PERANCANGAN MODUL PENGINDEKS PADA *SEARCH ENGINE* BERUPA *INDUCED GENERALIZED SUFFIX TREE* UNTUK KEPERLUAN PERANGKINGAN DOKUMEN

Proposal Skripsi

Disusun untuk memenuhi salah satu syarat
memperoleh gelar Sarjana Komputer

Oleh:
Zaidan Pratama
1313618013

PROGRAM STUDI ILMU KOMPUTER
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI JAKARTA

2022

LEMBAR PENGESAHAN

Dengan ini saya mahasiswa Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Negeri Jakarta

Nama	:	Zaidan Pratama
No. Registrasi	:	1313618013
Program Studi	:	Ilmu Komputer
Judul	:	Perancangan Modul Pengindeks Pada *Search Engine* Berupa *Induced Generalized Suffix Tree* Untuk Keperluan Perangkingan Dokumen

Menyatakan bahwa proposal skripsi ini telah siap diajukan untuk seminar pra skripsi.

Menyetujui,

Dosen Pembimbing I	Dosen Pembimbing II
Muhammad Eka Suryana, M.Kom.	**Med Irzal, M.Kom.**
NIP. 19851223 201212 1 002	NIP. 19770615 200312 1 001

Mengetahui,
Koordinator Program Studi Ilmu Komputer

Ir. Fariani Hermin M.T
NIP. 19600211 198703 2 001

KATA PENGANTAR

Puji syukur kepada Allah Subhanahu wa Ta'alla atas berkah rahmat, hidayah, dan karunia-Nya sehingga penulis dapat menyelesaikan proposal skripsi yang berjudul "Perancangan Modul Pengindeks Pada *Search Engine* Berupa *Induced Generalized Suffix Tree* Untuk Keperluan Perangkingan Dokumen". Proposal skripsi ini disusun untuk memenuhi syarat kelulusan studi pada Program Studi Ilmu Komputer Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Negeri Jakarta.

Dalam proses penyusunan proposal skripsi ini banyak pihak yang telah membantu dan memberikan dukungan, doa, serta bimbingannya. Oleh karena itu pada kesempatan kali ini penulis ingin mengucapkan terima kasih kepada:

1. Allah SWT atas segala rahmat dan lindungan-Nya yang telah memberikan kemudahan selama proses penyusunan proposal skripsi.
2. Orang tua dan keluarga saya yang selalu memberikan semangat serta doa yang tidak henti-hentinya untuk kelancaran saya menempuh pendidikan dan menyelesaikan proposal skripsi ini.
3. Bapak Muhammad Eka Suryana M.Kom selaku Dosen Pembimbing I yang telah memberikan arahan dan bimbingan dalam penulisan proposal skripsi ini.
4. Bapak Med Irzal M.Kom selaku Dosen Pembimbing II yang telah memberikan arahan dan bimbingan dalam penulisan proposal skripsi ini.
5. Teman-teman dan sahabat yang telah banyak membantu dan memberi dukungan moral sehingga proposal skripsi ini dapat selesai dengan sebaik-baiknya.

6. Semua pihak yang telah memberikan bantuan dan dukungan dalam penyusunan proposal skripsi ini yang tidak dapat penulis sebutkan satu per satu.

Proposal skripsi ini masih jauh dari kata sempurna dari segi isi maupun sistematika. Untuk itu kritik dan saran yang bersifat membangun sangat dibutuhkan guna menyempurnakan proposal skripsi ini menjadi lebih baik. Semoga proposal skripsi ini bisa bermanfaat bagi pembaca pada umumnya.

DAFTAR ISI

DAFTAR ISI **vi**

DAFTAR GAMBAR **vii**

DAFTAR TABEL **viii**

I PENDAHULUAN **1**

A. Latar Belakang Masalah 1

B. Rumusan Masalah 6

C. Pembatasan Masalah 6

D. Tujuan Penelitian 6

E. Manfaat Penelitian 7

II KAJIAN PUSTAKA **8**

A. *Search Engine* 8

B. *Document Retrieval* 11

C. Masalah *Color Set Size* 12

D. *Generalized Suffix Tree* 14

1. Konstruksi *Generalized Suffix Tree* 15

2. *Suffix Range* 16

3. *Optimal Index for Colored Range Query* 17

4. *Y-fast Trie for Efficient Successor Query* 18

E. *Induced Generalized Suffix Tree* 18

1. *IGST-f: IGST* Untuk Frekuensi f 18

2. Representasi *Array* dari *IGST-f* 20

F. Indeks Efisien Untuk *Top-k Document Retrieval Problem* 23

G. Konstruksi Algoritma Pengindeksan 26

III METODOLOGI PENELITIAN **30**

A. *Flowchart* Algoritma 30

1. *Pseudocode* Algoritma 32

B. Contoh Konstruksi *Generalized Suffix Tree* 34

C. Alat dan Bahan Penelitian 34

D. Dataset 35

E. Tahapan Pengembangan 38

DAFTAR PUSTAKA **42**

DAFTAR GAMBAR

Gambar 1.1 *High Level Google Architecture* (Brin dan Page, 1998) 3

Gambar 2.1 *High Level Google Architecture* (Brin dan Page, 1998) 9

Gambar 2.2 *CSS Example* (Chi dan Hui, 1992) 12

Gambar 2.3 *Generalized Suffix Tree* (Hon dkk., 2010) 14

Gambar 2.4 *Suffix Tree* (McCreight, 1976) 15

Gambar 2.5 Pembentukan *Suffix Tree ababc* (McCreight, 1976) 15

Gambar 2.6 *Suffix Range* (Hon dkk., 2010) 17

Gambar 2.7 *Pre-IGST-2* (Hon dkk., 2010) 19

Gambar 2.8 *IGST-2* (Hon dkk., 2010) 20

Gambar 2.9 Contoh dari penggabungan *list L* (Hon dkk., 2010) 24

Gambar 3.1 *Flowchart* Algoritma Indeks Efisien Untuk *Top-K Document Retrieval Problem* 30

Gambar 3.2 *Flowchart* Pencarian Nilai *Count* 31

Gambar 3.3 Contoh Konstruksi *Generalized Suffix Tree* 34

Gambar 3.4 5 Data Pertama Dataset pada Tabel *Page Information* 35

DAFTAR TABEL

Tabel 1.1 Persentase *global market share search engine* Februari 2022 (StatCounter, 2022) 2

Tabel 2.1 *Array I* dari IGST-2 pada gambar 2.8 (Hon dkk., 2010) 22

Tabel 3.1 Deskripsi Tabel *Page Information* 36

Tabel 3.2 Deskripsi Tabel *Linking* 36

Tabel 3.3 Deskripsi Tabel *Style Resource* 37

Tabel 3.4 Deskripsi Tabel *Script Resource* 37

Tabel 3.5 Deskripsi Tabel *Forms* 37

Tabel 3.6 Deskripsi Tabel *Images* 37

Tabel 3.7 Deskripsi Tabel *List* 38

Tabel 3.8 Deskripsi Tabel *Tables* 38

BAB I

PENDAHULUAN

A. Latar Belakang Masalah

Web Search Engine adalah program perangkat lunak yang melakukan pencarian di internet yang memiliki banyak *website* dan bekerja berdasarkan kata kunci atau *query words* yang kita tentukan. *Search engine* mencari kata kunci yang kita tentukan di dalam *database* atau basis data informasi yang dimiliki. Alan Emtage beserta dua orang temannya, yaitu Bill Heelan dan J. Peter Deutsch memperkenalkan *search engine* pertama kali pada tahun 1990 dengan nama *Archie*. *Archie* mengunduh daftar direktori semua *file* yang terletak di situs anonim publik FTP *(File Transfer Protocol)* lalu membuat *database* nama *file* yang dapat dicari. Namun, *Archie* tidak mengindeks isi situs-situs tersebut karena jumlah datanya sangat terbatas sehingga dapat dengan mudah dicari secara manual. (Seymour dkk., 2011).

