated or implemented the proposed algorithm and compared performance with that of the *Ramalingam and Reps* [113] algorithm (R&R) which has good performance in almost all the situations. Furthermore, we have also compared the proposed approach with static Dijkstra's algorithm [38], which re-computes the paths from scratch in case of dynamic changes. The graph data set has been taken from the experimental package available at the

http://www.dis.uniroma1.it/~demetres/experim/dsp/ provided by *Demetrescu et al*. [36]. The graph data used for experimentation is available in the package as XML files and correspond to the real world US road networks data obtained from ftp://edcftp.cr.usgs.gov. The implementation or simulation has been performed using c++ (compiled with gcc- 4.8.2). The comparative results obtained through experimentations on *Static Dijkstra, R&R* and *D_Dijk_Incr* have been shown through Table 3.2 and Table 3.3 and Figure 3.1.

Table 3.2 CPU Time in Seconds for 10^3 edge weight increments

Graph	**Static Dijkstra**	**R&R**	**D_Dijk_incr**
Graph-1	10.98	6.53	4.11
Graph-2	38.6	21.29	12.60
Graph-3	62.78	34.43	20.31
Graph-4	11.29	6.99	4.00
Graph-5	13.72	8.31	5.13
Graph-6	153.1	79.2	45.78
Graph-7	201.52	110.66	70.3
Graph-8	237.24	125.42	90.12
Graph-9	211.880	107.1	62.1

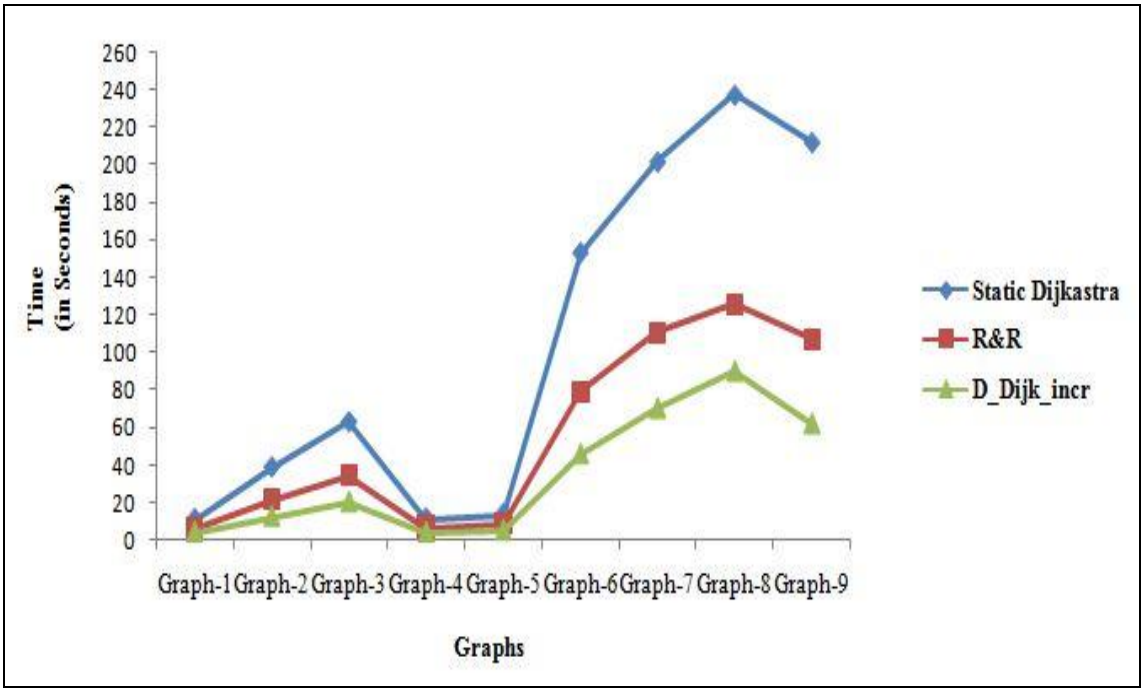

Figure 3.1 Time Usage of the Static Dijkstra, R&R and D_Dijk_Incr Algorithms

Table 3.3 Time Ratio of Static Dijkstra, R&R and D_Dijk_Incr Algorithms

Graph	Static Dijkstra-R&R	Static Dijkstra-D_Dijk_incr	R&R-D_Dijk_incr
Graph-1	1.68	2.67	1.59
Graph-2	1.81	3.06	1.69
Graph-3	1.82	3.09	1.7
Graph-4	1.62	2.82	1.75
Graph-5	1.65	2.67	1.62
Graph-6	1.93	3.34	1.73
Graph-7	1.82	2.87	1.57
Graph-8	1.89	2.62	1.39
Graph-9	1.98	3.41	1.72

As is visible from the results (Figure 3.1) the performance of the proposed algorithm is better than both other algorithms and also the performance gap among the algorithms increases with increase in the underlying graph size due to increase in overhead in the Static Dijkstra and R&R algorithms. Moreover, Table 3.3 indicates that the proposed algorithm is at least one order of magnitude faster than R&R algorithm. This is mainly due to the used Retroactive Priority Queue (RPQ) data structure which effectively stores the state of the graph for further updations without looking into the graph. As is visible from Figure 3.1, the performance gap among the algorithms increases with increase in the underlying graph size due to increase in overhead in the Static Dijkstra and R&R algorithms. We also observed while experimenting that the heap size in our approach is less than that of the other two algorithms. Dijkstra algorithm always inserts all the nodes in the heap for each dynamic update, so, heap size, in that case, is always equal to the number of nodes in the graph. But in case of R&R algorithm, a set of affected nodes is identified and the updations are then applied to only the identified nodes and their successor and predecessor nodes. So, heap size is reduced as compared to the Dijkstra algorithm. In our approach, updations are performed level by level in the graph, so, heap size at any point of time corresponds to the affected nodes of that level of the graph only and hence is significantly reduced as compared to the previous approaches

Our algorithms are based on the extension of the Dijkstra algorithm so that it can update T in $O(m \log n)$ worst case time while doing the minimal backtracking. We efficiently do this backtracking through the use of a non-oblivious retroactive data structure for the priority queue. *Acar et al.* [3] show that RPQ can be maintained in $O(\log n)$ time and $O(n)$ space with simple balanced tree operations. We implement this retroactive Priority queue with the same resource bounds as given above, using priority search tree data as the underlying data structure. The idea of our approach is to solve the dynamic shortest path problem by processing the inconsistencies arising in the solution in the order in which they are returned by the RPQ. Also, RPQ returns the inconsistencies step by step in the increasing order of the deletion time of vertices. A somewhat

similar idea has been given by [113] which say that to solve the dynamic shortest path problem efficiently; the inconsistencies should be processed in the right order i.e. in the increasing order of *d* values.

3.6 Conclusion

We have given efficient algorithms for solving the dynamic single source shortest path problem in which dynamic changes can be in the form of edge weight changes. Our approach is better than those that depend upon identifying the affected set of vertices and then updating their shortest paths according to the given changes. This method uses the retroactive data structures which automatically identify the vertices that are affected by a given change which can then be corrected in constant time. As the changes are propagated level by level i.e. the vertex affected by a given change is identified and corrected which in turn gives the next vertex that has been affected by this new change and so on. In this way, new shortest path tree is generated from the existing one using the minimum amount of topological changes. The advantage of our approach is it can be easily adapted for other types of dynamic changes that can occur like a number of update operations appear simultaneously or edge insertions or deletions.

CHAPTER-4

Proposed Approach for Dynamizing Dijkstra Algorithm using Retroactive Data Structures

In this chapter we propose another method for dynamizing *Dijkstra* algorithm using retroactive data structures. First of all we give an overview of how to implement retroactive priority queue using height balanced trees. After that, we define the problem we are considering along with the assumptions we have made about the problem to be solved. Then we explain how D*ijkstra* can be made dynamic by replacing the data structure i.e. priority queue by retroactive priority queue. After that the representation used for the underlying dynamic graph has been given. In the next section, we give the algorithm Dynamic Dijkstra using retroactive priority queue as a data structure.

4.1 Height-Balanced Search Trees

Search into a large amount of data can be implemented through binary search trees which have search time as *O(h)* where *h* is the height of the tree. Most other operations on a binary search tree also take time directly proportional to the height of the tree. But the height of the tree can vary from $log_2 n$ to *n* where *n* is the no. of nodes in the tree and hence the worst-case complexity for the above mentioned operations is *O(n)*. So, it is desirable to keep the height of the tree small so as to make these operations efficient. The concept of height balanced trees was introduced to keep the height of binary search trees close to optimal (approximately equal to $log_2 n$). A height-balanced binary search tree is a binary search tree that automatically keeps its height (i.e. balances its height) close to optimal even in the presence of on-line insertions and deletions of items. These binary trees balance the height by performing transformations on the tree (such as tree rotations) at key times, in order to keep the height proportional to $log_2 n$. No doubt an overhead is involved in transformations, but it is overcome by fast execution of all other

operations [83]. Various height balanced binary search trees have been defined like: AVL trees [4], weight-balanced trees [103], red black trees [65] and treaps [21] etc.

In all types of height–balanced search trees, balance information is maintained in each node and rebalancing is done after each insertion or deletion by performing a series of transformations (rotations) along the access path (the path from the root to the inserted or deleted item). We need to use these trees as a data structure for the implementation of retroactive priority queues. So, the cost of transformations of the tree will affect the efficiency of our implementation also. As red-black trees have efficient transformation operations as compared to the transformations of other kinds of height-balanced trees [121], so this is a suitable representation for our purpose.

4.1.1 Operation Definition

Red-black tree is a balanced binary search tree with the following properties:

- Every node has a color field along with the left and right links. Color field can have value either red or black.
- Color field of Root node of the tree has value black always.
- The children of a red node are always black.
- On every path of the tree, from the internal node to the leaf nodes, number of black children is same.

Three main operations for the tree are defined as:

Insert: Insertion into a red-black tree consists of following three steps:

- Simply insert the element as in a binary search tree.
- Assign color red to the inserted node.
- Maintain the red-black property of the tree by suitable rotations.

Delete: Deletion of any value from the tree is similar to the insertion operation, but it is more complicated than insertion:

- Delete element as from a binary search tree.
- Apply rotation to maintain the red-black property.

4.1.2 Operation Complexity

All the operations of red-black tree can be performed in *log n* time as given in Table 4.1.

Table 4.1 Time Complexity of Red-black Tree Operations

Operation	Average Case Complexity	Worst Case Complexity
Search	*O(log n)*	*O(log n)*
Insert	*O(log n)*	*O(log n)*
Delete	*O(log n)*	*O(log n)*

4.2 Dynamic Dijkstra

Dijkstra algorithm is the most widely used algorithm for the solution of single source shortest path problem. Originally, it was introduced to find shortest path between the nodes of a graph. Till then, the algorithm has been widely used in network routing protocols, most notably IS-IS and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's algorithm. As we know that solution for the shortest path problem also depends on the edge weights of the corresponding graph i.e. if the edges in the graph have only positive weights then the problem can be easily solved by *Dijkstra* algorithm but if there is no restriction related to the edge weights then Bellman Ford algorithm should be used. One more option available that can be used for general graphs is to transform the graph with negative weight edges into a graph that has only nonnegative weights and that has the same shortest-paths structure. Using such a transformation shortest path problem for general graphs can be easily solved using D*ijkstra* algorithm. We want to solve the single source shortest path problem for the dynamic graphs i.e. graph whose edge weights change. Formally, we can define our problem as:

Let $G = (V, E, w)$ be a directed graph with non-negative edge weights and at any time t the edge weights w of G can be updated (either incremented or decremented). We need to solve the single source shortest path problem for G with minimum number of re-computations.

4.2.1 Idea of Proposed Work

A new data structuring paradigm that has evolved recently is retroactive data structures which help to maintain the historical sequence of events on a data structure. These data structures can be effectively used for the dynamization of static algorithms as given by *Demaine et al.* [31]. The idea of our work is to use these data structures for the dynamization of the D*ijkstra* algorithm. As priority queue is used in the static implementation of the algorithm, so by using retroactive priority queue we can dynamize the algorithm. Retroactive priority queues are complex, as a modification in the past in a retroactive priority queue needs to be reflected in all the successor states of the queue. As any modification in the past in the priority queue has a cascading effect, we need to represent such cascading effects efficiently (in terms of space as well as time) in order to reduce the cost inherent in the explicit maintenance of element lifetimes.

For dynamic implementation of shortest path problem using D*ijkstra* algorithm and retroactive priority queue, we need an implementation of the retroactive priority queue and a suitable dynamic graph representation. So, we proceed further by giving a description of these.

4.3 Approach for Retroactive Priority Queue using Balanced Search Trees

To solve the dynamic shortest path problem we need to have an implementation for the retroactive priority queue so that the priority of the edges could be changed dynamically according to the dynamic changes in their weights i.e. we wish to maintain a set of items whose priority changes over time. So, we define the operations of the queue w.r.t. time. The revoking and invoking of the two main operations of the priority queue i.e. insert and delete also needs to be defined. First of all in section 4.3.1, we give a brief description of the notations to be used

throughout the rest of the chapter. Then in section 4.3.2, the main operations for the retroactive priority queue using red black trees are defined.

4.3.1 Basic Notations

Let $G = (V, E, w)$ be a simple directed graph, where V and E are the sets of vertices and edges, respectively, and w is a function from E to the set of non-negative real numbers i.e. w gives the weights of the corresponding edges. Let $e = (u, v) \varepsilon E$; then u is the tail of e denoted as e_t, and v is the head of e denoted as e_h. Each vertex v is represented by a key (like a, b etc.), and so is each edge *e (as (a,b), (a,c) etc.).* An edge *e (u, v)* in the graph is assigned with a weight *w (u, v).* Each vertex *v* contains the auxiliary information i.e. distance and the predecessor. The distance value denotes the predicted shortest distance of that vertex from the source vertex and the predecessor field gives the immediate predecessor of that vertex on the predicted shortest path.

For the dynamic shortest path problem, we consider that the edge weight changes can be in any of the forms: edge weight increases only or edge weight decreases only or both edge weight increases and decreases. Although we are considering only edge weight changes, but edge insertion/deletion can also be handled by these as edge insertions can be considered as edge weight decreases from ∞ to the weight of the inserted edge. Likewise, edge deletions can be considered as edge weight increases by changing the edge weights of the deleted edge to ∞.

Now, coming on to the retroactive priority queue, Let *Q* be the retroactive priority queue. An entry in queue is of the form *(ver, ins_time, dist, pred, del_time)* in which *ver* is a vertex, *ins_time* is the time at which that node has been inserted in the queue, *dist* contains the predicted shortest distance of the vertex from the source vertex, *pred* is the predecessor vertex for that vertex on the predicted shortest path and *del_time* is the time at which node is deleted from the queue. Nodes in the queue are ranked first according to the distance value (dist) and then on time. If more than one entry has same dist, then the sequence among them is arbitrary. Different operations supported by *Q* are explained in the next section in detail.

4.3.2 Retroactive Priority Queue Operation Definitions

Retroactive priority queue data structure allow all basic operations of data structures i.e. insert, del_min and find_min to be applied at any point of time. The definitions of all these operations are as follows:

- ***Invoke (Insert (x, t))****:* performs an insertion of the item *x* at time *t* i.e. a new element *x* is inserted in to the priority queue at time *t* according to its priority value.
- ***Invoke (Del_min (t)):*** performs the deletion of minimum value from the priority queue at time *t.*
- ***Revoke (Insert (t)):*** Undo the insert operation performed at time *t.*
- ***Revoke (Del_Min (t)):*** Undo the delete operation performed at time *t.* This is equivalent to inserting the item deleted at time *t* again in the priority queue.
- ***Find_min(t):*** Returns the minimum element existing in the priority queue at time *t*.

Starting with an empty set, we wish to perform a sequence of all the above operations online with the property that all operations can occur at any time. This property allows the operations to take place either in the present (i.e. after the most recent update) or in the past. In the time invariant data structures, the time of an operation is implicit, i.e. it is specified by the position of that operation in the sequence of operations. But the time parameter *t* used in the operations of retroactive data structures formalizes the notion of retroactivity and ties in operation times are broken by their order in the sequence of operations.

In the remainder of this section we show how retroactive priority queue data structure can be implemented using height-balanced search trees.

4.3.2.1 Operation Definitions

To implement the retroactive priority queue, we need to have a way to maintain the lifetimes of all the elements along with the elements themselves. So, to have such functionality we maintain 2 height-balanced binary search trees (i.e. red-black tree): T_{ins} and T_{d_m}, which maintain the sets

Insert and Del_Min respectively. The set Insert maintains the values which have been inserted in the priority queue along with their time of insertion and also time of deletion if any. Also, set Del_Min maintains the times at which the *del_min* operations are performed. The tree T_{d_m} is ordered by time. The other tree T_{ins} is indexed by the item value and second indexing is according to the time value i.e. $(k_1, t_1) < (k_2 , t_2)$ if and only if $(k_1 < k_2)$ or $((k_1 = k_2) \wedge (t_1 < t_2))$. The two trees have links between its nodes, as each node of tree T_{d_m} is linked with a node of T_{ins} whose key is the return value for that *del_min* operation. On the similar lines, each node of T_{ins} is linked with its corresponding *del_min* if that node is a result of Revoke (Del_Min(t)) operation. Whenever *del_min* operation is performed the node that is to be deleted from tree T_{ins} is not actually deleted but is marked as invalid as we need to maintain the past information also. At a later time if *del_min* is revoked that node is again marked as the valid one.