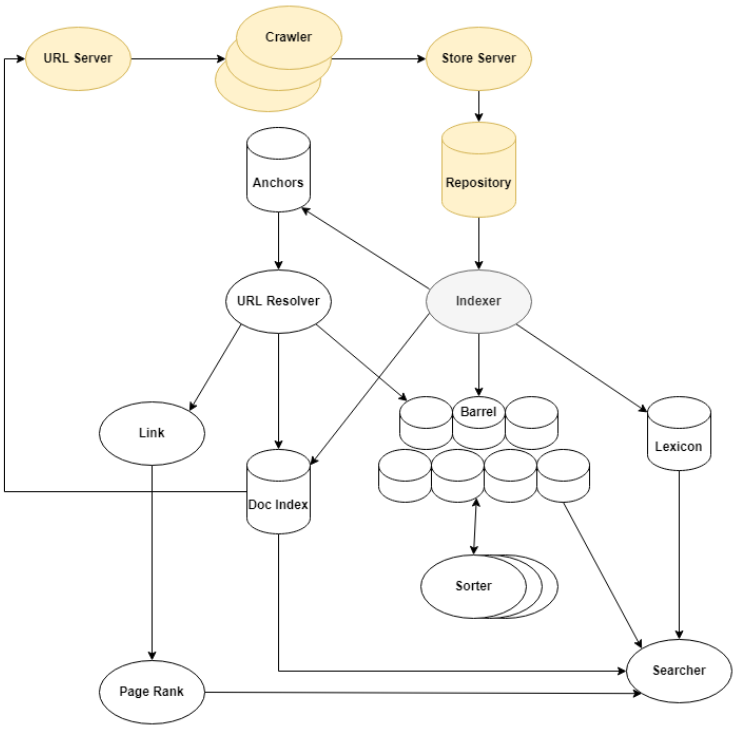
Di era modern ini, kegiatan kita sehari-hari tak lepas dari internet. *Search Engine* sendiri erat kaitannya dengan pengguna internet. Secara tidak langsung kita pasti menggunakan *search engine* ketika berselancar di internet. Meskipun aplikasi ponsel dengan berbagai tujuan sudah menjamur, peran peramban atau *browser* dan *search engine* masih cukup penting dalam melakukan pencarian informasi. Sistem pencarian, pemeringkatan, dan pengindeksan modern yang didukung oleh daya komputasi yang ditingkatkan, kecepatan jaringan yang cepat, dan kapasitas penyimpanan data yang hampir tak terbatas menjadikan kita memiliki akses mudah ke semua informasi yang kita butuhkan. Prinsip-prinsip yang menjadi dasar teknologi modern ini sudah ada dari sebelum tahun 1960-an. (Harman dkk., 2019).

Berikut persentase *global market share* atau pangsa pasar dari beberapa *search engine* per Februari 2022.

Tabel 1.1: Persentase *global market share search engine* Februari 2022 (StatCounter, 2022)

Search engine	**Market share (%)**
Google	92.01
Bing	2.96
Yahoo!	1.51
Baidu	1.17
Yandex	1.06
DuckDuckGo	0.68

Dapat kita lihat dari data di atas, *Google* masih menjadi *search engine* favorit bagi para pengguna internet untuk saat ini. *Google* sendiri ditemukan dan diperkenalkan pada tahun 1998 oleh Larry Page dan Sergey Brin. Salah satu keunggulan *Google* adalah pengaplikasian *PageRank*. Manfaat terbesar dari *PageRank* adalah kehebatannya dalam mengatasi *underspecified queries* atau kueri yang kurang spesifik. Sebagai contoh, pada mesin pencari konvensional jika kita mencari "Universitas Stanford" maka hasil pencariannya dapat mengembalikan sejumlah halaman web yang mencantumkan Stanford di dalamnya. Jika menggunakan *PageRank*, maka halaman web Universitas Stanford akan menjadi halaman pertama yang dikembalikan. *Google* sebagai *search engine* terdiri dari beberapa komponen utama. Gambar 1.1 menunjukkan komponen-komponen tersebut yang termuat dalam *High Level Google Architecture*. (Brin dkk., 1998).



Gambar 1.1: *High Level Google Architecture* (Brin dan Page, 1998)

Pada penelitian sebelumnya yang dilakukan oleh Muhammad Fathan Qoriiba yang berjudul "Perancangan *Crawler* Sebagai Pendukung Pada *Search Engine*", rangkaian komponen *crawler* berhasil dibuat dan ditandai dengan warna krem pada gambar 1.1. Untuk membuat *search engine*, mulanya dibutuhkan sebuah *crawler*. *Crawler* merupakan sebuah program untuk mengambil informasi yang ada di halaman web. Dibutuhkan perancangan *crawler* yang baik untuk mendapatkan hasil yang baik. Algoritma yang digunakan mengikuti algoritma awal perkembangan *Google*, yaitu *breadth first search crawling* dan *modified similarity based crawling*. *Crawler* yang dibuat dapat dijalankan untuk situs web statis yang dibuat menggunakan *html* versi 5 atau 4. Hasil *crawling* disimpan di dalam *database MySQL*. (Qoriiba, 2021).

Rangkaian komponen *crawler* yang dibuat menghasilkan dataset berisi 8 tabel yang berukuran 176 MiB atau sekitar 185 MB. Dataset ini disusun menggunakan

tools crawling yang dikembangkan. Situs yang digunakan untuk proses *crawling* adalah situs *indosport.com* dan *curiouscuisiniere.com*. Dataset ini akan digunakan untuk kelanjutan penelitian *search engine*. (Qoriiba, 2021).

Salah satu komponen lain yang merupakan kelengkapan dari arsitektur *search engine* adalah pengindeks atau *indexer*. *Indexer* melakukan beberapa fungsi penting sebagai salah satu komponen arsitektur *search engine*. *Indexer* membaca repositori, membuka kompresi dokumen, dan menguraikan dokumen tersebut. *Indexer* lalu mendistribusikan kumpulan kata yang disebut *hits* ke dalam satu set *barrels* yang akan membuat sebagian *forward index* terurut. *Indexer* juga menguraikan semua tautan pada halaman web dan akan menyimpan semua informasi penting ke dalam *anchor file*. (Brin dan Page, 1998).

Algoritma yang terkait dengan *indexing* dan *document retrieval* sendiri telah diteliti oleh cukup banyak orang sebelumnya. Salah satu pelopornya yaitu S. Muthukrishnan pada tahun 2002 dengan memanfaatkan penggunaan *Generalized Suffix Tree*. Algoritma Muthukrishnan menekankan pada pengambilan semua dokumen yang memiliki jumlah kemunculan pola input *query* jika melebihi ambang batas tertentu. Ambang batas tersebut ditentukan oleh pencari informasi atau dengan kata lain oleh *user*. (Hon dkk., 2010).

Pada penelitian yang berjudul "*Privacy-preserving string search on encrypted genomic data using a generalized suffix tree*", penggunaan *Generalized Suffix Tree* untuk pengindeksan memberikan peningkatan yang signifikan terhadap waktu eksekusi *query*. Struktur indeks dari *generalized suffix tree* ini digunakan untuk membuat sebuah protokol yang aman dan efisien dalam melakukan pencarian *substring* dan set-maksimal pada data genom. Hasil eksperimen menunjukkan bahwa metode yang diajukan pada penelitian ini dapat mengkomputasi pencarian *substring* dan set-maksimal yang aman pada sebuah dataset *single-nucleotide*

polymorphism (SNPs) dengan 2184 *records* (setiap *record* berisi 10000 SNPs) dalam 2.3 dan 2 detik. (Mahdi dkk., 2021).

Pada penelitian yang berjudul "*Parallel and private generalized suffix tree construction and query on genomic data*", konstruksi *Generalized Suffix Tree* secara paralel untuk pengindeksan dapat digunakan untuk data genom yang besar. GST menyediakan indeks yang efisien untuk data genomik yang dapat digunakan dalam banyak *query* penelusuran berbasis string, yang fundamental bagi kegunaan data genom. Konstruksi GST secara paralel dikerjakan secara terdistribusi dan terbagi. (Al Aziz dkk., 2022).

Penggunaan GST atau *Generalized Suffix Tree* sebagai struktur data pengindeksan juga dilakukan dalam penelitian yang berjudul "*Full-Text Search on Data with Access Control using Generalized Suffix Tree*". Dalam penelitian ini, alih-alih menggunakan *inverted index* yang mana lazim digunakan untuk pencarian *full-text*, *Generalized Suffix Tree* digunakan karena kemampuannya dalam melakukan pencarian *substring* dalam dokumen. Hasil penelitian ini menunjukkan bahwa penggunaan GST memerlukan memori yang besar bahkan melebihi jumlah dokumen tersebut. Namun, pencarian menggunakan GST 3 kali lebih cepat dibandingkan *inverted index*. (Zaky dan Munir, 2016).

Wing-Kai Hon dan beberapa temannya berhasil mengembangkan algoritma baru dengan pemanfaatan dari *Generalized Suffix Tree* yang lebih efisien dari segi ruang dan waktu. Algoritma ini digunakan untuk mendapatkan indeks yang efisien dalam masalah pencarian dokumen. Pendekatan ini didasarkan pada penggunaan baru dari pohon sufiks atau *suffix tree* yang dinamakan *induced generalized suffix tree (IGST)*. (Hon dkk., 2010)

Di dalam penelitian ini, penulis akan membuat modul pengindeks atau *indexer* sebagai salah satu komponen arsitektur *search engine* yang ditandai dengan warna

biru muda pada gambar 1.1. Penulis akan membuat *indexer* menggunakan algoritma yang sudah dikembangkan oleh Wing-Kai Hon dan teman-temannya. Dataset yang akan digunakan sebagai kumpulan dokumen adalah dataset hasil *crawling* yang sudah dibuat oleh Muhammad Fathan Qoriiba sebelumnya. Penelitian ini sekaligus akan melanjutkan rangkaian penelitian sebelumnya terkait *search engine*.

B. Rumusan Masalah

Berdasarkan uraian pada latar belakang yang ada di atas, maka rumusan masalah pada penelitian ini adalah "Bagaimana cara mengindeks dan melakukan pencarian dokumen menggunakan algoritma *Efficient Index For Retrieving Top-k Most Frequent Document*?"

C. Pembatasan Masalah

Pembatasan masalah pada penelitian ini adalah pembuatan sebagian arsitektur *search engine* yaitu modul pengindeks atau *indexer*. Kumpulan dokumen yang akan digunakan sebagai dataset adalah dataset yang telah disusun oleh Muhammad Fathan Qoriiba dalam penelitiannya yang berjudul PERANCANGAN *CRAWLER* SEBAGAI PENDUKUNG PADA *SEARCH ENGINE*.

D. Tujuan Penelitian

Adapun tujuan dari penelitian ini adalah membuat modul pengindeks atau *indexer* yang akan digunakan untuk kebutuhan efisiensi pencarian yang termasuk dalam perangkingan dokumen pada *search engine*.

E. Manfaat Penelitian

Penelitian ini diharapkan dapat mengoptimisasi pencarian pada *search engine* dari segi waktu dan ruang serta relevansi pencarian. Penelitian ini juga diharapkan memberikan manfaat lainnya, antara lain:

1. Bagi penulis

   Menambah ilmu dan pengetahuan mengenai *document retrieval* khususnya pada bagian *search engine* dan *indexing*, mempraktikkan ilmu-ilmu yang sudah didapat semasa kuliah serta mendapatkan gelar Sarjana Komputer.

2. Bagi Program Studi Ilmu Komputer

   Penelitian ini merupakan salah satu rangkaian dan lanjutan dari penelitian sebelumnya mengenai *search engine*. Penelitian ini dapat memberikan gambaran dan informasi mengenai *search engine* khususnya bagian *indexer* dan dapat dijadikan acuan untuk penelitian selanjutnya terkait *search engine*.

3. Bagi Universitas Negeri Jakarta

   Menjadi acuan untuk evaluasi dan pertimbangan serta pengembangan kualitas akademik di Universitas Negeri Jakarta khususnya Program Studi Ilmu Komputer.

BAB II

KAJIAN PUSTAKA

A. *Search Engine*

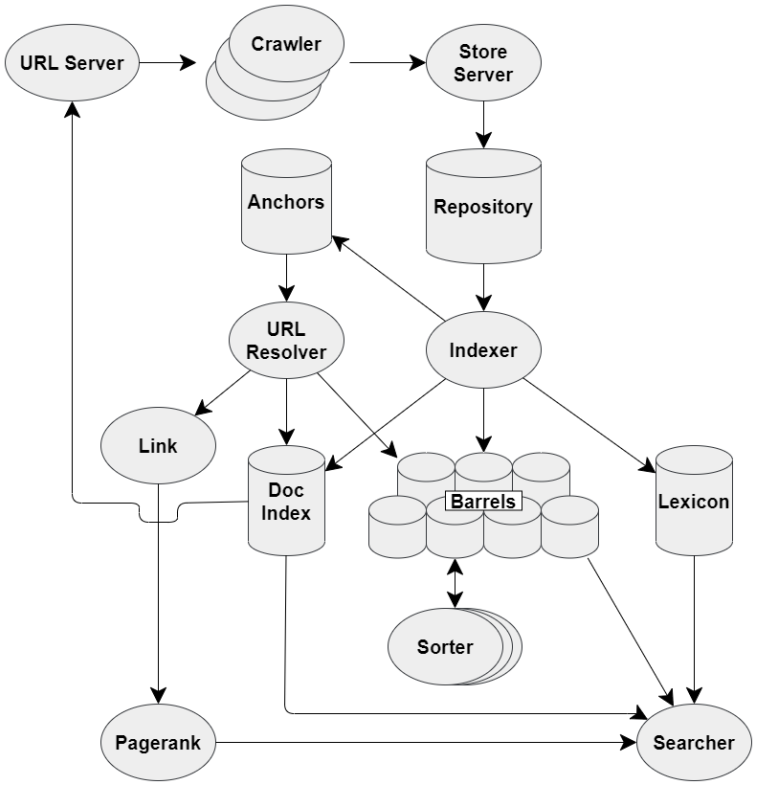
Search engine merupakan contoh dari *Information Retrieval System* berskala besar. Sejarah *search engine* bisa dibilang dimulai dari tahun 1990 ketika Alan Emtage dan kawan-kawannya membuat *Archie*. Setelah *Archie*, banyak *search engine* lain yang mulai diperkenalkan. *Gopher* dan *Veronica and Jughead* diperkenalkan pada tahun 1991. Pada tahun 1993, *W3Catalog and Wanderer*, *Aliweb*, dan *Jump Station* diperkenalkan ke publik. Menyusul di tahun 1994 ada *WebCrawler* dan pada tahun 1995 *MetaCrawler*, *AltaVista*, serta *Excite* diperkenalkan. (Seymour dkk., 2011).