The retroactive priority queue also needs to identify the operations whose return values are affected, in addition to above operations. The definition of all the above operations along with the pseudo code in terms of trees T_{ins} and T_{d_m} is as follows:

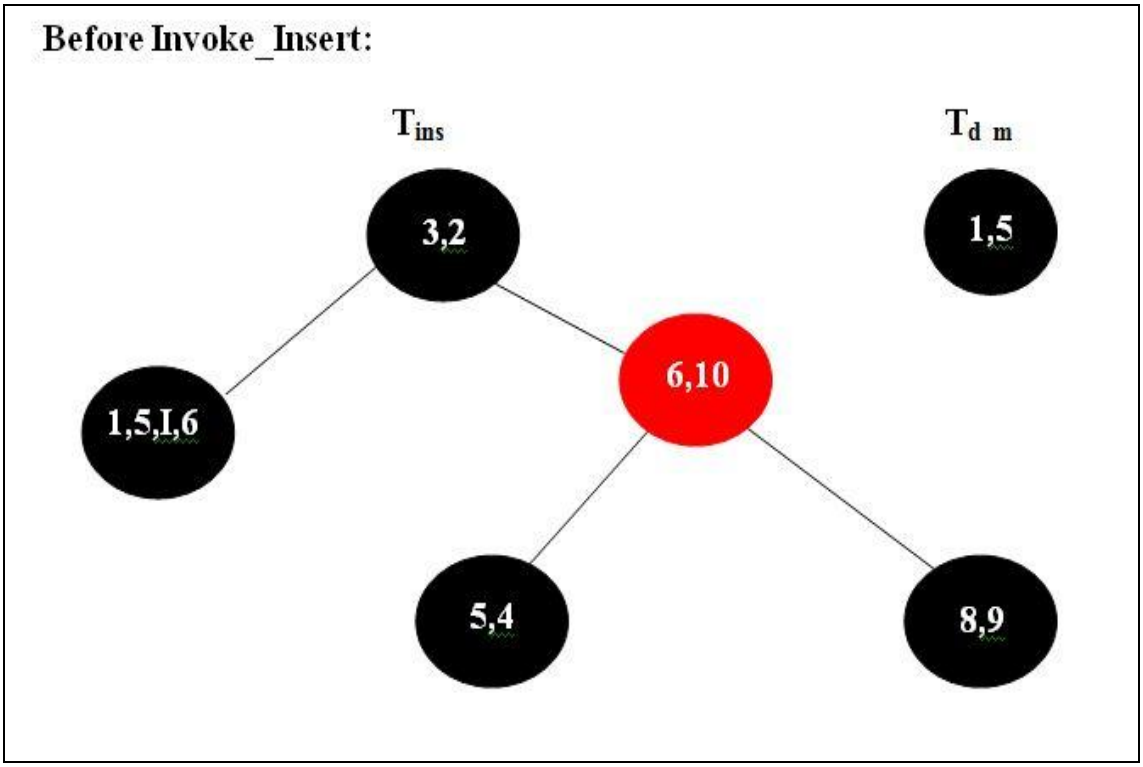

Figure 4.1 Trees T_{ins} and T_{d_m}

i) Invoke _Insert (x, t) performs an insertion of the key (x, t) in to the tree T_{ins} at time *t* according to its priority value. Insertion in the tree is done in following steps:

(i) If *t* is the present time means no *del_min* has been performed after time *t* then search into the tree according to the key value and then according to time to find the location of the key to be inserted. Where the search terminates, attach a new node containing the new key (x, t).

(ii) Else, find the minimum value say *k* in tree T_{d_m} that has been deleted after time *t* and return this as the first inconsistent operation while inserting *x* at time *t* in the priority queue.

```
Invoke_Insert (x, t)
Insert (T_ins, x, t);            // Insert x in to height balanced tree T_ins at time t.
P = Search_Min (T_d_m, t);  //Search in tree T_d_m for minimum key value deleted after time t.
if ( P !=  NULL)
    return (P);                  // Return the first inconsistent operation.
else
    return (NULL);
```

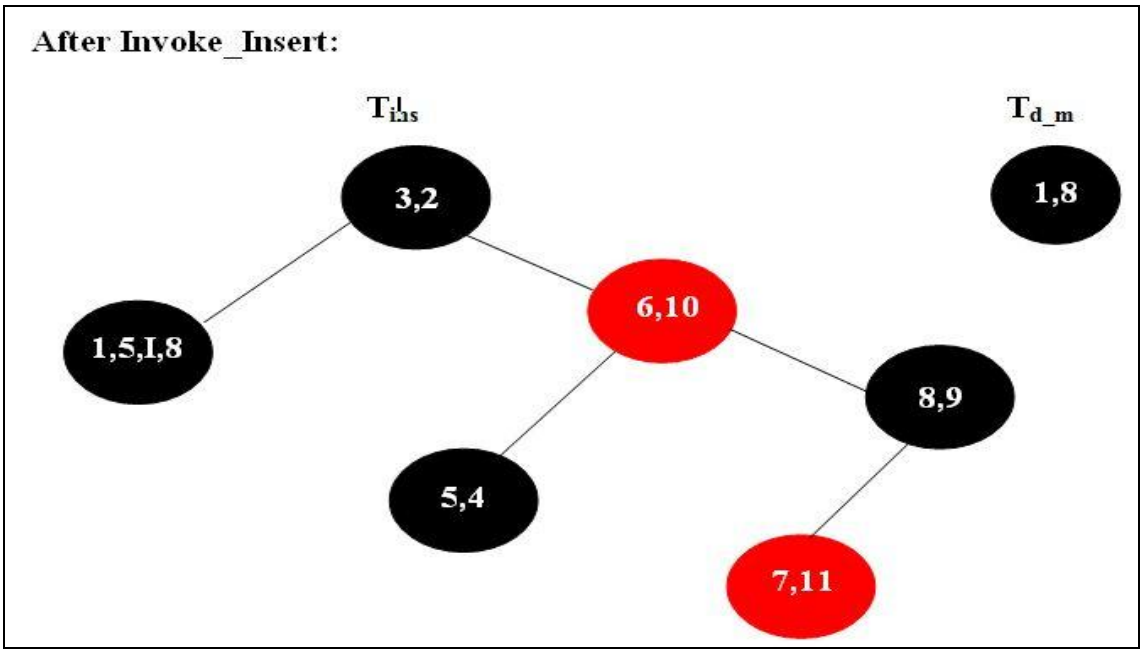

Figure 4.2 Trees T_{ins} and T_{d_m} after Invoke_Insert(7,11) operation in priority queue

ii) Invoke_Del_min (t) performs the deletion of minimum value from the priority queue at time *t*. To perform the deletion, we have to check for the delete minimum operations performed before that time. Steps taken to perform *del_min* are as:

(i) If *del_min* is to be performed at the most recent time, then simply find the minimum key value from T_{ins} and delete it from T_{ins} and insert it into T_{d_m}.

(ii) Else from the tree T_{d_m} find the key value *k´* that has been deleted at a time immediately before given time *t*. Now, from tree T_{ins} find the minimum key *k* greater than key *k´*. *k* is the result of *del_min* operation.

```
Invoke_Del_Min (t))
P = Search (T_d_m, t`);          // where t` = max ( t``ε T_d_m and t``< t)
  if ( P== Null)                 // No del_min has been performed before time t
      N = Find_min(T_ins );
      Return (N);
  else
      K` = P.Data;   // where k` is the maximum value deleted immediately before time t.
      Search (T_ins, k)     // where k = min ( k`` ε T_ins and k``> k`)
```

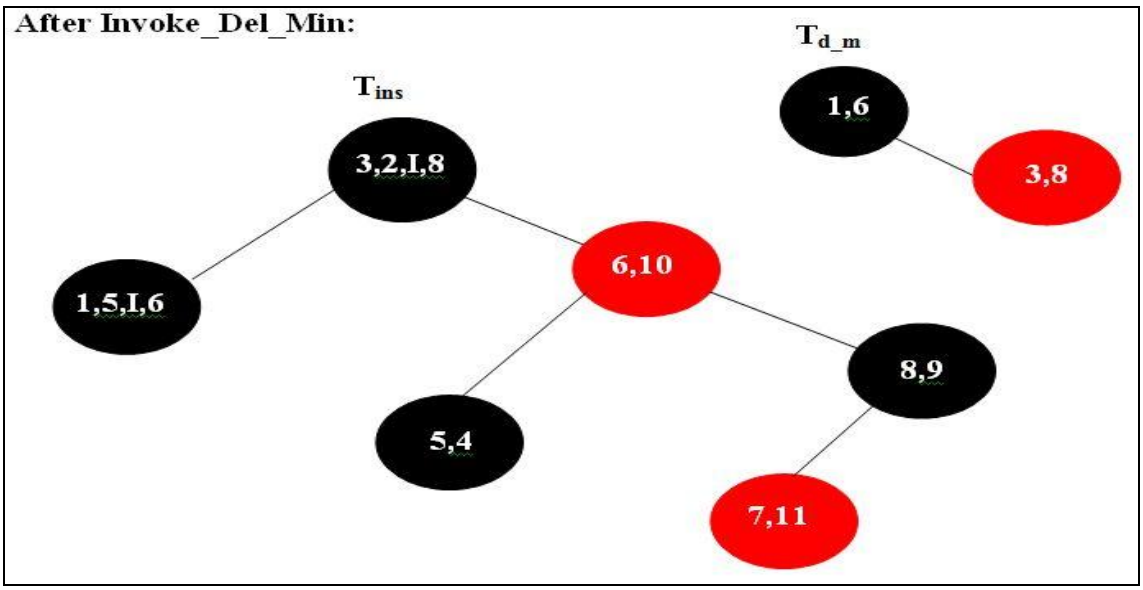

Figure 4.3 Trees T_{ins} and T_{d_m} after Invoke_Del_Min(8) operation in priority queue

iii) Revoke_Insert (t) undo the insert operation performed at time *t*. To do this, check if any *del_min* operation has been performed after this time *t*.

(i) If yes, return the first *del_min* performed after time *t* as the inconsistent operation.

(ii) Else, make the node inserted at time *t* as invalid and find the entry in tree T_{ins} corresponding to time *t* and mark this entry as invalid.

Revoke (Insert (t))

```
P = Search_Min (T_d_m, t);     //Search in tree T_d_m for minimum key value deleted after time t.
    if ( P !=  NULL)
        return (P);            // Return the first inconsistent operation.
    else
        P = Search (T_ins, t);
        P.Valid = False;
```

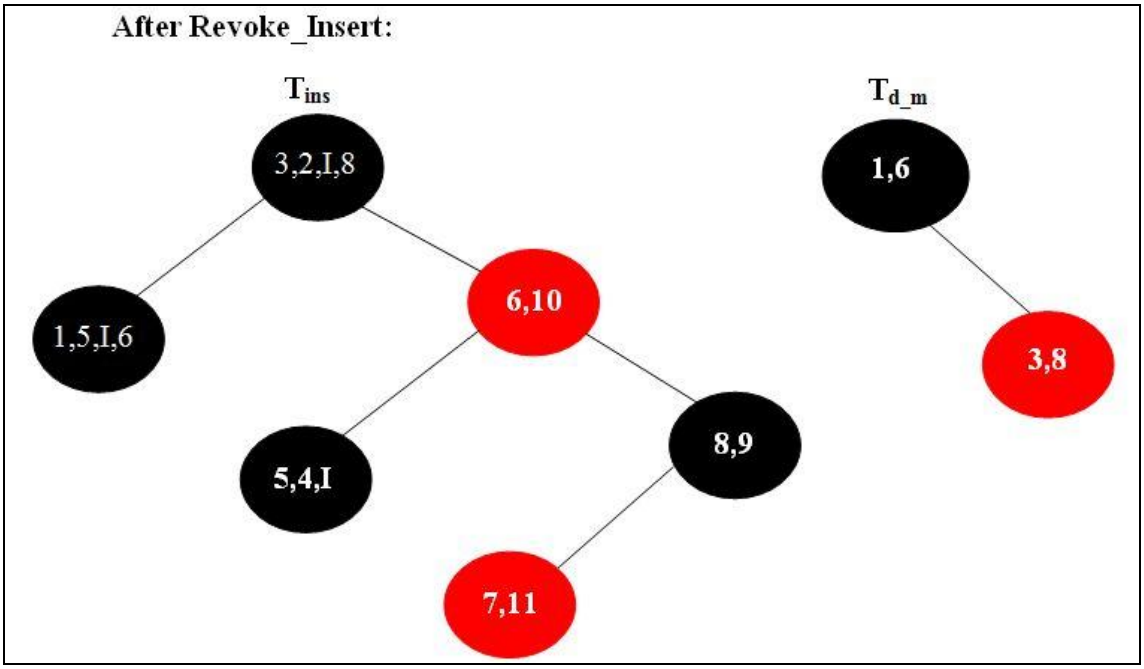

Figure 4.4 Trees T_{ins} and T_{d_m} after Revoke_Insert(4) operation in priority queue

iv) Revoke_Del_Min (t) Undo the delete operation performed at time *t*. This is equivalent to inserting the item deleted at time *t* again in the priority queue.

Revoke (Del_Min (t))

P = Search_Min (T_{d_m}, t); //Search in tree T_{d_m} for minimum key value deleted after time t.

if (P != NULL)

return (P); // Return the first inconsistent operation.

else

Insert (T_{ins}, x, t); // Insert x in to height balanced tree T_{ins} at time t

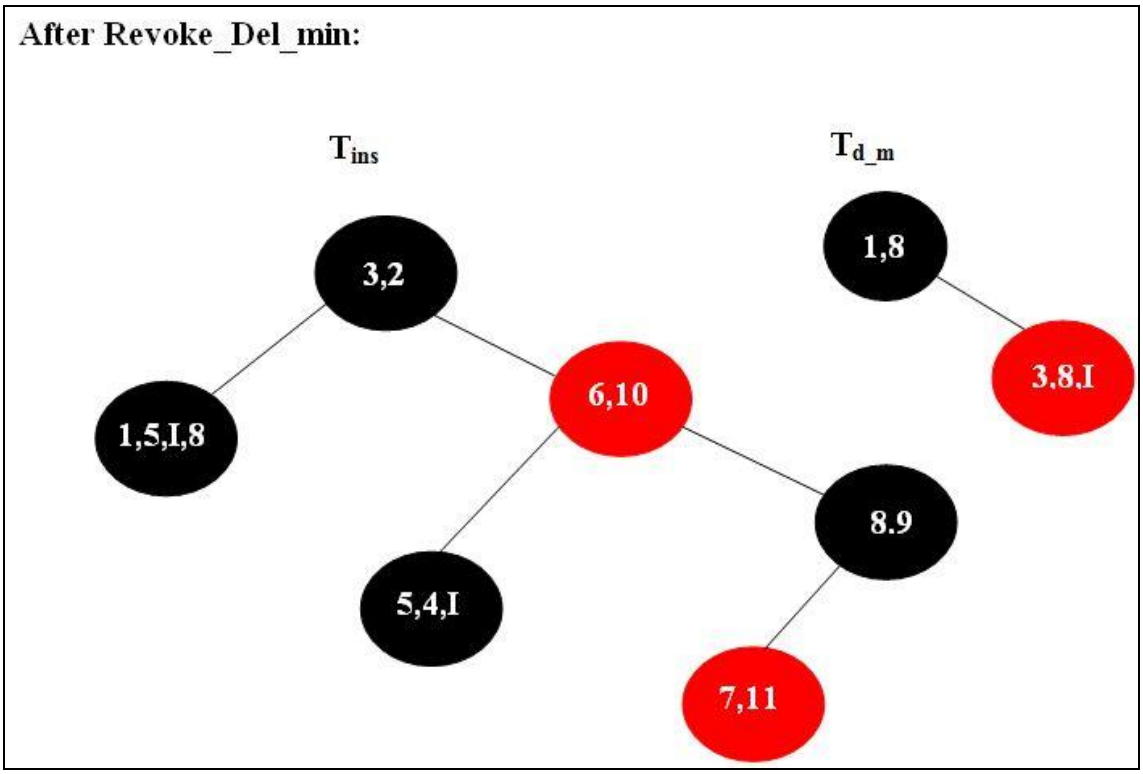

Figure 4.5 Trees T_{ins} and T_{d_m} after Revoke_Del_Min(8) operation in priority queue

v) Find_min(t): Returns the minimum element existing in the priority queue at time *t*. Simply perform the search operation in tree T_{ins} corresponding to time *t*, if *t* is the recent time after which no *del_min* has been performed else find the maximum key value (say *k`* from the tree $T_{d\_m}$ that has been deleted from priority queue before given time *t* and then from tree $T_{ins}$ find the minimum key that is greater than *k`* i.e.

$k`` = \min\{ k \varepsilon T_{ins}$ and k is a valid entry and $k`` > k`\}$.

K`` is the minimum value in the priority queue at time *t*.

Find_min(t)

```
P = Search_Min (T_d_m, t);      //Search in tree T_d_m for minimum key value deleted at time t.
    if ( P = NULL)
        Q = Search_Min(T_ins, t );      // Return the minimum key from T_ins at time t.
         return (Q.val);
    else
         return(P.val);
```

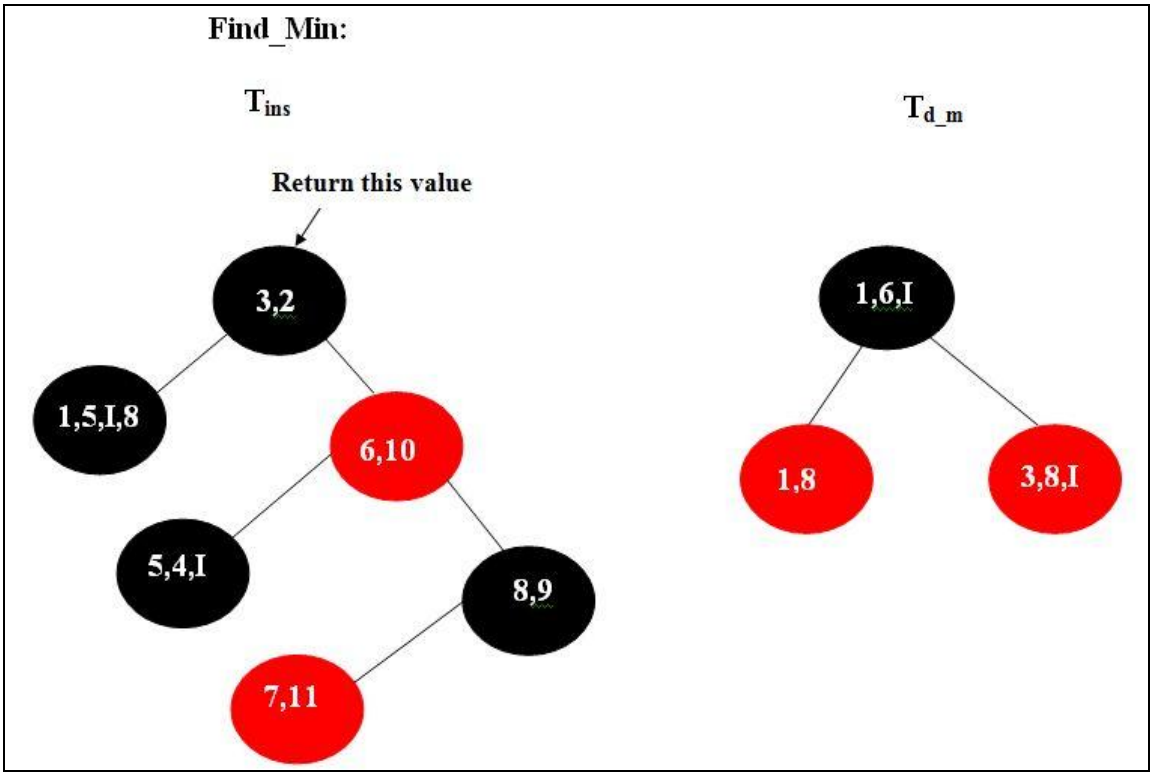

Figure 4.6 Trees T_{ins} and T_{d_m} after Find_Min(13) operation in priority queue

4.4 Dynamizing Dijkstra using Proposed Approach

Dijkstra algorithm can be dynamized using the retroactive priority queue with a slight modification of the basic algorithm. This algorithm consists of applying the standard D*ijkstra's*

algorithm with the modification that the graph is dynamic one and any change in the edge weight of the graph is also input into the priority queue during the algorithm execution. Priority queue accordingly accommodates the change and returns the first operation in the operation sequence that has been affected due to the change in the edge weight.