AltaVista menoreh popularitas yang sangat tinggi pada waktu itu sebelum akhirnya digeser oleh kenaikan popularitas *Google*. Salah satu fitur unggulan yang ditawarkan oleh *AltaVista* adalah adanya *natural languange search*. Hal ini memungkinkan pengguna untuk mencari informasi dengan mencari frase atau dengan kalimat tanya. Sebagai contoh, kita bisa mencari informasi letak kota London dengan menanyakan *"Where is London?"* dan mendapatkan jawaban yang relevan dengan tidak menyertakan kata *"where"* dan *"is"* dalam hasil pencarian. (Seymour dkk., 2011).

Google sendiri resmi diperkenalkan pada tahun 1998 oleh Sergey Brin dan Larry Page. Kesuksesan *Google* pada waktu itu sebagian besar adalah andil dari algoritma *PageRank* yang telah dipatenkan. *PageRank* sendiri bertugas untuk melakukan pemeringkatan atau me-*ranking* halaman web agar hasil informasi yang dicari oleh pengguna lebih relevan. Setelah *Google*, ada cukup banyak *search engine*

yang diperkenalkan ke publik. Beberapa yang mungkin kita tahu adalah *Yahoo! Search* dan *Bing!*. *Yahoo! Search* diperkenalkan oleh *Yahoo! Inc.* pada tahun 2004 sedangkan *Bing!* diperkenalkan oleh *Microsoft* pada tahun 2009. (Seymour dkk., 2011). Sampai saat ini, *Google* masih memimpin dalam hal jumlah pengguna dan popularitas.

Sergey Brin dan Larry Page memperkenalkan arsitektur *Google* sebagai sebuah *search engine* yang dinamakan dengan *High Level Google Architecture*. Gambar 2.1 menunjukkan komponen-komponen penyusun *search engine*. (Brin dan Page, 1998).



Gambar 2.1: *High Level Google Architecture* (Brin dan Page, 1998)

Web crawling dilakukan oleh beberapa *crawler*. *URLServer* mengirim daftar URL untuk diambil datanya oleh *crawler*. Halaman web yang sudah diambil datanya dikirim ke dalam *Store Server* lalu dikompresi dan dikirim ke dalam *Repository*.

Setiap halaman web memiliki nomor identifikasi yang disebut dengan *docID*. Proses penomoran ini dilakukan ketika URL diambil dari halaman web. Selanjutnya proses pengindeksan dilakukan oleh *Indexer* dan *Sorter*. *Indexer* membaca repositori, membuka kompresi dokumen, dan menguraikan dokumen tersebut. *Indexer* lalu mendistribusikan kumpulan kata yang disebut *hits* ke dalam satu set *barrels* yang akan membuat sebagian *forward index* terurut. *Indexer* juga menguraikan semua tautan pada halaman web dan akan menyimpan semua informasi penting ke dalam *anchor file*. *Anchor file* berisi informasi yang berguna untuk menentukan tautan tersebut merujuk ke mana dan juga berisi teks dari tautan tersebut.

Anchor file kemudian dibaca oleh *URLResolver* dan diubah URL nya dari URL relatif ke URL absolut serta pengubahan menjadi *docID*. *URLResolver* kemudian mengirim teks yang ada di dalam *anchor file* ke *forward index*, yang mana berhubungan dengan *docID* yang sudah dirujuk oleh *anchor file* sebelumnya. *URLResolver* juga membuat basis data yang berisi tautan yang merupakan pasangan dari *docID*-nya masing-masing. Basis data ini digunakan untuk menghitung *PageRank* dari seluruh dokumen.

Sorter lalu mengambil *barrels* yang sudah diurutkan berdasarkan *docID* dan mengurutkannya kembali berdasarkan *wordID* untuk membuat *inverted index*. *Sorter* juga membuat daftar yang berisi *wordID* dan *offset* ke dalam *inverted index*. *DumpLexicon* mengambil daftar tadi beserta *lexicon* yang dihasilkan oleh *Indexer* lalu membuat *lexicon* baru untuk digunakan oleh *Searcher*. *Searcher* dijalankan oleh web server dan menggunakan *lexicon* serta *inverted index* dan juga *PageRank* untuk menjawab *query* atau pertanyaan. (Brin dan Page, 1998).

B. *Document Retrieval*

Dalam basis data yang berisi teks atau *string*, kita memiliki kumpulan beberapa dokumen yang berisi teks atau *string* alih-alih hanya satu dokumen *string* saja. Dalam kasus ini, masalah dasarnya adalah mengambil semua dokumen di mana pola *query P* berada. Masalah ini disebut juga masalah pengambilan dokumen atau *document retrieval problem*. (Hon dkk., 2010). Masalah ini sudah pernah diteliti sebelumnya oleh S. Muthukrishnan.

Muthukrishnan memperkenalkan algoritma pertama yang dikenal untuk menyelesaikan *document listing problem*. Dalam *document listing problem*, diberikan sebuah *query online* yang terdiri dari pola string *p* atau *query pattern p* dengan panjang *m*. Hasil akhirnya adalah pengembalian kumpulan-kumpulan dokumen yang mengandung satu atau lebih pola *query p*. *Document listing problem* merupakan bagian dari *document retrieval problem*. (Muthukrishnan, 2002). Diberikan sebuah kumpulan *string*, *document listing* mengacu pada masalah dari menemukan semua *string* atau dokumen di mana *query string* muncul. (Puglisi dan Zhukova, 2021).

Permasalahan utama dari algoritma Muthukrishnan ini adalah bisa terjadi banyak kemunculan pola *query* lain pada seluruh koleksi dokumen. Di sisi lain, jumlah keseluruhan dokumen yang mengandung pola *query* yang dicari bisa saja jauh lebih kecil. Maka dari itu, metode yang mencari seluruh kemunculan pola *query* lalu melaporkan dokumen mana saja yang mengandung pola tersebut bisa dibilang jauh dari kata efisien. (Hon dkk., 2010).

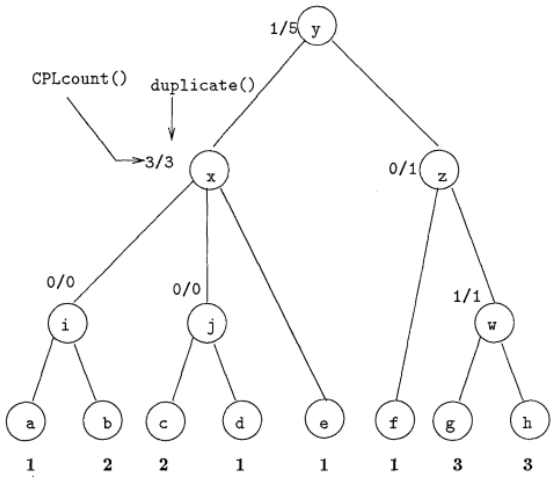
Muthukrishnan juga memperkenalkan varian lain pada penelitiannya. Dalam varian ini, dokumen yang dikembalikan hanya dokumen yang memiliki lebih dari *f* kemunculan pola. Varian ini disebut *f-mine problem*. Dari segi pencarian informasi, varian ini lebih menarik karena hanya dokumen dengan relevansi yang paling tinggi

yang dikembalikan. (Hon dkk., 2010).