4.4.1 Dynamic Graph Representation used

A graph is dynamic when some of the graph entities (vertices, edges, and weights) change with time. The most usual time-dependent changes are in the edge weights, which can also model edge connection/disconnection if we allow an arc cost to be infinite. As we are looking into the implementation of dynamic graph algorithm i.e dynamic D*ijkstra*, so we need a dynamic graph representation that efficiently accommodates the dynamic updation of edge weights. We have chosen a simple but effective representation for our purpose. The graph is represented using adjacency matrix which stores graph as a matrix of neighbours (assuming no new vertices are added to the graph). No doubt the representation will be expensive in case the underlying graph is sparse but in this representation any updation can be performed in constant time and thus changing the graph structure is very easy in this representation. Hence, we have chosen the representation so that the time bounds are not affected due to the changes that need to be made on the graph.

4.4.2 Proposed Algorithm-Dynamic_Dijk

In the dynamic *Dijkstra* algorithm we are first checking whether the update operation is effecting the operations performed till now and if yes identify those operations and redo them to accommodate the change.

Step 1: Check whether the head vertex of updated edge is there in the retroactive priority queue (RPQ).

Step 2: If it is not in RPQ (means no previously calculated path will be affected by this change) then go to Step 6.

Step 3: Else If the entry corresponding to head vertex of updated edge is an active entry in the RPQ (the vertex is not used in any of the existing shortest paths) , then go to Step 4, Else go to Step 5.

Step 4: If update is negative (i.e. weight of edge has been decreased), then update the distance of tail vertex of updated edge and go to step 1, Else go to Step 6.

Step 5: Else move in the priority queue immediately before the time that edge has been deleted and accommodate the change as required. Go to Step 1.

Step 6: Exit

Further, these steps are described in detail through a flow chart (Figure 4.7) as well as pseudocode.

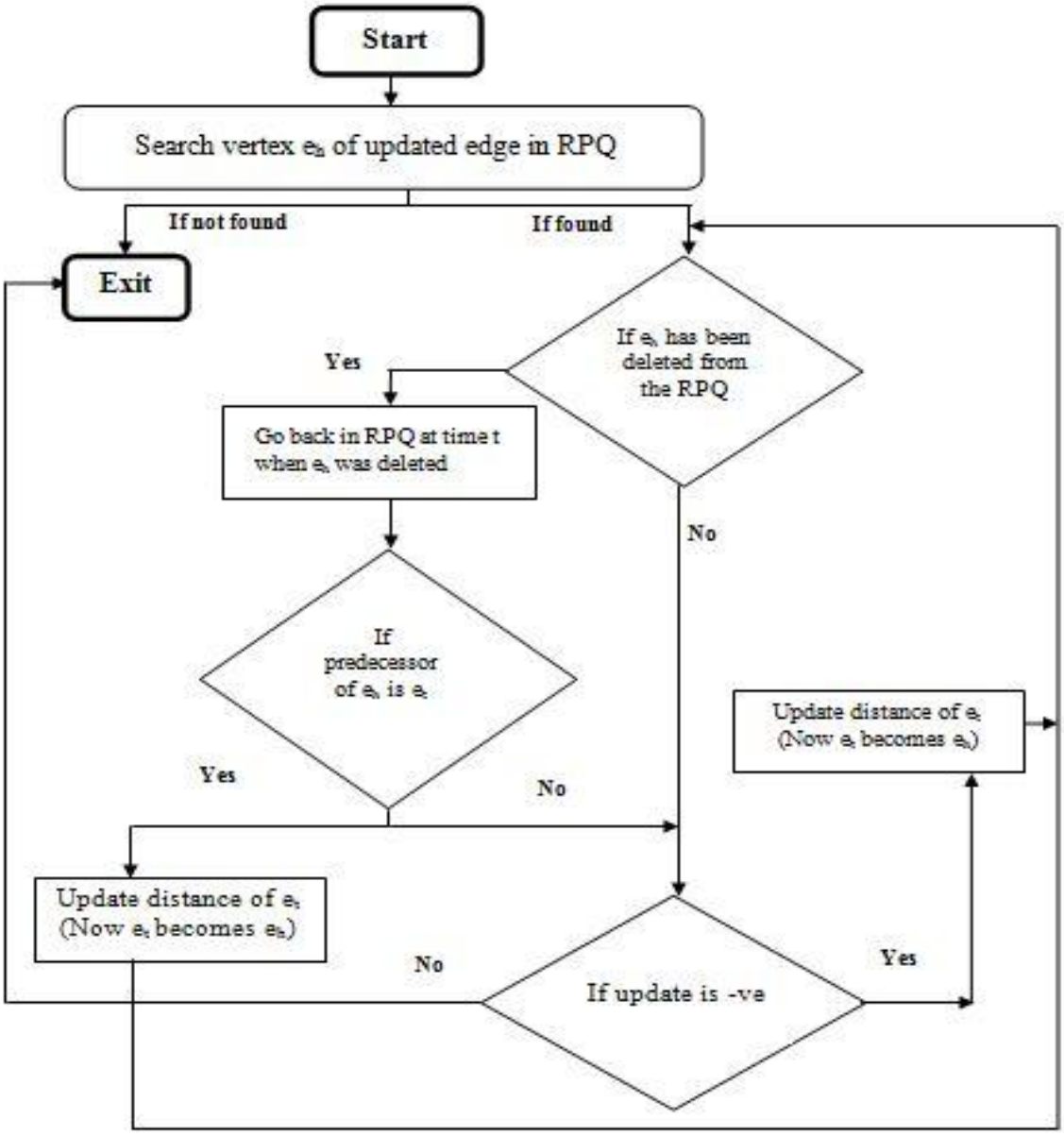

Figure 4.7 Flow chart of Dynamic Dijkstra (D_Dij)

Dynamic_Dijk

// Weight of edge e has changed by say α(increased) or -α(decreased)

1: N = Search(T_{ins}, e_h); // e_h denotes the head vertex of edge e

2: if (N == NULL) then

3: Exit // No change in shortest paths, proceed normally

4: else

if N is active entry in the priority queue then // N has not been deleted yet

5: if pred[e_h] == e_t then

6: d[e_h] = d[e_t] + α

7: end if

8: else

Move in the priority queue before time when e_h was deleted say node m.

9: If (pred(e_h) == e_t) then

10: m. value = d[e_h] + α // change the value of node m.

11: retroactive priority queue returns the del-min operation effected by this change say node m.

/* Relax outgoing edges of the vertex of node m (m.v).*/

12: for each $e` \varepsilon m.v$ do

13: if (pred [$e`_h$] == e_h) then

14: d[$e`_h$] = ∞

15: Relax edge e`

if relaxed insert(T_{ins} ,t.val)

16: insertion returns the next operation effected in the retroactive priority queue say node m.

17: end if

18: end for

19: end if

20: end if

21: end if

4.4.3 Example

Using the graph of Figure 4.8 we are showing the actions performed by our algorithm when edge weights of the edges are allowed to change (i.e. increase as well as decrease). Each row below represents the state of the priority queue at time t i.e. each entry in the priority queue is a valid entry in the queue at that specific time.

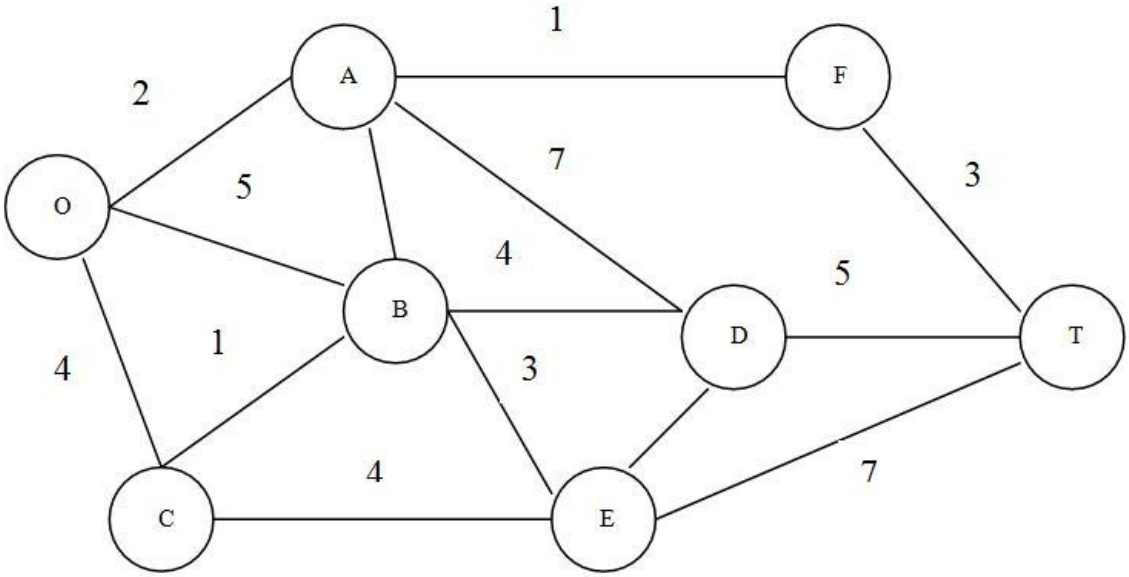

Figure 4.8 Example Graph for Dynamic Dijkstra Algorithm

Each entry in the queue is a triplet (v, val, pred) where v denotes the vertex, val denotes predicted distance of that vertex from source vertex and pred gives the predecessor of that vertex on the predicted shortest path.

Time	**Priority_Queue**			
t=0	O,0,Nil			
t=1	A,2,O	C,4,O	B,5,O	
t=2	C,4,O	B,4,A	D,9,A	F,14,A

t=3	B,4,A	E,8,C	D,9,A	F,14,A

t=4	E,7,B	D,8,B	F,14,A

At time t = 5 the weight of edge BD is increased to 5.

As Vertex D is still in the Priority queue that means edge BD has not been included in any shortest path till now. So, no change is required, proceed normally.

t=6	D,8,B	F,14,A	T,14,E

At time t = 7 the weight of edge OB is increased to 7.

As vertex B has been previously in the priority queue, we need to follow the paths in which edge OB has been used, so that they can be updated accordingly. Move in the priority queue at time immediately before the deletion of vertex B i.e. at time 3. We need not change the value of shortest path of B as it is less than the new one including the changed weight for edge OB. So, there will be no change in shortest paths in this case also and we proceed normally.

t=8	T,13,D	F,14,A

At time t = 9 the weight of edge AB is decreased to 1.

As vertex B has been previously in the priority queue, move in the priority queue at time immediately before the deletion of vertex B i.e. at time 3.Predecessor of B at that point is A, so its shortest path becomes 3. As value in the retroactive priority queue has changed, it will automatically rearrange itself and the return the operation effected by this change. At this point priority queue can be represented as:

t=3	B,3,A	E,8,C	D,9,A	F,14,A

So, *del_min* for will be again performed with new value of distance for vertex B.

t=10	E,6,B	D,7,B	T,13,D	F,14,A

Next operation effected is for vertex E. Now perform *del_min* for E.

t=11	D,7,B	T,13,D	F,14,A

Next operation affected is for vertex D. Perform *del_min* for D.

t=12	T,12,D	F,14,A

Now, we have reached to the point at which normal execution of *Dijkstra* should resume as all changes have been propagated completely.

4.5 Complexity Analysis

We have proposed to use the retroactive data structuring paradigm to represent the time-line required to be associated with each operation in the dynamic shortest path problem. Our structure has *O(log T)* access time as well as update time, where *T* is the number of valid nodes in the retroactive priority queue .The space bound is proportional to the total number of operations performed on the data structure till now i.e *O(N)* .Our resource bounds match those of *Ramalingam and Reps* [114], but our data structure is on-line and is simple enough to have potential practical applications.

Our method provides an efficient solution to the dynamic shortest path problem. As we are implementing the retroactive priority queue using the height balanced search trees, so the cost of all the operations of priority queue is dependent on the height of the underlying tree data structure. Also, the no. of computations is less in case of dynamic changes in the graph as we are moving back in time at a point immediately before the time when the edge under updation is going to be used in any of the shortest paths. We are selectively choosing the point which has been actually affected by the change and applying the updation to that portion only. The main limitation of our method is that we are considering a single change in the graph at a time, if a number of changes occur simultaneously then these will be treated as a sequence of changes which affects the computational cost of the system.

4.6 Empirical Analysis

We have also done the empirical analysis by comparing proposed algorithm with two practically efficient algorithms. Here, the performance of our Dynamic Dijkstra (D_Dijk) algorithm has been compared with the algorithms given by *Ramalingam and Reps* [113] and *Demetrescu et al.* [35]. The algorithm by *Ramalingam et al.* [113] is known to be very fast in practice [51, 52, 53], it works by identifying the subset of vertices whose distance from source *s* is affected by the update. Distances are then updated by running a Dijkstra algorithm on those nodes. The algorithm by *Demetrescu et al.* [35] uses the idea of uniform paths in Dijkstra-like algorithm so that the number of total edge scans can be reduced and performance bottleneck can be overcome. We have also compared our proposed algorithm with the heap reduction technique given by *Buriol et al.* [17] applied to the algorithms of *Ramalingam and Reps* [113] and *Demetrescu et al.* [35].

The experiments were performed on a 2.30 GHz Intel Pentium III processor with 2 GB of RAM. The implementation or simulation has been performed using C Language (compiled with gcc- 4.8.2 using the -O3 optimization option). CPU times were measured with the system function and memory profiling was done using tool Valgrind.

In our experiments, we have considered two types of graphs. One is random graph data sets as described in Table 4.2. Random graphs with *n* nodes, *m* edges and random integer edge weights are generated. Update sequence is also generated randomly by selecting any of the update operation on any random edge. Another graph data set has been taken from the experimental package available at the *http://www.dis.uniroma1.it/~demetres/experim/dsp/* provided by *Demetrescu et al.* [36]. The graph data used for experimentation is available in the package as XML files corresponds to the real world US road networks data obtained from ftp://edcftp.cr.usgs.gov. The table 4.3 describes the details of US Road Networks graphs from the package used for experimentation.

Table 4.2 Experimental Data Set: Random Graphs

Graph	Number of Vertices |v|	Number of Edges |E|
R_G1	100	500
R_G2	250	1250
R_G3	325	1625
R_G4	515	2575
R_G5	625	3125
R_G6	735	3675
R_G7	800	4000
R_G8	885	4425
R_G9	950	4750

Table 4.3 Experimental Data Set: US Road Network

Graph	XML File	Number of Vertices |v|	Number of Edges |E|
Graph-1	RI.xml	170	490
Graph-2	ID.xml	256	676
Graph-3	SD.xml	396	1132
Graph-4	MD.xml	490	1390
Graph-5	IN.xml	599	1890
Graph-6	MI.xml	695	2162
Graph-7	MO.xml	809	2510
Graph-8	CA.xml	945	2768
Graph-9	FL.xml	1368	3892

Further, the performance of the proposed Dynamic Dijkstra-(D-Dijk) algorithm has been compared with the algorithms D_Rr (given by *Ramalingam et al.* [113]), D_Up (given by *Demetrescu et al.* [35]) and D_Rr_B & D_Up_B (based on heap reduction technique given by *Buriol et al.* [17]). The comparative results obtained through experimentations on these five three algorithms on two different types of graphs have been shown through Figure 4.9 to 4.14.

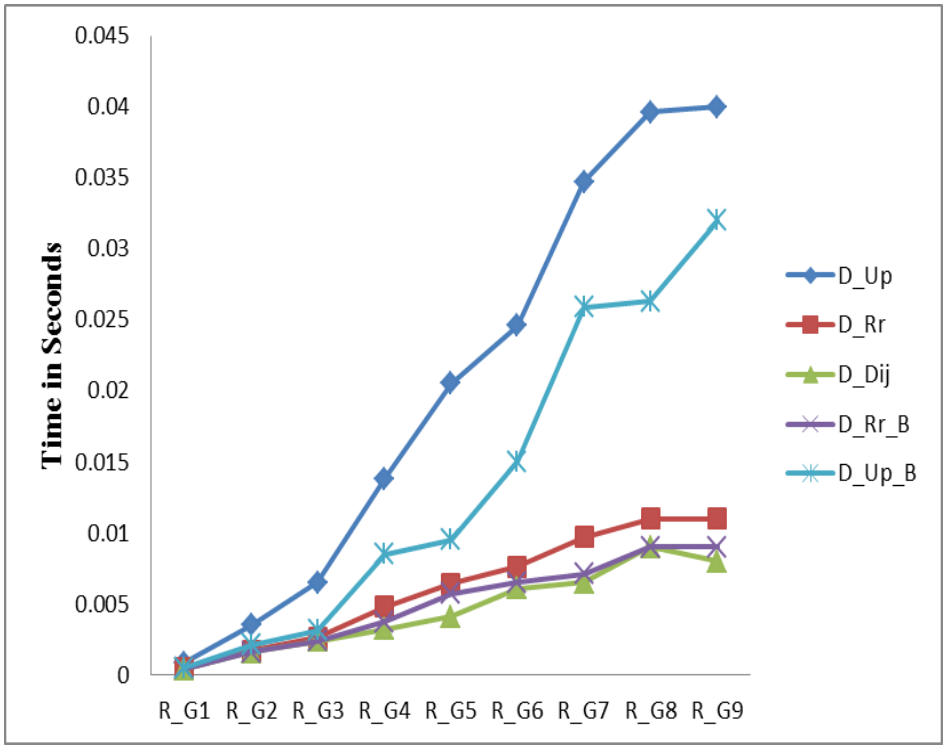

Figure 4.9 Comparison time taken by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dijk for Random Graphs

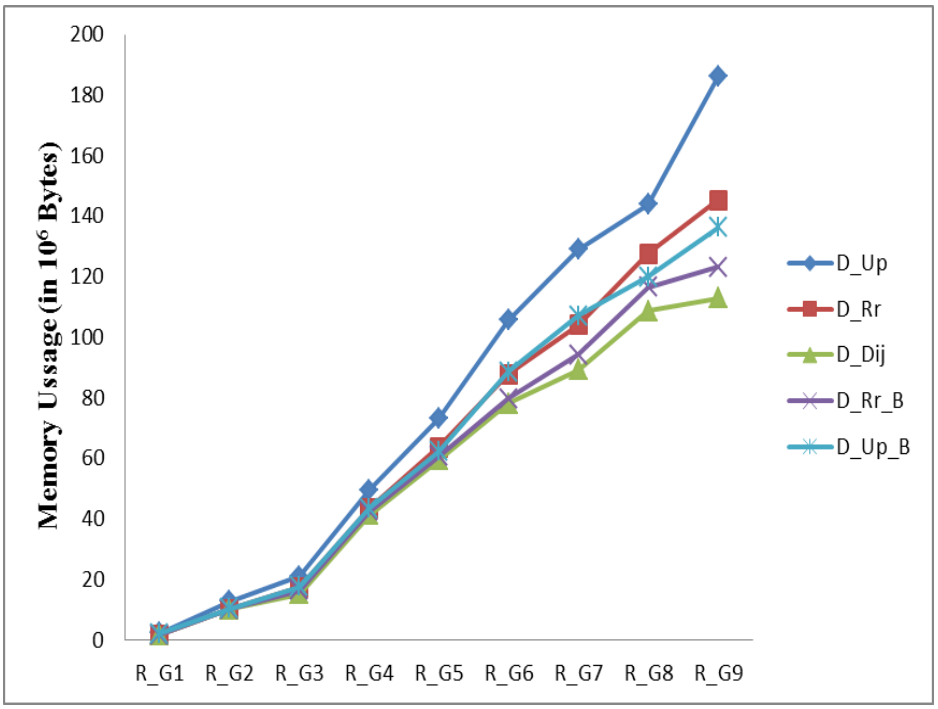

Figure 4.10 Comparison Memory usage by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dijk for Random Graphs

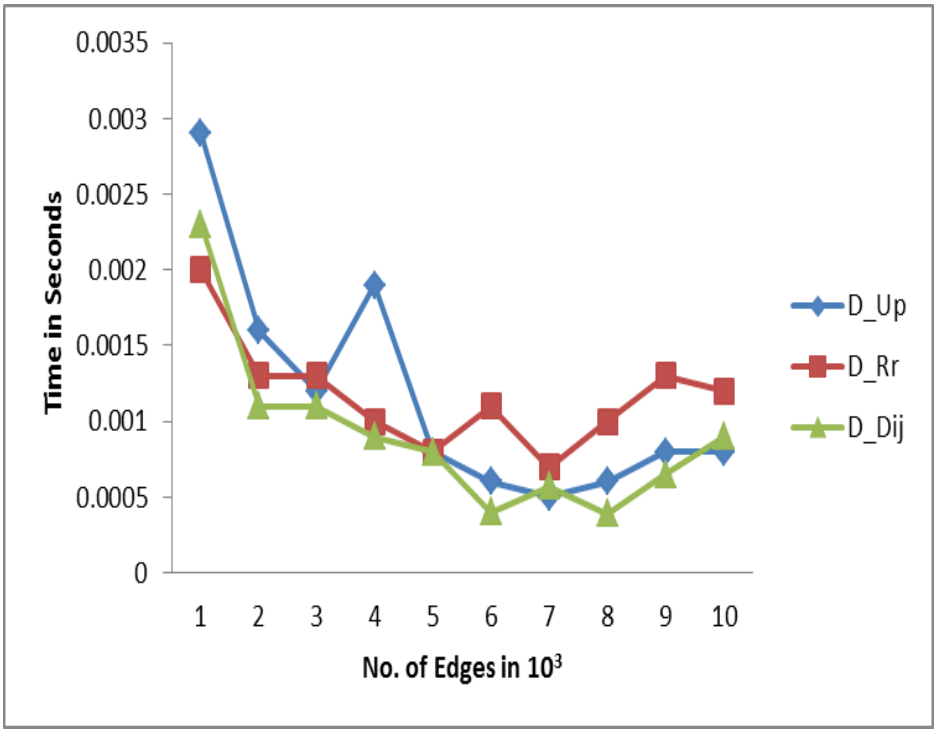

Figure 4.11 Comparison time taken by D_Up, D_Rr and D_Dijk for varying number of edges

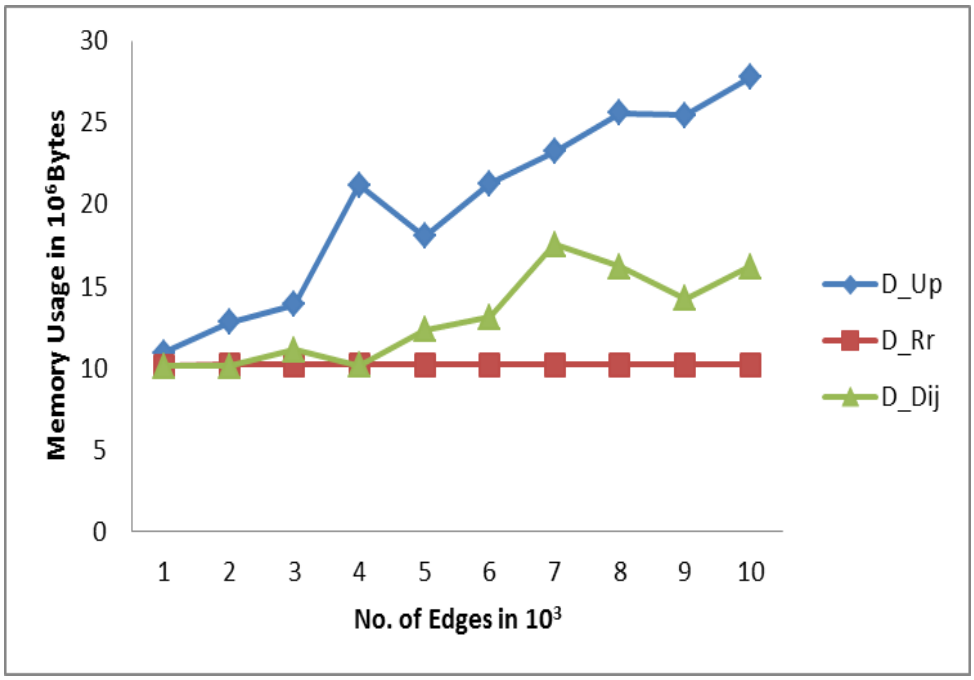

Figure 4.12 Comparison memory usage by D_Up, D_Rr and D_Dijk for varying number of edges

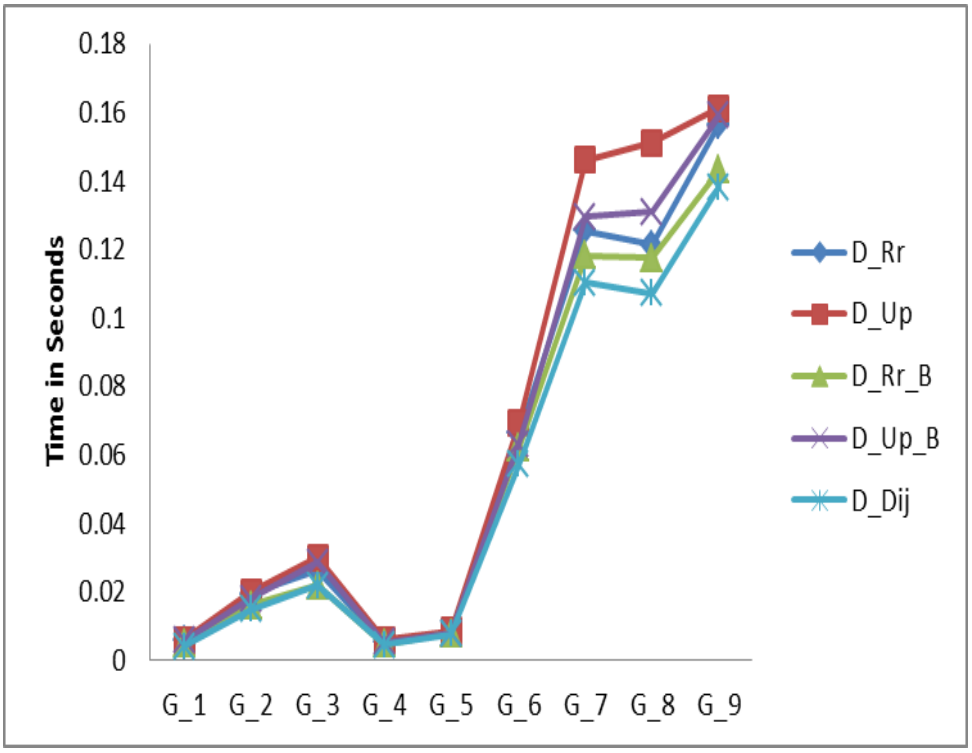

Figure 4.13 Comparison time taken by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dijk for US Road Network Data

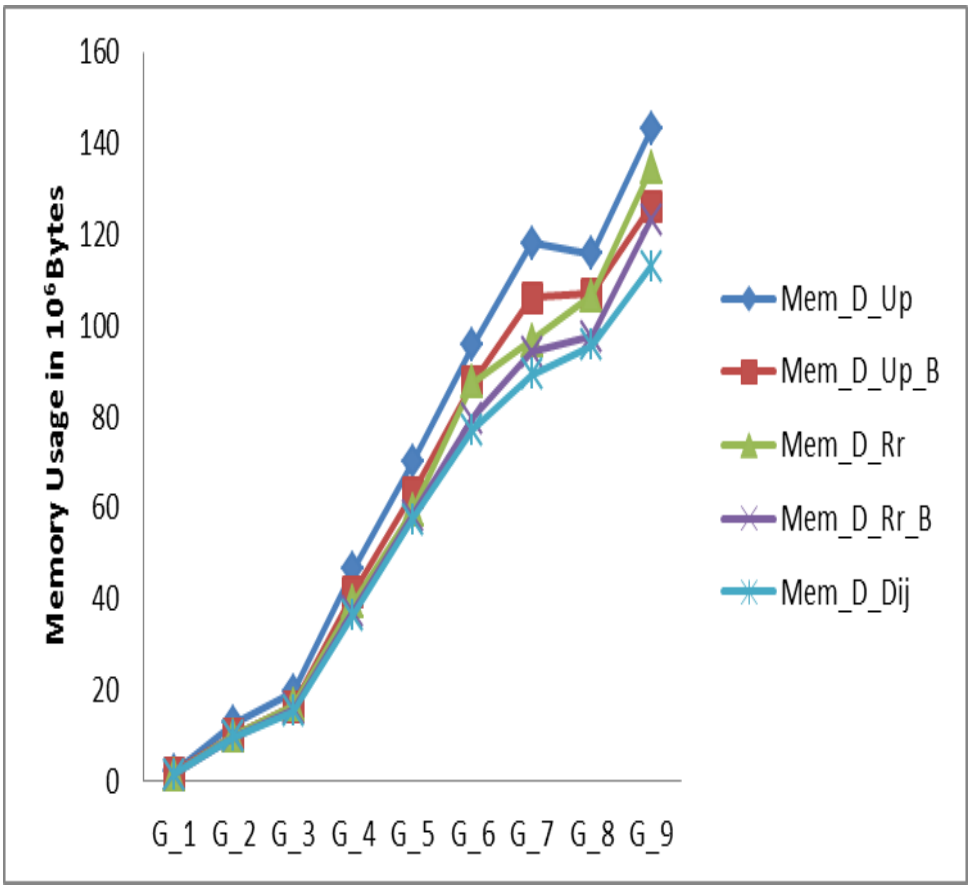

Figure 4.14 Comparison Memory usage by D_Up, D_Rr, D_Up_B, D_Rr_B and D_Dijk for US Road Netwok Data

Experiments with random graphs as well as US Road Network graphs point out that D_Dijk has performance (Figure 4.9 and Figure 4.13) comparable to that of D_Rr depending on the input graph topology, for the sparse graphs. Also memory usage (Figure 4.10 and Figure 4.14) for the proposed algorithm D_Dijk is lesser as compared to all the other algorithms, because proposed algorithm does not use any extra information for performing the dynamic updations. But D_Rr maintain the extra information like the subset of effected vertices and D_Up also maintains the uniform paths for performing the dynamic updations. The memory usage decreases by using the heap reduction technique leading to lesser memory requiremnts of D_Rr_B and D_Up_B.

As is visible from the results (Figure 4.9) the performance of the proposed algorithm D_Dijk is better than both other algorithms and also the time taken by the algorithms D_Rr and D_Up increases with increase in the underlying graph size (number of edges) due to increase in overhead in the D_Rr and D_Up algorithms. Moreover, Figure 4.12 shows that memory usage in case of D_Rr is not much effected by the graph size but in case of D_Up memory usage increases significantly with increase in number of edges. For the proposed algorithm D_Dijk, memory usage does not grow much with the increase in number of edges of the graph.

4.7 Conclusion

We have proposed to use the retroactive data structuring paradigm to represent the time-line required to be associated with each operation in the dynamic shortest path problem. Our method provides an efficient solution to the dynamic shortest path problem. As we are implementing the retroactive priority queue using the height balanced search trees, so the cost of all the operations of priority queue is dependent on the height of the underlying tree data structure. Also, the no. of computations is less in case of dynamic changes in the graph as we are moving back in time at a point immediately before the time when the edge under updation is going to be used in any of the shortest paths. We are selectively choosing the point which has been actually affected by the change and applying the updation to that portion only.

The solution of shortest path problem using retroactive data structures can be easily adapted to relax the assumption we have made in our work. Transformations (Reweighing) can be applied to adapt our solution in case of general graphs (graphs with positive or negative edge weights). Also, our dynamic Dijkstra algorithm can be used to solve the dynamic all-pair shortest-path problem. The performance of the solution may be somewhat less than the best resource bounds known so far. Also, improving the efficiency of the underlying retroactive priority queue may further improve the performance of the approach. In future, we will plan to process a number of update operations as a single one, this may further reduce the time bounds for the problem.

CHAPTER-5

Proposed Improved Algorithm for Indexing Massive Data: Extended Suffix Array Construction using Lyndon Factorization

In this chapter, we are proposing a method that constructs Extended Suffix Array that is the basic component of almost all the full-text indexes used for indexing massive data. This approach is based on the combinatorial relationship between the construction of suffix array and burrows-wheeler transform established based on the Lyndon factorization of text. Lyndon factorization of the text is used to divide the text into blocks for incremental construction. First of all, we give an overview of the extended suffix array and its existing construction methods. Then Lyndon factorization is explained. After that, we move on to the approach used for constructing suffix array using Lyndon factorization. Then the proposed approach for extended suffix array construction using Lyndon factorization is explained and analysed. Finally, conclusion is given along with the future perspectives.

5.1 Introduction of Index structures for Massive Data

Starting from the suffix tree data structure, there have been many breakthroughs in the text processing data structures like suffix array by *Manber and Myers* in 1990 [91], enhanced suffix array [2] and extended suffix array [120]. Further improving the capabilities of suffix array data structure led to a new class of data structure- abstract data structures like compressed suffix array [44, 45, 63, 72, 89] and then compressed suffix trees [20, 49, 118, 133] with many variations in their implementations. The research in this class of data structures has been oriented towards finding an optimal space/time trade-off in their implementation either during the construction or in their use as an indexing data structure. Space/time trade-off is a major concern during the construction phase, as the size of the text which needs to be indexed is increasing day by day far exceeding the memory capacity of today's computers. At the core of all these data structures is

the suffix array which is invariably used in one form or the other in all of these variants. Also, in almost all the variants, suffix array is used with additional information like longest common prefix (LCP) array to further strengthen the index structure. The suffix array along with the LCP array is called the extended suffix array. Extended suffix array can effectively replace a suffix tree in the applications which also need some topological information of the suffix tree. The LCP array along with providing topological information of the suffix tree also helps in efficient pattern matching using suffix array[69, 135], fast disk based suffix array arrangements [43, 128] and compressed suffix trees [49].

5.2 Index Construction Methods

First step in the construction of any full-text index structure is the construction of suffix array (SA). After constructing suffix array all the other components are constructed based on the SA, like BWT and LCP. So, the construction efficiency of all these approaches is dependent on the efficiency of SA construction. There are many approaches for SA construction which can be classified depending on the type of memory used for storing the text to be indexed while construction. There are two main types of construction algorithms: internal memory construction algorithms and external memory construction algorithms. Internal memory algorithms are the ones in which the whole text to be indexed is in the internal memory of the system during the construction. The main drawback of these algorithms is that these can't be used for indexing texts of very large sizes due to limited internal memory of the systems. To overcome this problem researcher's developed external memory algorithms. In these algorithms the text to be indexed is stored in external memory during the construction. Most of the external memory algorithms are based on incremental approach by *Karkkainen* [78]. In the incremental approach, the whole text to be indexed is divided into blocks. The text is stored in external memory. A block of text is transferred from external memory to internal memory and its partial SA is constructed. Now, that block of text is removed and next block is transferred from external memory. SA for that text block is constructed and merged with the previous block's SA. And the

procedure is repeated for all the remaining blocks. This incremental approach significantly reduces the working space. But the merging step of the approach is a big hindrance to the scalability of these algorithms.