Pada penelitian Wing-Kai Hon dan teman-temannya, masalah yang diteliti erat kaitannya dengan *f-mine problem*. Mereka langsung mencari *top-k* dokumen yang memiliki jumlah kemunculan maksimum untuk pola yang diberikan. Mereka menambahkan sesuatu yang bekerja seperti *inverse document mine query*, yang dengan cepat memungkinkan mereka menemukan ambang f_k (atau cukup f ketika konteksnya jelas) yang merupakan frekuensi pola dalam frekuensi dokumen ke-k. Berdasarkan ini kita bisa melihat hubungan yang kuat dengan *f-mine problem*. (Hon dkk., 2010).

Komponen utama dari solusi Wing-Kai Hon dan kawan-kawannya adalah *induced generalized suffix tree (IGST)*. IGST sendiri kurang lebih memiliki struktur yang sama dengan indeks yang diperkenalkan oleh Muthukrishnan. Namun, terdapat pengaplikasian baru pada IGST di mana IGST itu sendiri dilinearisasi dan dikombinasikan dengan fungsionalitas dari pencari selanjutnya untuk mendapatkan hasil yang diinginkan. (Hon dkk., 2010).

C. Masalah *Color Set Size*



Gambar 2.2: *CSS Example* (Chi dan Hui, 1992)

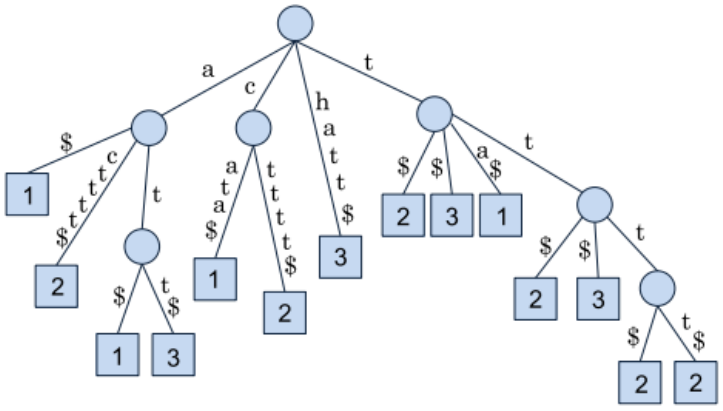
Diketahui *C* = $\{1, ..., m\}$ adalah himpunan warna dan *T* adalah pohon berakar yang memiliki *n* daun di mana setiap daun memiliki warna *c*, $c \in C$. *Color Set Size* untuk *vertex v*, yang dinotasikan sebagai *css(v)*, adalah jumlah warna daun berbeda dalam subpohon yang berakar pada *v*. Masalah *Color Set Size* adalah untuk mencari *color set size* pada setiap *internal vertex v* di dalam *T*. Sebagai contoh pada gambar 2.2, *css(x)* = 2 dan *css(y)* = 3. (Chi dan Hui, 1992).

LeafList adalah daftar daun yang sudah terurut berdasarkan *post-order traversal* dari pohon *T*. Untuk *v* yang merupakan daun dari *T* yang memiliki warna *c*, *lastleaf(x)* adalah daun terakhir yang memiliki warna sama yaitu *c* yang mendahului *x* pada *LeafList*. Jika tidak ada daun yang mendahului *x* yang memiliki warna sama dengan *x* maka *lastleaf(x)* = *Nil*. Pada gambar 2.2, *lastleaf(a)* dan *lastleaf(b)* adalah *Nil* karena tidak ada daun sebelumnya yang memiliki warna sama dengan daun a atau b. Untuk *lastleaf(c)* adalah b karena b memiliki warna yang sama dengan c dan mendahului c. (Chi dan Hui, 1992).

Untuk *vertex* atau *node x* dan *y* dari *T*, *lca(x,y)* adalah *least common ancestor* dari *x,y* atau *parent* yang paling tidak sama dari *x,y*. Subpohon dari *T* yang berakar pada *node u* disebut dengan *subtree(u)* dan *leafcount(u)* adalah jumlah seluruh daun dari *subtree(u)*. Sebuah *internal node u* dikatakan *color-pair-lca* atau CPL jika ada daun *x* yang memiliki warna *c*, di mana *u* = *lca(lastleaf(x),x)*. Sebagai contoh pada gambar 2.2, *vertex w* adalah CPL dari warna 3. Sebuah *node* bisa menjadi CPL dari beberapa warna. *CPLCount(u)* = *k* jika dari semua warna, terdapat *k* kemunculan dari pasangan daun di mana *u* merupakan CPL dari pasangan daun tersebut. $duplicate(u) = \sum_{v \in subtree(u)} CPLCount(v) = CPLCount(u) + \sum_{w \in W} duplicate(w)$ di mana *W* adalah himpunan anak dari *u*. (Chi dan Hui, 1992).

D. *Generalized Suffix Tree*

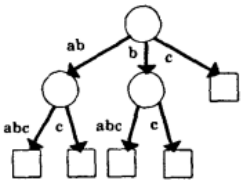
Diketahui $\Delta = \{T_1, T_2, ..., T_m\}$ menunjukkan satu set dokumen. Setiap dokumen adalah *string* karakter dengan karakter yang diambil dari $\sum$ alfabet umum yang ukurannya $|\sum|$ bisa tak terbatas. Kita asumsikan untuk setiap *i*, karakter terakhir dari dokumen T_i ditandai dengan $\$_i$, yang mana bersifat *unique* untuk semua karakter pada semua dokumen. *Generalized Suffix Tree (GST)* untuk Δ adalah sebuah *compact trie* yang menyimpan semua sufiks dari dokumen T_i. Lebih tepatnya, setiap sufiks dari setiap dokumen sesuai dengan daun yang berbeda di GST. Setiap tepi diberi label oleh urutan karakter, sehingga untuk setiap daun yang mewakili beberapa sufiks *s*, rangkaian label tepi di sepanjang jalur akar ke daun adalah *s*. Selain itu, untuk setiap *internal node u*, tepi yang mengarah ke anak-anaknya semuanya berbeda dengan karakter pertama pada label tepi yang sesuai, sehingga anak-anak *u* diurutkan menurut urutan abjad dari karakter pertama tersebut. Pada gambar 2.3, terdapat 3 set dokumen yaitu T_1 = cata, T_2 = actttt, dan T_3 = hatt. (Hon dkk., 2010).



Gambar 2.3: *Generalized Suffix Tree* (Hon dkk., 2010)