5.3 Extended Suffix Array: Index Structure for Massive Data

As we have seen in section 5.1, in case we need to index massive data, constructing extended suffix array as the basic component of the final index leads to lesser construction time. Construction of the extended suffix array can be done in two ways: First method is to construct the suffix array and after the construction of suffix array is complete, use it to construct the LCP array using any of the existing methods. Time complexity in this case is *O(nlogn)* for the suffix array construction and then *O(n)* for computing LCP array from suffix array [80]. Another way is to incrementally construct the suffix array by dividing the text into blocks of suitable size and along with suffix array also compute the LCP array side by side. This is a good idea to construct the LCP as a by-product of the suffix array construction as sorting the suffixes of a text involves breaking ties after the common prefixes. Also, this method is space efficient as construction is done block by block. In this chapter, we are going to introduce a new strategy for the incremental construction of the extended suffix array using Lyndon Factorization that opens new scenarios for the space efficient construction of the different variants of suffix array like extended suffix array, compressed suffix array etc.

5.4 Lyndon Factorization

Lyndon Factorization was introduced by Chen et al. [24] as the factorization of a sequence of an ordered set and later *Duval* in [40] gave a linear time algorithm to find this factorization for a word. Since then it has been used in different applications like Digital Geometry [15, 70], description of Free Lie Algebra [12], and Efficient Construction of particular de Bruijn sequence in linear time [116, 122]. In Lyndon factorization, any string can be decomposed uniquely into a sequence of lexicographically non increasing factors i.e. $L_1L_2L_3\ldots\ldots L_k$, with the condition that

$L_1 \geq L_2 \geq L_3 \geq \ldots\ldots \geq L_k$. Each factor L_i is called as a Lyndon word and has the property that each L_i is lexicographically least among its own circular shifts. For each factor L_p, let the position of the first character in that Lyndon factor is represented by F_L_p and the position of the last character of each factor L_r is given by F_L_{p+1}-1 (character preceding the first character of the next Lyndon factor). So, only the position of first character of each Lyndon factor needs to be stored. End of a Lyndon factor can be defined using the first character of the succeeding Lyndon factor. For the text *T= bananaanaa$* the Lyndon factorization is:

$b(L_1)$, $an(L_2)$, $an(L_3)$, $aan(L_4)$, $a(L_5)$, $a(L_6)$.

Each $L_i \geq L_j$ for all i<j, as $b > an \geq an > aan > a \geq a$.

Also, each L_i is least among its cyclic shifts.

5.5 Suffix Array Construction using Lyndon Factorization

Bonomo et al. [14] gave the idea of using the Lyndon factorization of a text for the computation of the suffix array of a text. The main advantage of using Lyndon Factorization for incremental construction of suffix array is that it helps to reduce the merging time required for the incremental construction. Before starting with the detailed description of the approach, let us briefly review the basic concepts and notations to be used throughout this chapter. Let us consider a text *T* of size *n* denoted as *T[0…n-1]*. All the characters of the text are from an ordered alphabet $\sum$ of size σ. The text *T* is appended with the character *$* which is smallest of all the characters in $\sum$ and it also doesn't appear anywhere in the text. The suffix of a text is represented as $suf_i(T)$ which is the suffix of text *T* starting at position *i* i.e. *T[i….n]*. A text of size *n* can have *n* suffixes. Lexicographic order of the suffixes of text is established based on the lexicographic order of the characters of the individual suffixes as: let $suf_i(T)$ and $suf_j(T)$ be two suffixes then $suf_i(T) < suf_j(T)$ if

$$T[i] < T[j]$$

Or if $T[i] = T[j]$ and $suf_{i+1}(T) < suf_{j+1}(T)$ for all $i <= n$ and $j <= n$

An array representing all the suffixes of the text in their lexicographic order is called as the suffix array (SA) [92]. So, suffix at position *i* in the SA is i^{th} smallest suffix of text starting at position SA[i] in the text. Alternatively, $suf_{SA[0]} < suf_{SA[1]} < \ldots\ldots < suf_{SA[n]}$. As each suffix array entry is a position of the corresponding text so, can be represented in *O(log n)* bits and the whole suffix array takes *O(n log n)* bits, where *n* is the size of the text.

The burrows–wheeler transform (BWT) [19] is a permutation of the text with some properties that make it suitable for compression as well as other text processing applications. BWT of text *T* of length *n* is an array bwt[0..n-1] where bwt[i] is the last character of the i^{th} smallest cyclic shift of the text *T*. Sorted cyclic shifts of a text appended with the smallest character (not existing in the text) also give the sorted order of the suffixes i.e. suffix array of that text. So, the first column (F) of the sorted cyclic shifts represents the first character of the sorted suffixes and the last column (L) corresponds to BWT. There is a function LF() which gives the correspondence between the characters of the columns L and F. This LF() function helps to reconstruct the text from its BWT.

An array representing the longest common prefixes of all pairs of adjacent suffixes in SA is called as longest common prefix (LCP) array. Using LCP array as an auxiliary data structure with the SA speeds up querying in to the text. LCP array is an array of size *n* whose entries are defined as:

$$LCP[i] = \begin{cases} -1 & \text{if } i=0 \\ lcp(suf_{SA[i-1]}, suf_{SA[i]}) & 0< i <= n \end{cases}$$

Querying into the text using only SA takes *O(m log n)* time where *m* is the size of the query text. This time can be reduced to *O(m+log n)* by using the LCP array [92].

i	**SA[i]**	**LCP[i]**	**Sorted Suffixes / Conjugate**	**BWT[i]**	**LF(i)**
0	10	-1	**$**bananaanaa	a	1

1	9	0	a$bananaana	a	2
2	8	1	aa$bananaan	n	8
3	5	2	aanaa$banan	n	9
4	6	1	anaa$banana	a	3
5	3	4	anaanaa$ban	n	10
6	1	3	ananaanaa$b	b	7
7	0	0	bananaanaa$	$	0
8	7	0	naa$bananaa	a	4
9	4	3	naanaa$bana	a	5
10	2	2	nanaanaa$ba	a	6

Figure 5.1 Suffix Array, LCP, Sorted Suffixes/Conjugates, BWT, LF() for text bananaanaa$

5.5.1 Suffix Sorting and Lyndon Factorization

Lyndon factorization of a text gives a sequence of factors such that each factor in the sequence is succeeded by a factor which is smaller than it, so lexicographic ordering of suffixes and conjugates of a Lyndon factor also coincide. Based on the above definition, the following lemmas have been given by *Bonomo et al.* [14] which define the relationship between the suffixes of a text and the corresponding conjugates and among the suffixes of adjacent Lyndon factors.

Lemma 1:

Let $T \in \sum$ and let $T = L_1L_2 \ldots L_k$ be its Lyndon factorization. For any two suffixes i and j of a Lyndon factor, if $suf_i < suf_j$ then the corresponding conjugates $conj_{n-i} < conj_{n-j}$.

Lemma 2:

Lexicographic order of suffixes $suf_i(L_m)$ of a Lyndon factor gives the lexicographic order of the corresponding suffixes of the text i.e. for positions i and j in Lyndon factor L_m if $suf_i(L_m) < suf_j(L_m)$ then $suf_i(T) < suf_i(T)$

PROOF:

According to the property of Lyndon factors - A Lyndon factor L_r is lexicographically smaller than all the suffixes of Lyndon factors L_1 to L_{r-1}. So, each Lyndon factor in the Lyndon factorization as well as all of its suffixes is succeeded by a smallest suffix. Suffix *i* of Lyndon factor *m* can be represented as:

$suf_i(L_m) = T[F\_L_m+i\ldots\ldots F\_L_{m+1}-1]$

and suffix *i* of text *T* is:

$suf_i(T) = T[F\_L_m+i\ldots\ldots F\_L_{m+1}-1]\ L_{m+1}\ L_{m+2\ldots\ldots}L_k$

Equivalently, $suf_i(T) = suf_i(L_m)\ L_{m+1}\ L_{m+2\ldots\ldots}L_k$

This signifies that the lexicographic order of suffixes of a text is equivalent to the lexicographic order of corresponding suffixes in Lyndon factor. □

Next, we show the relationship between the conjugates for a sequence of Lyndon factors.

Theorem 1:

Let $u = L_rL_{r+1}\ldots\ldots L_k$ be a sequence of consecutive Lyndon factors and L_i and L_j be two Lyndon factors in u such that $L_i < L_j$ means $i>j$. Then, the sorting of the conjugates of u coincides with the sorting of suffixes of u.

PROOF:

Let $suf_{n-h}(u)$ be a suffix of *u* starting at index *(n-h)* in the text which belongs to Lyndon factor L_i and $suf_{n-k}(u)$ be a suffix of *u* starting at index *(n-k)* position occurring in Lyndon factor L_j. Corresponding conjugates be $conj_h(u)$ and $conj_k(u)$.

$suf_{n-h}(u) = suf_{n-h}(L_i)L_{i+1}L_{i+2}\ldots L_k$

$suf_{n-k}(u) = suf_{n-k}(L_j)L_{j+1}L_{j+2}\ldots L_i\ldots\ L_k$

By the property of Lyndon factorization, if $L_i < L_j$ then $L_{i+1} < L_{j+1}$ so, the lexicographic order of $suf_{n-h}(u)$ and $suf_{n-k}(u)$ is same as that of $suf_{n-h}(L_i)$ and $suf_{n-k}(L_j)$ (As in lemma 3.2). Also, conjugates of *u* can be represented as:

$$conj_h(u) = T[n-h....F\_L_{i+1}-1]L_{i+1}L_{i+2}\ldots L_kL_1L_2\ldots.L_{i-1}\,T[F\_L_i\ \ldots\ldots n-h-1]$$

Equivalently, $conj_h(u) = suf_{n-h}(u)\ L_1L_2\ldots.L_{i-1}\,T[F\_L_i\ \ldots\ldots n-h-1]$

So, the lexicographic order of conjugates of a sequence of Lyndon factors is determined by the corresponding suffixes.

Based on the above theorem, *Bonome et al.* have proposed to do the suffix sorting using the Lyndon factorization of the text. The method proposed by them is a combination of both the incremental and the divide and conquer approach for SA construction. The problem is reduced to smaller sub-problems recursively finding their SA and then induction is used on the generated solution to find the solution of the remaining problem and then finally merging the solutions of the sub problems.

5.5.2 Incremental Suffix Array Construction

Let $T \in \sum$ be a text with *n-1* characters represented as *T[0...n-2]* and *T[n-1] = $*. There are many algorithms already proposed for the incremental construction of the Suffix Array (SA) for the text *T*. Almost all of these methods are based on dividing the text into blocks say $T_1T_2....T_k$ and then constructing the Suffix Arrays for the individual blocks say SA_1, SA_2 and merging the two, giving SA for block 1 and 2 represented as SA(12) , then finding SA_3 and merging it with SA(12) giving SA(1..3) and so on finally computing SA_k and merging it with SA(1…k-1) giving SA(1…k) which is the SA for the whole text. The complexity of these methods depends upon the complexity of the two steps mentioned above i.e block-wise SA construction and then merging of the two partial SA's. For incremental SA construction dividing the text into blocks using the Lyndon factorization of the text greatly simplifies both these steps. Also, it allows the

incremental LCP construction efficiently along with the SA. First of all, we will show the complexity reduction achieved by the Lyndon Factorization.

Each factor in the Lyndon factorization is succeeded by a factor that is smallest of all its preceding factors as well as its own cyclic shifts. So, each factor can be safely assumed as a text terminated by a smallest end of the text marker. And the problem of building SA of Lyndon Factor L_p is equivalent to the problem of building the suffix array of a text of size $m = |L_p|$. So, we can compute SA in linear time i.e. *O(m)* and *O(m log m)* bits of space using any of the existing linear time suffix array construction algorithm like Difference Cover(DC3) [77] or libdivsufsort [104]. Also, once we have sorted the suffixes of a Lyndon factor, their relative order remains same when these are merged with the sorted suffixes of adjacent Lyndon factors (as given by theorem 1). Hence the lexicographic order between the two suffixes of *T* can be established by a number of symbol comparisons that is bounded by either the size of the Lyndon factor to be merged or the LCP whichever is smaller.

To incrementally construct the suffix array, the input text *T[1....n]* is logically partitioned into *k* blocks based on the Lyndon Factorization of text. So, each block corresponds to a Lyndon Factor i.e. $T = L_1, L_2, \ldots, L_k$. The Suffix Array is computed incrementally in *k* passes, where each pass corresponds to suffix sorting of a Lyndon factor. Lyndon Factors are examined from left to right so that in pass *p*, $SA(L_1 \ldots L_p)$ is computed by first computing $SA(L_p)$ and then merging it with the $SA(L_1 \ldots L_{p-1})$ already computed. The task of computing the $SA(L_1 \ldots L_p)$ from $SA(L_1 \ldots L_{p-1})$ needs only inserting the characters from L_p in $SA(L_1 \ldots L_{p-1})$. Also, adding L_p does not modify the relative order of the suffixes already in $SA(L_{p+1} \ldots L_k)$. The algorithm can be easily adapted to external memory scenarios giving a scan based implementation due to sequential accesses to the input text.

The crucial point of the algorithm is to compute some additional information that allows merging the two SA's efficiently: One way is to simply merge the two SA's on the basis of one to one comparison of the suffixes. But this will take much time. Another method is to find the rank of each suffix of $SA(L_p)$ in $SA(L_1 \ldots L_{p-1})$ and then simply merge the two SA's based on

this rank information without any character or suffix comparison. This rank information for Lyndon factor L_p consists of an array rank[0….|L_p|] which stores in rank[j] the number of suffixes of the SA($L_1 \ldots L_{p-1}$) which are smaller than the suffix *j* of Lyndon factor L_p i.e $suf_j[L_p]$. The method given by *Ferragina et al.* [47] to find the gap array is modified to compute the values for rank[i]. Rank of each suffix is calculated using the rank of its successor suffix. Following lemma gives the computation method:

Lemma 3:

Let C[a] be the number of characters smaller than a in BWT($L_1 \ldots L_{p-1}$) and let occ(a, i) be the number of occurrences of character a in BWT($L_1 \ldots L_{p-1}$) up to position i. Assume that $suff_{j+1}[L_p]$ be a suffix in Lyndon factor L_p whose rank in BWT($L_1 \ldots L_{p-1}$) is r and let a be character at position j in T , so $suff_j[L_p] = a\, suff_{j+1}[L_p]$, then rank value of $suff_j[L_p]$ is given by

$$rank[j] = C[a] + occ[a, r]$$

Based on this, *Mantaci et al.* [93] have given an incremental approach for Suffix Array/BWT construction as:

- *Sort the suffixes of the first Lyndon factor L_1 of the text.*
- *Sort the suffixes of Lyndon Factor L_2.*
- *Merge the suffixes of the two Lyndon Factors, giving the sorted suffixes of the text T[1…F_L_3-1].*
- *Repeat the procedure for Lyndon Factors L_3 up to L_k.*

5.5.3 Limitation of the Existing Approach

As we know the performance of incremental approach depends upon the number of blocks. As the size of the text to be indexed increases, number of blocks also increases which in turn increases the runtime. So, to check for the applicability of the approach of using Lyndon factorization for dividing text into blocks for incremental suffix array construction, we have calculated the number of Lyndon Factors for files of different types of data and different sizes.

Table 5.1 Number of Lyndon Factors and size of the largest Lyndon Factor

Data Set	File Size (in MB)	No. of Lyndon Factors	Size of the Largest Factor (in MB approx)
Sources	50	27	26
	100	30	42
	200	31	108
Proteins	50	24	27
	100	25	63
	200	27	109
DNA	50	16	21
	100	16	71
	200	16	170
English	50	12	49
	100	14	54
	200	14	145

As is shown by the Table 5.1, the size of Lyndon factors as well as the number of Lyndon factors varies widely according to the type of data. Also, the memory requirement of the incremental construction depends on the size of blocks used for the construction. So, if we use Lyndon factorization in its raw form for the SA construction, as the number of Lyndon factors increases the overhead of handling different blocks increases. Also, the working memory requirements are dependent on the size of the factors. As given by table for almost all the data sets that we have considered, the size of the largest Lyndon Factor is mostly half of the original data and thus the advantage of using the incremental approach is lost, due to high working memory requirements.

5.6 Proposed Method for Extended Suffix Array Construction using Lyndon Factorization

As we know that almost all the full-text indexes use Suffix array along with their LCP array to efficiently emulate the functionality of the index. So, we are extending the approach of suffix array construction to that of the construction of extended suffix array. Constructing the suffix

array and the corresponding LCP array simultaneously will save time as well as space, as incremental construction of suffix array involves identifying common prefixes and breaking ties if any which is the procedure for constructing of LCP also. Further, constructing LCP requires some additional information like inverse suffix array that is also available during incremental suffix array construction. We move on to explain our approach by first giving an overview of the existing approach. Then, in the next subsection we explain the extensions applied to the approach to use it for the construction of extended suffix array.

To compute ESA[0…n-1] for the text *T* incrementally, using Lyndon factorization, we first partition the text into Lyndon factors using any of the existing linear methods for Lyndon factorization [40, 7, 58] of the text. Let the text be divided into *k* Lyndon factors represented as L_1, L_2….., L_k. Each Lyndon factor L_i corresponds to the text block $T[F\_L_i...F\_L_{i+1}-1]$. The algorithm constructs ESA of *T* incrementally, starting with ESA of last Lyndon factor L_k. In the next step, it constructs ESA for L_{k-1} L_k which is the ESA for $T[F\_L_{k-1}....n]$ and so on finally giving ESA for $T[F\_L_1...n]$ where F_L_1 is 0.

Extended SuffixArray(ESA) contains both the Suffix Array (SA) and Longest Common Prefix (LCP). So, for each Lyndon factor, we calculate both the SA as well as the LCP and merge it with the already calculated SA and LCP of succeeding Lyndon factors. The rank array is used to aid in the merging process. This array gives the rank of each new suffix of Lyndon factor L_p which needs to be merged with SA(L_{p+1}….L_k). The rank array gives the ranks of suffixes in their text order, but to actually use it for merging process we compute a new array called as gap[] which gives the ranks of suffixes of L_p in their SA order.

$$gap[SA_p[i]] = rank[i]$$

To calculate the rank array and finally the gap array, we need both the BWT(L_{p+1}….L_k) and first column(F) of the sorted suffixes. BWT can be stored using only $nH_k(T)$ bits, where H_k is the k^{th} order entropy. Also, column F is stored simply as the intervals of the characters instead

of being stored directly using a total space of *O(n)*. Both the values required for computation of rank[i] can be calculated in constant time [62, 90] using column F and BWT respectively.

5.6.1 Computing SA and LCP of a Lyndon Factor and Merging

As we know that for any Lyndon factor (L_p) the next Lyndon factor (L_{p+1}) is the smallest of all the suffixes of the factor L_p. So, we can compute SA(L_p) in $O(|L_p|)$ time and $O(|L_p| log\ |L_p|)$ bits of space using any of the existing linear time suffix array construction methods like Difference Cover(DC3) [77] or libdivsufsort proposed by [104] and implemented by *Yuta Mori* [96]. Once we have calculated SA(L_p) , the corresponding LCP can also be computed in time linear to the size of the Lyndon factor. Now, the next step is to merge both the SA(L_p) and LCP(L_p) with SA(L_{p+1}….L_k) and LCP(L_{p+1}….L_k) respectively.

The two SA's can simply be merged by comparing the suffixes of the SA's linearly and this merging will take time proportional to the size of the two SA's i.e $O(|L_p|\ |L_{p+1}...L_k|)$. But with the help of some additional information i.e rank we can do this merging in $O(|L_p|)$ time. To do the merging, for every suffix of L_p starting from right to left we compute its rank in the SA(L_{p+1}….L_k) . Using array rank we compute gap array which gives the rank of suffixes of SA(L_p) and after calculating this, we simply merge the two SA's. Starting from gap[0], number of suffixes equal to gap[0] are shifted from SA(L_{p+1}….L_k) to SA(L_p….L_k) then SA_p[0] is inserted at $SA_{p...k}$[gap[0]+1] and so on.

5.6.2 Updating the LCP

As we are maintaining the Extended Suffix Array, we have to calculate the LCP value side by side. As is the case with suffix array, we have both LCP(L_p) and LCP(L_{p+1}…..L_k) and we need to merge these two LCP's to get the LCP(L_p….L_k). The merging of LCP can be easily done side by side as we do the merging of corresponding SA's. The merging process we explained earlier can also be viewed as placing the suffixes of L_p in between the sorted suffixes of L_{p+1}……L_k according to their LF/rank values. So, new LCP(L_p….L_k) will be the LCP(L_{p+1}…..L_k) added

with the new LCP values at the positions where new suffixes are inserted and updating the already existing values wherever required. This can easily be done in-place in the LCP array itself. Suppose a suffix from L_p has been inserted between position *j* and *j+1* in SA(L_{p+1}……L_k). Now shift all the values till position *j* starting from position F_L_p in the LCP array. Then we need to find LCP for the suffix at position *j* and the newly inserted suffix (which has been inserted at *j+1*) and for the newly inserted suffix and suffix that was previously at position *j+1*.

i.e. LCP[F_L_p+ (j+1)] = lcp(j, j+1) where *j+1* now represents the newly inserted suffix

LCP[F_L_p+ (j+2)] = lcp(j+1, j+2) where *j+2* represents the suffix previously existing at *j+1*

SA	**Suffix**	**LCP**
1	AABABABABAA$	-1
2	ABABABAA$	1
3	ABABBBAABABABABAA$	4

Suppose suffix ABABABABAA$ is to be inserted in these suffixes.

SA	**Suffix**	**LCP**
1	AABABABABAA$	-1
2	ABABABAA$	1
3	ABABABABAA$	? need to be calculated
4	ABABBBAABABABABAA$	4 need to be updated

Now suppose a block of say *m* suffixes is inserted between a pair of suffixes at position j and j+1, even then only two LCP values need to be calculated as:

LCP[F_L_p + (j+1)] = lcp(j, j+1), where first suffix of the block is inserted at *j+1*

LCP [F_L_p + (j+m+1)] = lcp(j+m, j+m+1), where *m* is the number of suffixes in the inserted block.

SA	**Suffix**	**LCP**
1	AAABAABA$	-1
2	ABAABA$	1
3	BAABA$	0

Now, let a block of suffixes in sorted order to be inserted in the given suffixes in sorted order:

SA	Suffix	LCP
1	AABA$	-1
2	AABAABA$	4
3	AABA$	4

New block of sorted suffixes after merging:

SA	Suffix	LCP	
1	AAABAABA$	-1	
2	AABA$	?	need to be calculated
3	AABAABA$	4	
4	AABA$	4	
5	ABAABA$	1	need to be updated
6	BAABA$	0	

There is no change in the LCP value of suffixes within the block as there is no change in the sorted order of the suffixes within the block. So, we need to calculate only these two LCP value's corresponding to the block of suffixes inserted. Now to calculate these two values we use the property that lcp(i, j) of two arbitrary suffixes at position *i* and *j* in a suffix array with $i < j$ is given as:

$$lcp(i, j) = \min((LCP[i+1], LCP[i+2], \ldots\ldots, LCP[j])$$

Now, suppose we have three consecutive suffixes in the suffix array at positions *i*, *i+1* and *i+2*, and we know the LCP of suffixes at *i* and *i+2* i.e lcp(i, i+2) then lcp(i, i+1) and lcp(i+1, i+2) satisfy the condition that :

$$lcp(i, i+1) \geq lcp(i, i+2) \text{ and}$$

$$lcp(i+1, i+2) \geq lcp(i, i+2)$$

It is obvious from the above condition that the newly inserted suffix will have at least lcp(i, i+2) characters in common with its adjacent suffixes. So, to find lcp(i, i+1) we start comparing from character at position lcp(i, i+2)+1 in the suffixes *i* and *i+1*. If the character compared of two suffixes is different, then LCP of two suffixes is equal to lcp(i, i+2) otherwise LCP is computed using the LCP value of the successor suffixes. To ensure that while calculating the LCP value of two suffixes using their successor suffixes, LCP value of both the successor suffixes is available, we calculate LCP values starting from right to left in the Lyndon factor. As when we insert a suffix in between already sorted suffixes, we know the LCP value of already sorted suffixes and using the above equation, we can find LCP of the newly inserted suffix with already existing ones. Let we need to compute the LCP value of suffixes suf_i and suf_j and we already know the LCP value of suffixes suf_{i+1} and suf_{j+1} and using these values we can easily compute LCP($suf_{i+1,}$ suf_{j+1}) as:

$suf_i = c\ suf_{i+1}$

$suf_j = c\ suf_{j+1}$

so, $LCP(suf_{i,}\ suf_j) = LCP(suf_{i+1,}\ suf_{j+1}) + 1$

Also, suffix suf_{i+1} and suf_{j+1} will be close to each other in $SA(L_i \ldots L_k)$. Hence, at the maximum we need character comparisons equal to the size of the Lyndon factor being merged to compute LCP($suf_{i,}$ suf_j).

As the proposed method for updating the LCP sequentially accesses the SA as well as BWT so, it computes the LCP array sequentially. Thus, we can use this method as an external memory algorithm in which only a portion of text and SA are in the primary memory. The proposed method uses $O(n)$ bits of working space and time complexity is $O(n^2/L)$ where n is the size of the text and L be the size of the largest Lyndon factor. Also, while calculating the LCP array, partial LCP for a Lyndon factor i can be computed in $O(|L_i|)$ time and while merging the two LCP's we need to update at most $2|L_i|$ LCP entries and each LCP entry can be updated in constant time

(either a single character comparison or finding minimum LCP entry for the two successor suffixes.)

5.6.3 Experimental Work

To test how the proposed approach works in practice, we have implemented it in C language and compared it with an existing implementation of incremental construction bwt-disk by *Ferragina et al.* [47]. As to the best of our knowledge no such implementation of extended suffix array exist so, we have used bwt-disk to do the incremental construction of ESA and used it to compare our approach with.

Initially, the incremental construction of SA was done using the bwt-disk library [47] (updated to construct the SA). Afterwards, LCP array was calculated using the method of *Kasai et al.* [80]. Finally, ESA was calculated for the proposed approach. We start suffix array construction starting from the last Lyndon factor moving from right to left in the text. Also along with suffix array construction we also construct the LCP array for the current Lyndon factor. While merging the partial suffix arrays we also merge the corresponding LCP arrays. In the proposed approach while merging sorted suffixes of Lyndon factor L_{k-1} with that of L_k, first suffix of L_k is the least of all the suffixes of the text. So, it is the first suffix in the suffix array and can be placed at location 0 in the resulting SA. Similarly first suffix of L_{k-1} is smallest of all the suffixes of factors $L_1$to L_{k-1}. And all the suffixes of L_k which are smaller than this suffix have their sorted order defined in the final suffix array and hence can be placed accordingly. Thus with each merging phase some suffixes are placed in their final position in the suffix array and hence are excluded from rearrangements and comparisons in the successive phases.

Experiments were carried out with the files of different sizes (50 MB, 100MB and 200 MB) of the data sets mentioned in table 5.2 taken from Pizzza and Chilli corpus (http://pizzachili.dcc.uchile.cl/texts/). Using different data sets for the evaluation purpose helped to analyze the proposed approach for different types of data and its suitability to be used for indexing of those data.

5.6.3.1 Experimental Setup

Table 5.2 Data sets taken from Pizza & Chilli Corpus

Data Set Name	Size(in MB)	Description
Sources	50 (File 1) 100 (File 2) 200 (File 3)	Concatenation of all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions.
Proteins	50 (File 1) 100 (File 2) 200 (File 3)	Sequence of newline-separated protein sequences obtained from the Swissprot database.
DNA	50 (File 1) 100 (File 2) 200 (File 3)	Sequence of newline-separated gene DNA sequences obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project.
English	50 (File 1) 100 (File 2) 200 (File 3)	Concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project.

5.6.4 Results and Discussion

The main point of analysis is the construction time of the method. We have run the implementation for the files given in table 5.1 and have calculated the corresponding construction times. We have plotted the construction times for all the files of same size for both the approaches i.e bwt-disk and our proposed approach of Lyndon factorization. Figure 5.1 plots the construction time of all files of size 200MB for both the approaches. Similarly Figure 5.2 and Figure 5.3 plot the construction times of files of sizes 100MB and 50MB respectively. Figure 5.1 and 5.2 show that, our method shows improvement over the existing approach in terms of construction time for all data sets of size 200MB and 100MB. This performance gain is attributed to the lesser merging time of our approach.

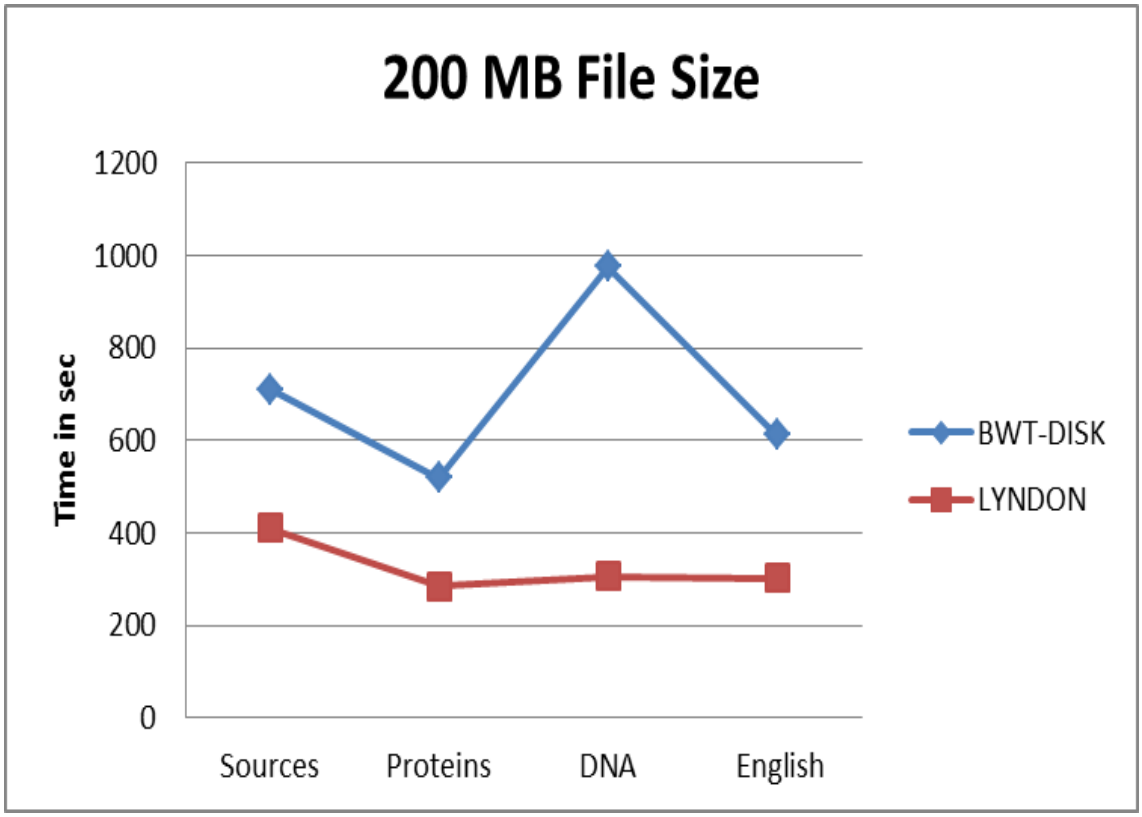

Figure 5.2 Construction Time of both approaches for files of size 200 MB

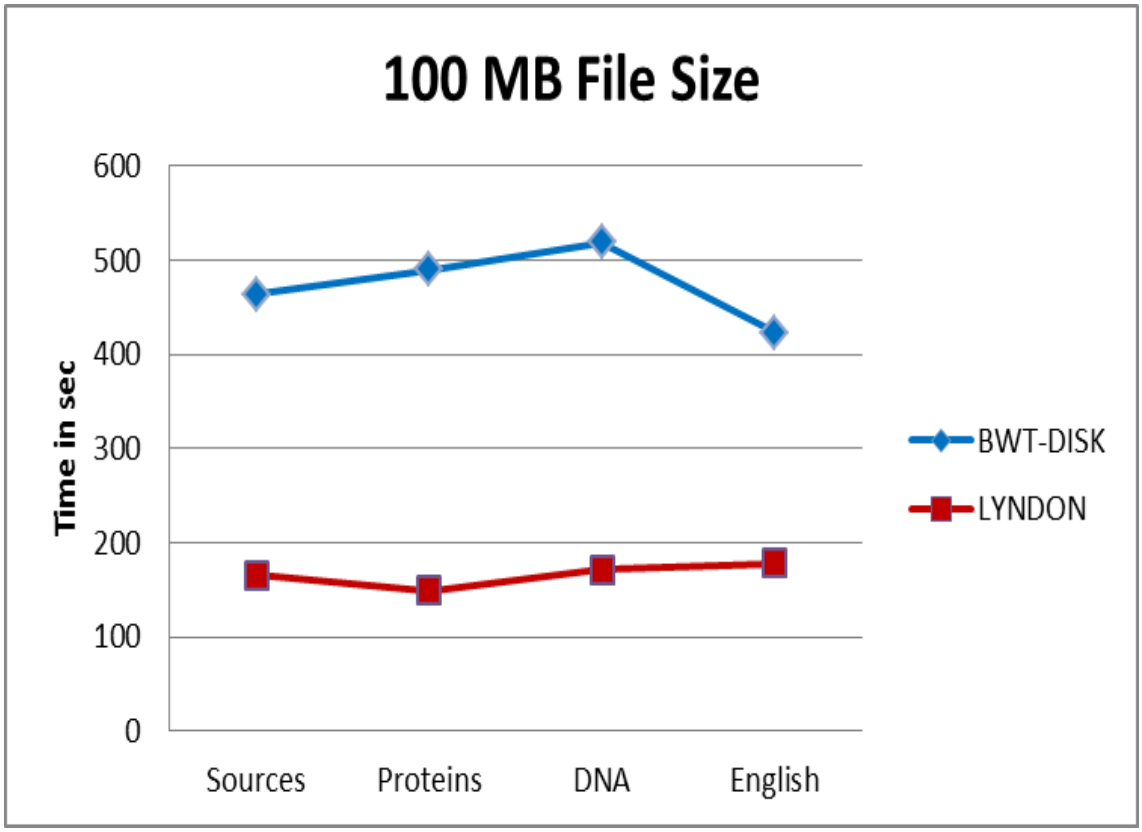

Figure 5.3 Construction Time of both approaches for files of size 100 MB

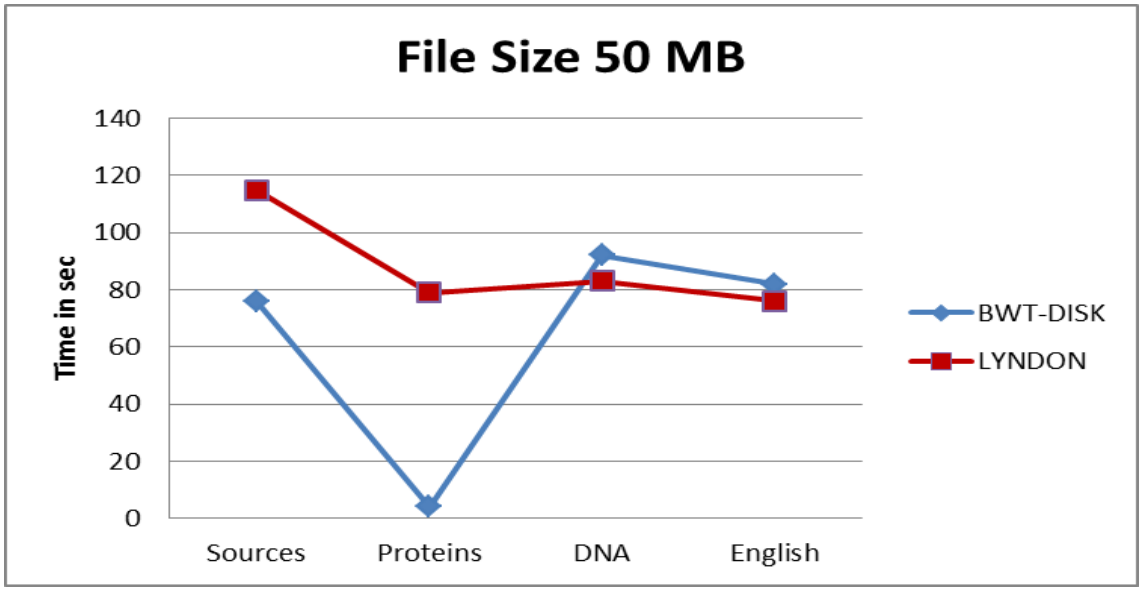

Figure 5.4 Construction Time of both approaches for files of size 50 MB

Figure 5.3 shows that the existing approach has better performance in case of files of size 50 MB. The main reason is that for small file sizes the direct construction time is so less that doing it the incremental way, simply adds the overhead. We have also plotted the construction times for all the files of the three sizes for the two approaches in figure 5.5 and 5.6. Figure 5.5 shows that Also in our approach the no of factors is more for all the data sets and hence the overhead. Additional experiments to get the effect of number of factors on the incremental construction were conducted and the results show that as the number of factors increase the merging time of bwt-disk increases significantly thus making it prohibitive to be used for increasingly large data sets.