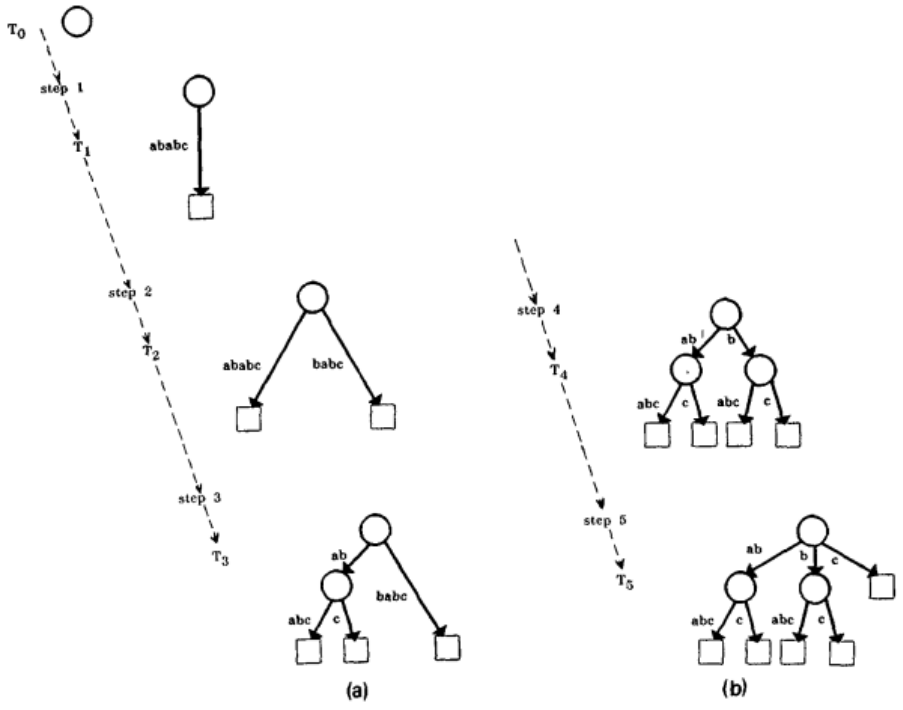
1. Konstruksi *Generalized Suffix Tree*

Sebuah pohon sufiks dari *string* $W = w_1, ..., w_n$ adalah sebuah pohon dengan tepi $O(n)$ dan *n* daun. Setiap daun di dalam pohon sufiks terhubung dengan indeks *i* $1 \leq i \leq n$. Tepi-tepinya dilabeli dengan karakter sehingga rangkaian tepi berlabel dari akar ke daun dengan indeks *i* adalah sufiks ke-i dari *W*. Sebuah pohon sufiks untuk *string* dengan panjang *n* dapat dibuat dalam $O(n)$ ruang dan waktu. Sebagai contoh *string ababc* akan disusun seperti gambar 2.4. (McCreight, 1976).



Gambar 2.4: *Suffix Tree* (McCreight, 1976)

Proses pembentukan *suffix tree ababc* digambarkan pada gambar di bawah ini:



Gambar 2.5: Pembentukan *Suffix Tree ababc* (McCreight, 1976)

Konsep dari pohon sufiks bisa dikembangkan untuk menyimpan lebih dari satu masukan *string*. Pengembangannya disebut juga dengan *Generalized Suffix Tree*. Terkadang dua atau lebih *input string* bisa memiliki sufiks yang sama, dalam kasus ini GST akan memiliki dua daun yang berhubungan dengan sufiks yang sama, setiap daun berkaitan dengan sebuah *input string* yang berbeda. Sebuah GST bisa dibangun dalam $O(n)$ ruang dan waktu di mana *n* adalah total jumlah seluruh *input string*. (Chi dan Hui, 1992).

Properti dari GST adalah sebagai berikut:

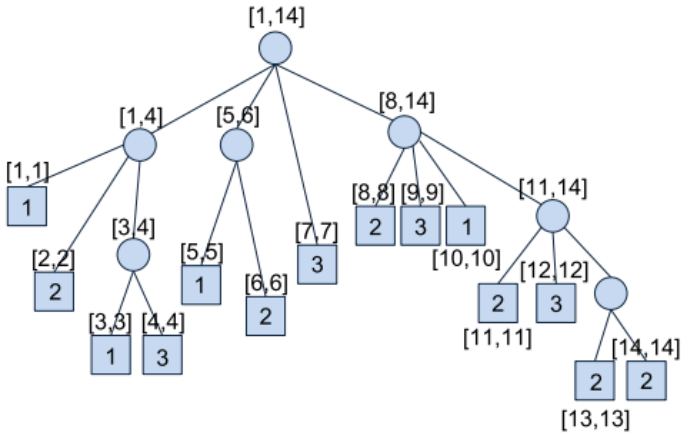
1. Setiap sufiks dari setiap *input string* diwakilkan oleh sebuah daun di dalam GST
2. Panjang dari tepi berlabel di dalam pohon sufiks dapat dicari dalam $O(1)$ waktu
3. Jika sebuah daun pada pohon sufiks mewakili sebuah string *V*, maka setiap sufiks dari *V* juga diwakilkan oleh sebuah daun lain di dalam pohon sufiks
4. Jika sebuah *node u* adalah leluhur dari *node v*, maka *string* yang *u* wakili adalah *prefix* dari *string* yang diwakili oleh *v*
5. Jika *node* $v_1, v_2, ..., v_k$ mewakili *string* $V_1, V_2, ..., V_k$, maka *least common ancestor* dari $v_1, ..., v_k$ merepresentasikan *longest common prefix* dari $V_1, ..., V_k$

(Chi dan Hui, 1992).

2. *Suffix Range*

Bayangkan ada sebuah subpohon atau *subtree* dari node *u* di dalam GST. Diketahui *v* dan *w* adalah daun paling kiri dan daun paling kanan dari subpohon ini. Jika *l* mewakili urutan dari *v* dan *r* mewakili urutan dari *w*, maka *v* adalah daun

paling kiri ke-*l* dan *w* adalah daun paling kanan ke-*r* di dalam GST. Jangkauan *[l,r]* adalah yang kita sebut *suffix range* dari *u*. (Hon dkk., 2010).



Gambar 2.6: *Suffix Range* (Hon dkk., 2010)

3. *Optimal Index for Colored Range Query*

Diketahui $A[1..n]$ adalah *array* dengan panjang *n*, di mana setiap entri menyimpan warna yang diambil dari $C = \{1, 2, ..., c\}$. Sebuah *colored range query* atau $CRQ(i, j)$, menerima 2 masukan integer yaitu *i* dan *j* di mana $1 \leq i \leq j \leq n$, dan memberikan hasil berupa *subarray* $A[i..j]$. Sebagai contoh, anggap *A* memiliki 7 entri, yang diwarnai dengan 1, 3, 2, 6, 2, 4, dan 5. Maka, $CRQ(2, 5)$ akan meminta set warna dari *subarray* $A[2..5]$ dan akan mengembalikan hasil 2,3,6. Indeks untuk *array A* dapat dibuat menggunakan $O(n)$ ruang penyimpanan, sehingga untuk *i* dan *j* apa pun, *colored range query* $CRQ(i, j)$, dapat dijawab dalam $O(\gamma)$ waktu, di mana γ menunjukkan jumlah warna berbeda pada set *output*. (Hon dkk., 2010).

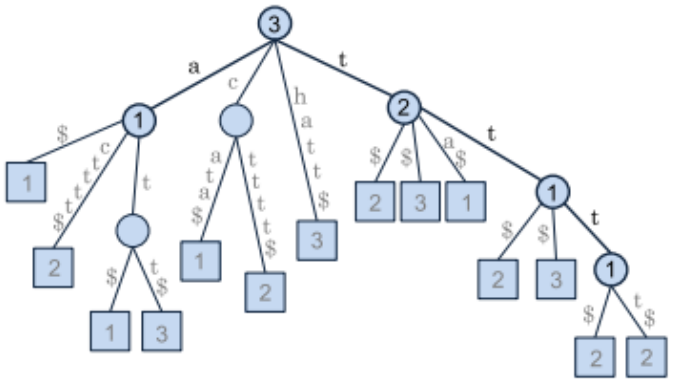
4. *Y-fast Trie for Efficient Successor Query*

Misalkan *S* adalah himpunan *n* bilangan bulat berbeda yang diambil dari $[1, D]$. Diberikan masukan *x*, *query* penerus atau *successor query* pada *S* mengembalikan bilangan bulat terkecil di *S* yang lebih besar dari atau sama dengan *x*. Sebuah index dari himpunan *S* yang tersusun dari *n* bilangan bulat dapat dibuat menggunakan $O(n)$ ruang penyimpanan, sehingga untuk setiap masukan *x*, *query* penerus $succ(S, x)$ dapat dijawab dengan $O(\log \log D)$ waktu, di mana *D* menunjukkan semesta tempat bilangan bulat *S* dipilih. Indeksnya bisa dibangun dalam waktu acak $O(n \log D)$. (Hon dkk., 2010).