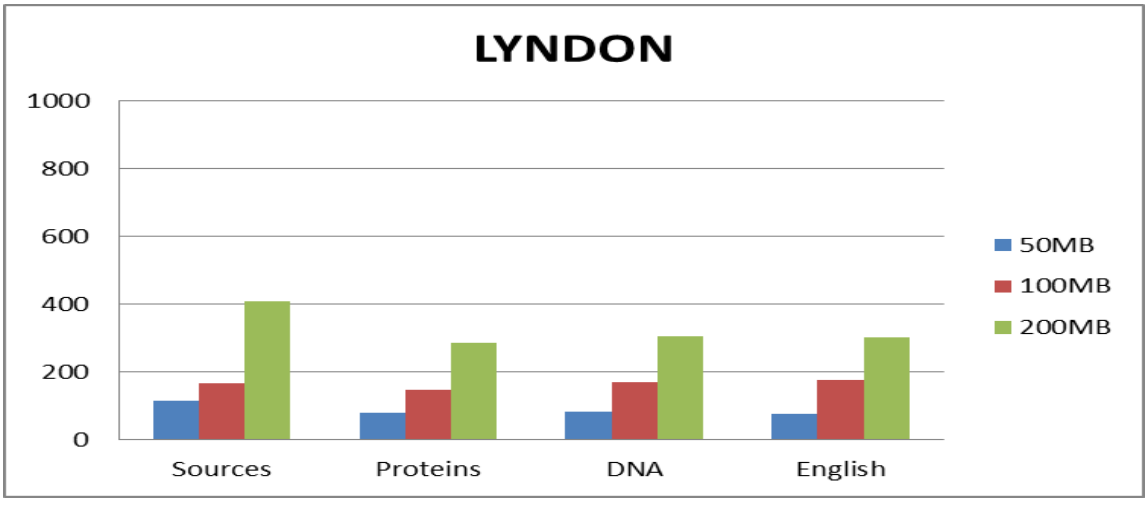

Figure 5.5 Construction Time of All files for Lyndon approach

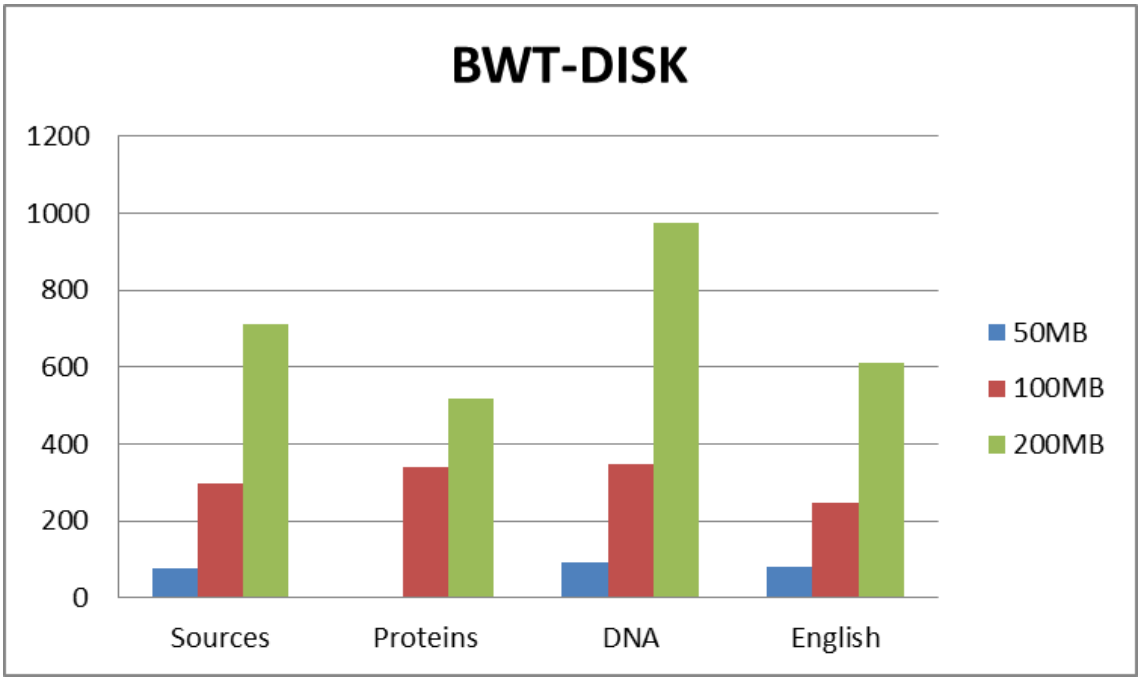

Figure 5.6 Construction Time of All files for BWT-DISK approach

But the slow increase in merging time of our approach with the increase in number of factors makes it suitable to be used for large data sets also.

5.7 Conclusion

In this chapter, we have given a method to construct the extended suffix array of a text *T* using the Lyndon factorization of text. This is an incremental method, which constructs the extended suffix array block by block with small memory requirement, which includes space for BWT, SA and rank array of a Lyndon Factor along with the space for whole SA and LCP. Experimental analysis of the approach shows the performance gain achieved by using the Lyndon factorization for incremental construction. This method can be easily adapted for the space efficient construction of other variants of suffix array. Also, the method is easily adaptable to external memory or parallel scenarios. Space/time tradeoff can be adjusted according to the requirements by using the dynamic data structures for rank and select operations, which provide the additional information required during the merging step.

As future perspectives, we will try to take advantage of the sampling of SA to further reduce the space requirements of the method. Similarly to what we developed for the *ESA*, we will study

how we can adapt this method for incremental construction of compressed Suffix array. Finally, a promising perspective will be to reduce the space occupancy of the index by considering the compressed counterparts of various additional structures used during the construction process.

CHAPTER-6

Conclusion and Future Work

This thesis has presented improved data structures and Algorithms for storing and processing dynamic and massive data. In this chapter, we conclude our work and outline some directions for future work.

6.1 Conclusions

Nowadays, huge amount of data is produced from varying sources like biological experiments, Internet routing information, sensor data, search engines etc. some of which is also dynamic in nature. Managing such a massive and dynamic data needs to consider all the issues like storage, searching, retrieval along with maintaining the dynamism of the data. So, data efficient data handling in today's scenario leads to design of new computational methods to represent, store, and process this data efficiently both in terms of time and space. Due to the development of new data structuring paradigm called as retroactive data structures, it has become quite easy to develop dynamic algorithms from the existing static ones.

In our work, we have considered the dynamization of a static algorithm using the innovative concept of retroactive data structures. As graphs/ networks, are the most basic and powerful objects to model relations and processes to solve the real world problems in varying domains, so, we have taken one of the most widely used graph problem i.e. the shortest path problem which we aimed to dynamize. We have proposed the implementation of retroactive priority queue for the solution of DSSSP (Dynamic Single Source Shortest Path) problem. Here, we have given an approach to implement the retroactive priority queue data structure with the help of priority search tree. A retroactive priority queue maintains all its update as well as query operations with respect to time parameter, so we use priority search tree as it can store values along two different dimensions. Then, we have used this to solve the dynamic single source shortest path problem in which dynamic changes can be in the form of edge weight changes.

This approach of using retroactive data structures for adding dynamism to existing solution of single source shortest path problem is better than the previous approaches that depend upon identifying the affected set of vertices and then updating their shortest paths according to the given changes. The retroactive data structures automatically identify the vertices that are affected by a given change which can then be corrected in constant time. As the changes are propagated level by level i.e. the vertex affected by a given change is identified and corrected which in turn gives the next vertex that has been affected by this new change and so on. In this way, new shortest path tree is generated from the existing one using minimum amount of topological changes. One more advantage of the approach is that it can be easily adapted for other types of dynamic changes that can occur like a number of update operations appear simultaneously or edge insertions or deletions.

Further, we have changed the underlying data structure for implementing the retroactive priority queue and have used balanced search trees (i.e. Red-black tree). This method provided an efficient solution to the dynamic shortest path problem. We have implemented the retroactive priority queue using the height balanced search trees, so the cost of all the operations of priority queue is dependent on the height of the underlying tree data structure. Also, the number of computations is reduced in case of dynamic changes in the graph as we are moving back in time at a point immediately before the time when the edge under updation is going to be used in any of the shortest paths. We are selectively choosing the point which has been actually affected by the change and applying the updation to the affected portion only.

Moreover, to have an efficient external memory algorithm for the construction of index structure for massive data, we have designed an algorithm for the construction of extended suffix array using the Lyndon factorization of text. We have used Lyndon factorization of text to overcome the disadvantage of the previous algorithms. We have proposed a method to construct the extended suffix array of a text T using the Lyndon factorization of text. This is an incremental method, which constructs the Extended SA block by block with memory requirements which include space for BWT, SA and rank array of a Lyndon factor along with the

space for whole SA and LCP. Experimental analysis of the approach indicated the performance gain achieved by using the Lyndon factorization for incremental construction. The proposed method can be easily adapted for the space efficient construction of other variants of suffix array. Also, the method is easily adaptable to parallel scenarios and the space/time trade-off can be adjusted according to the requirements by using the dynamic data structures for rank and select operations, which provide the additional information required during the merging step.

6.2 Future Work

The advantage of dynamizing the static algorithm of single source shortest path problem using retroactive data structures is that the resultant dynamic algorithm for the solution of the problem using retroactive data structures can be easily adapted to relax the assumption we have made in our work. Transformations (Reweighing) can be applied to adapt our solution in case of general graphs (graphs with positive or negative edge weights). Also, our dynamic dijkstra algorithm can also be extended to solve the dynamic all-pairs shortest path problem.

As future perspectives for efficient extended suffix array construction, we will try to take advantage of the sampling of suffix array to further reduce the space requirements of the method. Similar to what we have developed for the extended suffix array, we will study how this method can be adapted for incremental construction of compressed suffix array. Finally, a promising perspective will be to reduce the space occupancy of the index by considering the compressed counterparts of various additional structures like rank and select array used during the construction process.

Reference

[1] Abouelhoda M. I., Kurtz S. and Ohlebusch, E., "The Enhanced Suffix Array and Its Applications to Genome Analysis," *Algorithms in Bioinformatics*, Lecture Notes in Computer Science, vol. 2452**,** pp. 449, 2002. doi:10.1007/3-540-45784-4_35.

[2]Abouelhoda M. I., Kurtz S. and Ohlebusch, E., "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[3] Acar U. A., Blelloch G. E. and Tangwongsan K., "Non-oblivious retroactive data structures," Carnegie Mellon University, Pittsburgh PA, Technical Report, CMU-CS-07-169, 2007.

[4] Adelson-Velskii G. and Landis E. M., "An algorithm for the organization of information," *Soviet Mathametics Doklady*, vol. 3, pp. 1259–1263, 1962.

[5] Aggarwal A. and Vitter J. S., "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116-1127, 1988.

[6] Ahmed D. M., Sundram, D. and Piramuthu, S., "Knowledge based Scenario Management-Process and Support," *Decision Support Systems*, vol. 49, no. 4, pp. 507-520, 2010.

[7] Apostolico A. and Crochemore M., "Fast parallel Lyndon factorization with applications," *Mathematical Systems Theory*, vol. 28, no. 2, pp. 89-108, 1995.

[8] Arge L., Ferragina P., Grossi R. and Vitter J. S., "On sorting strings in external memory," in *29th ACM Symposium on Theory of Computing*, Eds. ACM Press, El Paso, pp. 540–548, 1997.

[9] Basch J., Guibas L. J. and Hershberger J., "Data structures for mobile data," *Journal of Algorithms*, vol. 31, no. 1, pp. 1–28, 1999.

[10] Bender M. A. and Farach-Colton M., "The LCA problem revisited," in *Proceedings of Theoretical Informatics, 4th Latin American Symposium (LATIN 2000)*, Gonnet G. H., Panario D. and Viola A., Eds. Springer Verlag, Lecture Notes in Computer Science, vol. 1776, pp. 88–94, April 2000.

[11] Bentley J. L. and Saxe J. B., "Decomposable searching problems I: Static-to-dynamic transformation," *Journal of Algorithms*, vol. 1, no. 4, pp. 301–358, 1980.

[12] Berstel J., Lauve A., Reutenauer C. and Saliola F., "Combinatorics on words: Chritoffel words and repetition in words," *CRM Monograph Series*, American Mathematical Society, vol. 27, 2008.

[13] Bingmann T., Fischer J. and Osipov V., "Inducing suffix and LCP arrays in external memory," in *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pp. 88-102, 2013.

[14] Bonomo S, Mantaci S, Restivo A, Rosone G and Sciortino M, "Suffixes, Conjugates and Lyndon words," *Lecture Notes of Computer Science*, vol. 7907, pp. 131-142, 2013.

[15] Brlek S., Lachau J. O., Provençal X. and Reutenauer C., "Lyndon + Christoffel=Digitally Convex," *Pattern Recognition*, vol. 42, no. 10, pp. 2239-2246, 2009.

[16] Buriol L. S., Resende M. G. and Thorup M., "Speeding up dynamic shortest path algorithms," AT&T Labs Research, Technical Reprot TR TD-5RJ8B, 2003.

[17] Buriol L. S., Resende M. G. and Thorup M., "Speeding Up Dynamic Shortest-Path Algorithms," *INFORMS Journal on Computing*, vol. 20, pp. 191-204, 2008.

[18] Burkhardt S. and K¨arkk¨ainen J., "Fast lightweight suffix array construction and checking," in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, Eds. Springer Verlag, June 2003, Lecture Notes in Computer Science, vol. 2676, pp. 55–69.

[19] Burrows M. and Wheeler D. J., "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Technical Report 124, 1994.

[20] Canovas R. and Navarro G., "Practical compressed suffix trees," in *Proceedings of the International Conference on Experimental Algorithms*, Lecture Notes in Computer Science, vol. 6049**,** pp. 94-105, 2010.

[21] Cecilia R. A. and Raimund G. S., "Randomized Search Trees," in *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, Ed. IEEE, New York, pp. 540 –545, 1989.

[22] Chan T. M., "Dynamic planar convex hull operations in near logarithmic amortized time," *Journal of ACM*, vol. 48, pp. 1-12, 2001.

[23] CHAZELLE B, "How to search in history," *Information and Control*, vol. 77, pp. 77-99, 1985.

[24] Chen K. T., Fox R. H. and Lyndon R. C., "Free differential calculus IV, the quotient groups of the lower central series," *Annals of Mathematics*, vol. 68, no. 1, pp. 81-95, 1958.

[25] Chen G., Puglisi S. J. and Smyth W. F., "Lempel-Ziv factorization using less time and space," *Mathematics in Computer Science*, vol. 1, no. 4, pp. 605–623, 2008.

[26] COLE R, "Searching and storing similar lists," *Journal of Algorithms*, vol. 7, pp. 202-220, 1986.

[27] Crauser A. and Ferragina P., “Theoretical and experimental study on the construction of suffix arrays in external memory,” *Algorithmica*, vol. 32, no. 1, pp. 1–35, 2002.

[28] Culpepper J. S., Navarro G., Puglisi S. J. and Turpin A., “Top-k ranked document search in general text databases,” in *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*, Meyer U. and Berg M. D., Eds. Springer, September 2010. Lecture Notes in Computer Science, vol. 6347, pp. 194–205.

[29] Debbarma A., Annappa B. and Mude R. G., “Performance analysis of graph based iterative algorithms on MapReduce framework,” in *International Conference for Convergence of Technology (I2CT)*, Ed. IEEE, pp. 1-6, 2014

[30] Demaine E. D., Iacono J. and Langerman S., “Retroactive data structures,” in *The Proceedings of the 15th Annual ACM-SIAM. Symposium on Discrete Algorithms*, pp. 274–283, 2004.

[31] Demaine E. D., Iacono J. and Langerman S., “Retroactive data structures: Extended Abstract,” *ACM Transactions on Algorithms*, vol. 3, no. 2, 2007.

[32] Demaine E. D., Kaler T., Liu Q., Sidford A. and Yedidia A., “Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing,” in *Algorithms and Data Structures-WADS 2015*, Dehne F., Sack JR. and Stege U., Lecture Notes in Computer Science, vol. 9214, 2015.

[33] Dementiev R., Karkkainen J., Mehnert J. and Sanders P., “Better external memory suffix array construction,” *ACM Journal of Experimental Algorithmics*, vol. 12, article 3.4, 2008.

[34] Demers A., Reps T. and Teitelbaum T., “Incremental evaluation of attribute grammars with application to syntax directed editors,” in *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp. 105–116, 1981.

[35] Demetrescu C., Frigioni D., Marchetti-Spaccamela A. and Nanni U., “Maintaining shortest paths in digraphs with arbitrary arc weights: an experimental study,” *Algorithm Engineering*, pp. 218-229, 2000.

[36] Demetrescu C. and Italiano G. F., “Experimental analysis of dynamic all pairs shortest path algorithms,” *ACM Transactions on Algorithms*, vol. 2, no. 4, pp. 578-601, 2006.

[37] Dickerson M.T., Eppstein D., Goodrich M.T., “Cloning Voronoi Diagrams via Retroactive Data Structures,” in *Algorithms – ESA 2010*, de Berg M., Meyer U., Eds Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, vol. 6346.

[38] Dijkstra E. W., "A note on two problems in connexion with graphs," *Journal Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[39] Dobkin D P and Munro J I, "Efficient uses of the past," in *Proceedings of 21st Annual IEEE Symposium on Foundations of Computer Science*, pp. 200-206, 1980.

[40] Duval J. P., "Factorizing words over an ordered alphabet," *Journal of Algorithms*, vol. 4, no. 4, pp. 363-381, 1983.

[41] Even S. and Shiloach Y., "An on-line edge-deletion problem," *Journal of the ACM*, vol. 28, no. 1, pp. 1–4, 1981.

[42] Farach M., "Optimal suffix tree construction with large alphabets," in *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 1997)*, pp. 137–143, October 1997.

[43] Ferragina P. and Grossi R., "The string B-Tree: a new data structure for string search in external memory and its applications," *Journal of the ACM*, vol. 46, pp. 236-280, 1999.

[44] Ferragina P and Manzini G, "Opportunistic data structures with applications," in *Proceedings of Annual Symposium on Foundations of Computer Science*, pp. 390-398, 2000.

[45] Ferragina P. and Manzini G., "Indexing compressed text," *Journal of the ACM*, vol. 52, pp. 552-581, 2005.

[46] Ferragina P. and Manzini G., "On compressing the textual web," in *Proceedings of the 3rd ACM International Conference on Web Search and Web Data Mining (WSDM)*, Davison B. D., Suel T., Craswell N. and Liu B., Eds ACM, February 2010. pp. 391–400.

[47] Ferragina P., Gagie T. and Manzini G., "Lightweight data indexing and compression in external memory," *Algorithmica*, vol. 63, no. 3, pp. 707-730, 2012.

[48] Fiat A. and Kaplan H., "Making data structures confluently persistent," in *Proceedings of 12th Annual Symposium on Discrete Algorithms*, Washington, DC, pp. 537–546, January 2001.

[49] Fischer J., Makinen V. and Navarro G., "An(other) entropy-bounded compressed suffix tree," in *Proceedings of 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Lecture Notes in Computer Science, vol. 5029, pp. 152-165, 2008.

[50] Flicek P. and Birney E., "Sense from sequence reads: methods for alignment and assembly," *Nature Methods (Suppl)*, vol. 6, no. 11, pp. S6–S12, 2009.

[51] Fortz B. and Thorup M., "Internet traffic engineering by optimizing OSPF weights," in *the proceedings of the IEEE Conference on Computer Communications*, pp. 519-528, 2000.

[52] Frigioni D., Ioffreda M., Nanni U. and Pasqualone G., "Analysis of dynamic algorithms for the single source shortest path problem," *ACM Journal on Experimental Algorithmics*, vol. 3, article 5, 1998.

[53] Frigioni D., Marchetti-Spaccamela A. and Nanni U., "Fully Dynamic Algorithms for Maintaining Shortest Paths Trees," *Journal of Algorithms*, vol. 34, no. 2, pp. 251-281, 2000.

[54] Fujishige S., "A note on the problem of updating shortest paths," *Networks*, vol. 11, pp. 317-319, 1981.

[55] Gallagher S. and Savage T., "Cross-cultural analysis in online community research: A literature review," *Computers in Human Behaviour*, vol. 29, pp. 1028–1038, 2013.

[56] Gallo G., "Reoptimization procedures in shortest path problems," *Rivista di Matematica per le Scienze Economiche e Sociali*, vol. 3, pp. 3-13, 1981.

[57] Galperin and Rivest R. L., "Scapegoat trees," in *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete algorithms*, pp. 165–174, 1993.

[58] Ghuman S. S., Giaquinta E. and Tarhio J., "Alternative algorithms for Lyndon Factorization," *Stringology*, pp.169-178, 2014.

[59] Gonnet G. H., "Pat 3.1: An efficient text searching system," user's manual, University of Waterloo, Canada, 1987.

[60] Gonnet G. H., Baeza-Yates R. and Snider T., "New indices for text: PAT trees and PAT arrays" in *Information Retrieval: Data Structures & Algorithms*, Frakes W. B. and Baeza-Yates R., Ed., Prentice-Hall, 1992.

[61] Gonzalez C. and Dutt V., "Exploration and exploitation during information search and experimential choice, Journal of Dynamic Decision Making, vol. 2, no. 1, 2016.

[62] González R. and Navarro G., "Improved dynamic rank-select entropy-bound structures," in *Proceedings of the Latin American Theoretical Informatics (LATIN)*, Lecture Notes in Computer Science. vol. 4957, 2008.

[63] Grossi R. and Vitter J., "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, pp. 378-407, 2005.

[64] Grossi R, "A quick tour on suffix arrays and compressed suffix arrays," Theoretical Computer Science, vol. 412, no. 27, pp. 2964–2973, 2011.

[65] Guibas L. J. and Sedgewick R., "A dichtomatic framework for balanced trees," in *Proceedings of IEEE Symposium on Foundations of Computer Science*, pp. 8-21, 1978.

[66] Gupta, A., Hon, W. K., Shah, R. and Vitter, J. S., "Compressed Dictionaries: Space Measures, Data Sets, and Experiments," Springer Verlag, Lecture Notes in Computer Science, vol. 4007, pp. 158–169, 2006.

[67] Gupta, S. K., Bhatnagar, Vasudha and Wasan S. K., "On Mining of Data," *IETE Journal of Research, Special issue on Data and Knowledge Engineering*, vol. 47, no. 1, pp. 5-18, 2001.

[68] Gupta V. and Chhabra J. K., "Measurement of dynamic metrics using dynamic analysis of programs," in *Proceedings of the WSEAS International Conference on Applied Computing Conference*, pp. 81- 86, 2008.

[69] Gusfield D., "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," Cambridge University Press, Cambridge, UK, 1997.

[70] Hohlweg C. and Reutenauer C., "Lyndon words, permutations and trees," *Theoretical Computer Science*, vol. 307, no. 1, pp. 173-178, 2003.

[71] Hon W., Sadakane K. and Sung W., "Breaking a time-and-space barrier in constructing full-text indices," in *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science (FOCS'03)*, IEEE Computer Society Press, pp. 251–260, 2003.

[72] Hon W., Lam T., Sadakane K., Sung W. and Yiu S., "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, vol. 48, pp. 23-36, 2007.

[73] Hood R. and Melville R., "Real-time queue operations in pure LISP," *Information Processing Letters*, vol. 13, pp. 50-54, 1981.

[74] Itoh H. and Tanaka H., "An efficient method for in memory construction of suffix arrays," in *Proceedings of the 6th Symposium on String Processing and Information Retrieval*, IEEE Computer Society, Cancun, Mexico, pp. 81–88, 1999.

[75] Kamburugamuve S., Christiansen L. and Fox G., "A Framework for Real Time Processing of Sensor Data in the Cloud", *Journal of Sensors, vol.* 2015, article ID 468047, pp. 1-11, 2015

[76] Karkkainen J. and Sanders P., "Simple linear work suffix array construction," in *Proceedings of the 30th International Colloquium Automata, Languages and Programming*. Eds. Springer-Verlag Berlin, Lecture Notes in Computer Science, vol. 2971, pp. 943–955, 2003.

[77] Karkkainen J., Sanders P. and Burkhardt S., "Linear work suffix array construction," *Journal of the ACM*, vol. 53, no. 6, pp. 1–19, 2006.

[78] Karkkainen J., "Fast BWT in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 249–257, 2007.

[79] Karp R. M., Miller R. E. and Rosenberg A. L., “Rapid identification of repeated patterns in strings, trees and arrays,” in *Proceedings of the 4th annual ACM symposium on Theory of computing*, Denver, Colorado, United States, pp. 125–136, 1972.

[80] Kasai T., Lee G., Arimura H., Arikawa S. and Park K., “Linear-time longest-common-prefix computation in suffix arrays and its applications,” in *Proceedings of Annual Symposium on Combinatorial Pattern Matching*, vol. 2089, pp. 181-192, 2001.

[81] Kim D. K., Sim J. S., Park H. and Park K., “Linear-time construction of suffix arrays,” in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, Eds. Springer Verlag, Lecture Notes in Computer Science, vol. 2676, pp. 186–199., June 2003.

[82] Kim D. K., Jo J. and Park H., “A fast algorithm for constructing suffix arrays for fixed size alphabets,” in *Proceedings of the 3rd Workshop on Experimental and Efficient Algorithms (WEA 2004)*, Ribeiro C. C. and Martins S. L., Eds. Springer-Verlag, Berlin, pp. 301–314, 2004.

[83] Knuth D., “Sorting and Searching,” in *The art of computer programming*, 2nd ed., vol. 3, Ed. Massachusetts: Addison-Wesley, 1998, pp. 458-481.

[84] Ko P. and Aluru S., “Space efficient linear time construction of suffix arrays,” in *Proceedings of the 14th Annual Symposium (CPM 2003)*, Baeza-Yates R., Chavez E. and Crochemore M., Eds. Springer-Verlag, Berlin, Lecture Notes in Computer Science, vol. 2676, pp. 200–210, 2003.

[85] Kurtz S., “Reducing the space requirement of suffix trees,” *Software- Practice and Experience*, vol. 29, no. 13, pp. 1149–1171, 1999.

[86] Larsson N. J. and Sadakane K., “Faster suffix sorting,” Department of Computer Science, Lund University, Technical Report LU-CS-TR:99-214, FUNDFD6/(NFCS-3140)/1–20/(1999), May 1999.

[87] Lilien L., Kamal Z., Bhuse V. and Gupta A., “Opportunistic networks: the concept and research challenges in privacy and security,” in *International Workshop on Research Challenges in Security and Privacy for Mobile and Wireless Networks (WSPWN)*, Miami, FL, pp. 131-147, 2006.

[88] Lilien L., Gupta A., Kamal Z. and Yang Z., “Opportunistic Resource Utilization Networks ”,skrowteN coH dA dezilaicepS rof mgidaraP weN A–*Computers and Electrical Engineering*, vol. 36, no. 2, pp. 328-340, 2010.

[89] Makinen V. and Navarro G., “Succinct suffix arrays based on run-length encoding,” *Nordic Journal of Computing*, vol. 12, pp. 40-66, 2005.

[90] Makinen V. and Navarro G., "Dynamic entropy-compressed sequences and full-text indexes," *ACM Transactions on Algorithms*, vol. 4, no. 3, article 32, 2008.

[91] Manber U. and Myers G., "Suffix arrays: A new method for on-line string searches," in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 1990, pp. 319–327.

[92] Manber U. and Myers G., "Suffix arrays: a new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935-948, 1993.

[93] Mantaci S., Restivo A., Rosone G. and Sciortino M., "Sorting suffixes of a text via its Lyndon Factorization," *Stringology*, pp. 119-127, 2013.

[94] Manzini G. and Ferragina P., "Engineering a lightweight suffix array construction algorithm," *Algorithmica*, vol. 40, no. 1, pp. 33–50, 2004.

[95] McCreight E. M., "Priority search trees," *SIAM Journal of Computing*, vol. 14, no. 2, pp. 257–276, 1985.

[96] Mori Y., "sais: An implementation of the induced sorting algorithm," 2008. URL http: //sites.google.com/site/yuta256/sais.

[97] Mulmuley K., "Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract)," in *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 180–196, 1991.

[98] Myers E. W., "AVL Dags," Dept. of Computer Science, The University of Arizona, Tucson, AZ, Technical Reprot TR 82-9, 1982.

[99] Myers E. W., "An applicative random-access stack," *Information Processing Letters*, vol. 17, pp. 241-248, 1983.

[100] Myers E. W., "Efficient applicative data types," in *Conf. Record 11th Annual ACM Symposium on Principles of Programming Languages*, pp. 66-75, 1984.

[101] Narviaez P., Siu K. and Tzeng H., "New Dynamic SPT Algorithm Based on a Ball-and-String Model," *ACM Transactions on Networking*, vol. 9, no. 6, pp. 706-718, 2001.

[102] Navarro G. and Makinen V., "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, 2007.

[103] Nievergelt J. and Reingold E. M., "Binary search trees of bounded balance," *SIAM Journal of Computing*, vol. 2, pp. 33-43, 1973.

[104] Nong G., Zhang S. and Chan W. H., "Linear suffix array construction by almost pure induced-sorting," i*n Proceedings of the 19th IEEE Data Compression Conference (DCC'09) Washington D. C.,* Storer J. A. and Marcellin M. W., Eds *IEEE Computer Society*, pp. 193-202, 2009.

[105] Nong G., Zhang S., and Chan W. H., "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, 2011.

[106] Overmars M. H., "Searching in the past I," Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, Technical Report RUU-CS-81-7, 1981.

[107] Overmars M. H., "Searching in the Past II: General Transforms," Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, Technical Report RUU-CS-81-9, 1981A.

[108] Overmars M. H., "Dynamization of order decomposable set problems," *Journal of Algorithms*, vol. 2, pp. 245–260, 1981P.

[109] M. H. Overmars. The Design of Dynamic Data Structures. Springer, 1983.

[110] Patil M., Thankachan S. V., Shah R., Hon W.K., Vitter J.S. and Chandrasekaran S., "Inverted indexes for phrases and strings," in *Proceedings of the 34th International ACM SIGIR conference on Research and Development in Information Retrieval,* Ma W. Y., Nie J. Y., Baeza-Yates R.A., Chua T. S. and Croft W. B., Eds, ACM, July 2011, pp. 555–564.

[111] Pugh W. and Teitelbaum T., "Incremental computation via function caching," in *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pp. 315–328, 1989.

[112] Puglisi S. J., Smyth W. F. and Turpin A., "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, 2007.

[113] Ramalingam G. and Reps T. W., "An Incremental Algorithm for a Generalization of the Shortest-Path Problem," *Journal of Algorithms*, vol. 21, no. 2, pp. 267-305, 1996.

[114] Ramalingam G. and Reps T. W., "On the computational complexity of dynamic graph problems," *Theoretical Computer Science*, vol. 158, pp. 233–277, 1996A.

[115] Reps T., Teitelbaum T. and Demers A., "Incremental context-dependent analysis for language-based editors," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 449- 477, 1983.

[116] Ruskey F. and Williams A., "The coolest way to generate combinations," *Discrete Mathematics*, vol. 17, no. 309, pp. 5305–5320, 2009.

[117] Sadakane K., "A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation," in *DCC: Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 129–138, 1998.

[118] Sadakane K., "Compressed suffix trees with full functionality," *Theory of Computing Systems*, vol. 41, pp. 589-607, 2007.

[119] Sahni S., "*Analysis of algorithms, Data Structures and Applications*," Chapman-Hall/CRC Press, 2005.

[120] Salson M., Lecroq T., Leonard M., and Mouchard L., "Dynamic extended suffix arrays," *Journal of Discrete Algorithms*, vol. 8, pp. 241-257, 2010.

[121] Sarnak N. and Tarjan R. E., "Planar point location using persistent search trees," *Communications of the ACM*, vol. 29, pp. 669- 679, 1986.

[122] Sawada J. and Williams A., "A surprisingly simple de Bruijn sequence construction," *Discrete Mathematics*, vol. 339, no. 1, pp. 127-131, 2016.

[123] Schurmann K. and Stoye J., "An incomplex algorithm for fast suffix array construction," in *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX05)*, SIAM, pp. 77–85, 2005.

[124] Schwarzkopf O., "Dynamic maintenance of geometric structures made easy," in *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 197–206, 1991.

[125] Seward J., "On the performance of BWT sorting algorithms," in *DCC: Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 173–182, 2000.

[126] Singh A. and Sharma T. P., "A survey on area coverage in wireless sensor networks," in *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pp. 900-907, 2014.

[127] Singh M.P., Kumar S., Kumar A. and Kumar A., "Comprehensive Study of Search Engine," *Recent Advances in Mathematics, Statistics and Computer Science*, pp. 621-633, 2016.

[128] Sinha R., Puglisi S. J., Moffat A. and Turpin A., "Improving suffix array locality for fast pattern matching on disk," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 661-672, 2008.

[129] Sniedovich M., "Dijkstra's algorithm revisited: the dynamic programming connexion," *Journal of Control and Cybernetics*, vol. 35, pp. 599-620, 2006.

[130] Swart G. F., “Efficient Algorithms for Computing Geometric Intersections,” Department of Computer Science, University of Washington, Seattle, WA, Technical Report 85-01-02, 1985.

[131] Tangwongsan K., “Active data structures and applications to dynamic and kinetic algorithms,” Dissertation, Carnegie Mellon University, 2006.

[132] Vakili P., “Massively parallel and distributed simulation of a class of discrete event systems: a different perspective,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 2, no. 3, pp. 214-238, 1992.

[133] Valimaki N., Makinen V., Gerlach W. and Dixit K., “Engineering a compressed suffix tree implementation,” *ACM Journal of Experimental Algorithmics*, vol.14, article 2, 2009.

[134] Vitter J., “External memory algorithms,” invited Tutorial in *ACM Symposium on Principles of Database Systems (PODS’98)*, Seattle, WA, 1998.

[135] Weiner P., “Linear pattern matching algorithms,” in *Proceedings of the Annual Symposium on Foundations of Computer Science*, pp. 1-11, 1973.

[136] Xiao B., Zhuge Q. and Sha EH-M., “Efficient algorithms for dynamic update of shortest path tree in networking,” *Journal of Computer Application.*, vol. 11, pp. 60-75, 2004.