E. *Induced Generalized Suffix Tree*

1. *IGST-f: IGST* Untuk Frekuensi *f*

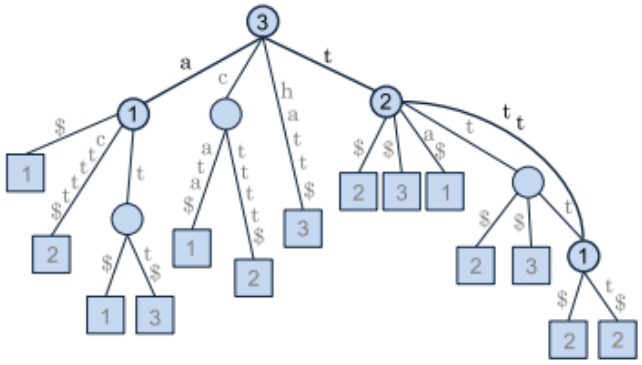
Misalkan ada GST untuk Δ dan sebuah bilangan bulat f dengan $1 \leq f \leq D$. Anggap setiap daun pada GST diberi label original dari sufiks yang sesuai. Untuk setiap *internal node v* dari GST, kita katakan *v* adalah *f-frequent* jika di dalam *subtree* yang berakar pada *v*, setidaknya ada f daun yang memiliki label yang sama. Subpohon terinduksi dari GST atau *induced subtree* yang dibentuk dengan mempertahankan semua *node* frekuensi-*f* disebut *pre-IGST-f*. Pada gambar 2.7 semua *internal node* yang memiliki setidaknya 2 daun dengan label yang sama ditandai dengan warna lebih terang atau *highlight*. Subpohon terinduksi yang terbentuk oleh *node* yang disorot ini adalah *pre-IGST-f*. (Hon dkk., 2010).



Gambar 2.7: *Pre-IGST-2* (Hon dkk., 2010)

Misalkan *v* adalah sebuah *node* di dalam *pre-IGST-f* dan *v* adalah sebuah *internal node* di dalam GST tersebut. Istilah *count(f, v)* digunakan untuk menyatakan jumlah dokumen berbeda yang memiliki label *f* dalam subpohon yang berakar pada *v* di GST. Istilah *count(v)* digunakan ketika konteksnya jelas. Pada *pre-IGST-2* yang ditunjukkan pada gambar 2.7, setiap *internal node v* diberi label dengan nilai *count(v)* yang sesuai. Nilai *count(v)* harus berada di antara 1 dan *m*, di mana *m* adalah jumlah dokumen dalam Δ. (Hon dkk., 2010).

Misalkan ada sebuah *pre-IGST-f.* Untuk setiap *internal node v*, kita katakan *v* redundan apabila *v* adalah *node* derajat-1 atau *degree-1 node* dan *count(v) = count(child(v))* di mana *child v* menyatakan *unique child* dari *v*. Pohon induksi atau *induced tree* yang dibentuk dengan menyusutkan atau mereduksi semua *node* redundan pada *pre-IGST-f* disebut *IGST-f.* (Hon dkk., 2010).



Gambar 2.8: *IGST-2* (Hon dkk., 2010)

Struktur dari *IGST-f* yang kita miliki ekuivalen dengan struktur indeks yang diperkenalkan oleh Muthukrishnan yang digunakan untuk menyelesaikan masalah *document mining problem*. Di mana diberikan frekuensi f dan pola P, lalu kembalikan occ_f dokumen yang memiliki setidaknya f kemunculan dari P dalam $O(|P| + occ_f)$ waktu. Dengan menggunakan struktur ini, kita sudah bisa menjawab masalah *inverse mine (P,k) query* dengan melakukan *binary search* pada *f*. Di mana setiap langkah dari *binary search* memeriksa apakah f tertentu adalah jawaban yang diinginkan untuk *inverse mine (P,k)*. Perlu kita ketahui bahwa setiap langkah dari *binary search* hanya perlu mengkonfirmasi apabila $k \geq occ_f$, sehingga kita hanya membutuhkan $O(|P| + k)$ waktu. Dalam total keseluruhan, jawaban yang diinginkan dapat diambil dalam $O((|P| + k) \log D)$ waktu.

2. Representasi *Array* dari *IGST-f*

Untuk menjawab permasalahan *inverse mine query* yang harus dilakukan adalah menemukan *locus* atau lokasi dari pola P di setiap IGST selama melakukan *binary search*. Untuk memangkas waktu pencarian *locus* pada setiap langkah, penggunaan *suffix ranges* dilakukan. Pertama-tama, anggap *suffix range* dari *locus* P sudah dikomputasi. Diketahui $[l_p, r_p]$ mewakilkan *range* ini jika *locus*nya ada.

Selanjutnya, anggap setiap *node* di dalam *IGST-f* terhubung dengan *suffix range* yang sesuai dengan *node* di dalam GST. Maka, *locus* dari *P*, jika ada, adalah *node v* sedemikian rupa sehingga *suffix range* yang berhubungan dengan keturunan dari *v* (termasuk *v*), adalah *subrange* dari $[l_p, r_p]$, dan *suffix range* yang berhubungan dengan *parent node* bukan merupakan *subrange* dari $[l_p, r_p]$. (Hon dkk., 2010).

Selanjutnya, lakukan *pre-order traversal* pada *IGST-f* dan hitung *suffix range* yang terhubung dengan sebuah *node* selagi *node* itu dikunjungi. Diketahui $[l(z), r(z)]$ mewakili *suffix range* dari *node* ke-z yang sudah dihitung selama proses *traversal*. Sekarang, anggap *locus* dari *P* dalam *IGST-f* (asumsikan ada) adalah *node* ke-*j* di dalam *pre-order traversal* dari *IGST-f*. Karena itu, *locus* dari *P* di dalam *IGST-f* memiliki *suffix range* yang berhubungan yang dinotasikan sebagai $[l(j), r(j)]$. Lebih lanjut, anggap kita menguji $[l(z), r(z)]$ untuk beberapa z. Ingat bahwa $[l_p, r_p]$ merupakan *suffix range* dari *locus P* di dalam GST. Maka, pernyataan di bawah ini adalah benar dan mengandung semua kemungkinan hubungan antara $l_p, r_p, l(z)$, dan $r(z)$. (Hon dkk., 2010).

1. Jika $r_p < l(z)$, maka $j < z$;
2. Jika $l_p > r(z)$, maka $j > z$;
3. Jika $[l(z), r(z)]$ adalah *subrange* dari $[l_p, r_p]$, maka $j \leq z$;
4. Jika $[l_p, r_p]$ adalah *subrange* dari $[l(z), r(z)]$, maka $j \geq z$.

Diketahui *c(z)* mewakili nilai *count* dari *node* ke-z yang dikunjungi selama masa *pre-order traversal* dari *IGST-f*. Alih-alih menyimpan *IGST-f* sebagai struktur pohon, kita akan merepresentasikannya ke dalam *array I*. Sehingga entri ke-z dari *I*, $I\,[z]$, menyimpan *3-tuple* yaitu $(l(z), r(z), c(z))$. Sebagai contoh, *array I* dari *IGST-2* pada gambar 2.8 adalah sebagai berikut:

Tabel 2.1: *Array I* dari IGST-2 pada gambar 2.8 (Hon dkk., 2010)

k	1	2	3	4
I	(1, 14, 3)	(1, 4, 1)	(8, 14, 2)	(13, 14, 1)

Berdasarkan teorema sebelumnya, kita bisa mendapatkan nilai *j* menggunakan *binary search* pada *array I*, sedemikian rupa sehingga $[l(j), r(j)]$ adalah *suffix range* yang terhubung dengan *locus P*. Lalu, nilai *c(j)* menyimpan jumlah dokumen dengan setidaknya *f* kemunculan dari *P*. Jika *locus P* dalam *IGST-f* tidak ada, maka tidak ada *z* sedemikian rupa dengan $[l(j), r(j)]$ yang merupakan *subrange* dari $[l_p, r_p]$ dan *binary search* akan mendeteksi ini. Dikarenakan jumlah *nodes* dalam *IGST-f* adalah *O(D)*, panjang *array I* juga *O(D)*, maka *binary search* akan memakan waktu sebanyak $O(\log D)$ waktu. (Hon dkk., 2010).

Sejatinya, kita bisa mempercepat waktu *query* dengan cara mengganti setiap *binary search* dalam *array IGST* dengan sebuah *succesor query* dalam sebuah *array* yang sedikit dimodifikasi. Akibatnya, waktu yang dihabiskan pada setiap kunjungan IGST akan dikurangi dari $O(\log D)$ menjadi $O(\log \log D)$. Langkah pertama, kita amati bahwa setiap *node* dalam IGST memiliki keterhubungan natural di dalam GST. Lebih tepatnya, sebuah *node u* dengan *suffix range [l,r]* yang berada dalam IGST menunjukkan bahwa sebuah *node u'* yang memiliki *suffix range* yang sama juga ada dalam GST original. Selanjutnya, kita lakukan *pre-order traversal* di dalam GST original agar setiap *node v* menerima kunjungan $\alpha(v)$ pertama kali dan $\beta(v)$ terakhir kali selama proses *traversal*. Lalu, setiap *node u* dalam IGST ditambah dengan informasi $\alpha(u')$ dan $\beta(u')$ yang mana *u'* adalah korespondensinya sendiri di dalam GST. Setelah itu, untuk setiap IGST, kita kumpulkan sebuah kumpulan nilai α dari semua *node* dan menyimpan sebuah *y-fast trie* agar *successor query* pada nilai α dapat dijawab dalam $O(\log \log D)$ waktu. (Hon dkk., 2010).

Sekarang, untuk melakukan pencarian kita harus mendapatkan *locus P* di dalam GST original, katakanlah u_p, yang waktu *pre-order traversal*nya adalah $\alpha(u_p)$ dan $\beta(u_p)$. Karena waktu traversal memiliki *nested property* yang bagus, untuk mencari *locus P* di *IGST-f* setara dengan mencari *successor* dari $\alpha(u_p)$ dalam himpunan nilai α dari *IGST-f*. Lebih tepatnya, diketahui *u* adalah *node* dalam *IGST-f* dengan $\alpha(u)$ menjadi *succesor* dari $\alpha(u_p)$. Mudah untuk membuktikan bahwa $\beta(u) \leq \beta(u_p)$ jika dan hanya jika *locus P* ada dalam *IGST-f* dengan *u* sebagai *locus*. Setelah *succesor query* di atas, kita dapat memeriksa *3-tuple* yang sesuai *l, r, c* dari *u* untuk menentukan berapa banyak dokumen yang mengandung setidaknya *f* kemunculan dari *P*. Dengan menyimpan GST dan semua *array I* untuk semua IGST-1, IGST-2,....,IGST-D, *inverse mine(P,k) query* dengan *P* adalah pola dan *k* adalah bilangan bulat dapat terjawab dalam $O(|P| + \log D \log\log D)$ waktu. (Hon dkk., 2010).

Seperti yang ditunjukkan oleh Muthukrishnan, jumlah *node* dalam *IGST-f* adalah *O(D/f)*. Secara singkat, setiap daun atau *node* derajat-1 di *IGST-f* sesuai dengan himpunan *disjoint* dari setidaknya f sufiks dari dokumen di δ, sehingga ada *O(D/f)*. Di sisi lain, jumlah *node* sisanya tidak bisa melebihi jumlah seluruh daun dan *node* derajat-1. Maka, jumlah seluruh *node* adalah *O(D/f)*. Total ruang yang digunakan oleh indeks Muthukrishnan dan Wing-Kai Hon jumlahnya sama yaitu $O(D \log D)$. (Hon dkk., 2010).

F. Indeks Efisien Untuk *Top-k Document Retrieval Problem*

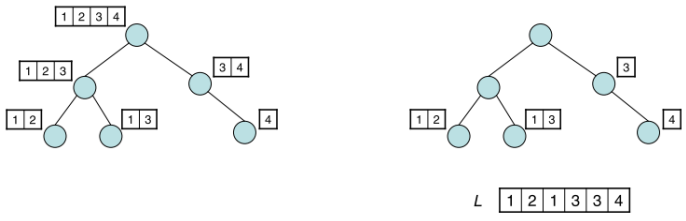
Ketika kita sudah bisa menemukan indeks untuk menjawab *inverse mine query*, kita juga bisa mencari kumpulan dokumen dari *top document(P,k)* yang mana dokumen-dokumen *k* tersebut adalah dokumen yang memiliki kemunculan *P* paling banyak. Langkah-langkah menyelesaikan *top document(P,k)*:

1. Cari *f** di mana *f** = *inverse mine(P,k)*. Setidaknya ada *k* dokumen dengan setidaknya *f** kemunculan *P*, tetapi ada kurang dari *k* dokumen dengan setidaknya *f** + 1 kemunculan *P*.

2. Keluarkan semua dokumen dengan setidaknya *f** + 1 kemunculan *P*. Biarkan *k'* menjadi jumlah dari dokumen-dokumen tersebut.

3. Keluarkan *k-k'* dokumen tambahan, berbeda dengan yang diperoleh pada langkah ke-2, yang mana memiliki setidaknya *f** kemunculan *P*.

Salah satu cara untuk menyelesaikan langkah 2 adalah dengan menambah setiap *node v* dari *IGST-f* dengan daftar dokumen *count(v)* terkait, masing-masing dari yang memiliki setidaknya *f* label di subpohon yang berakar pada *v*. Kemudian, mudah untuk melihat bahwa untuk menjawab langkah 2, kita hanya perlu mencari *locus P* dalam *IGST-(f*+1)* dan mengembalikan daftar dokumen dalam *locus*. Metode ini membutuhkan waktu optimal *O(k')* untuk mengembalikan dokumen. Sayangnya, pada skenario terburuk, kita membutuhkan ruang tambahan untuk proses penambahan sebesar $O(Dm \log D)$ di mana *m* mewakili jumlah dokumen dalam Δ. Metode ini kita sebut dengan *Heuristic I*. (Hon dkk., 2010).



Gambar 2.9: Contoh dari penggabungan *list L* (Hon dkk., 2010)

Salah satu cara lain yang lebih baik untuk menyelesaikan langkah 2 adalah dengan menerapkan indeks Muthukrishnan yang digunakan untuk *colored range query* sebagai struktur data tambahan. Cara ini digunakan oleh Muthukrishnan

dalam menyelesaikan *document mining problem*. Idenya adalah untuk setiap *node v* di *IGST-f* kita hanya menyimpan *sublist* tereduksi dari dokumen *count(v)* terkait, di mana setiap dokumen tersebut tidak muncul dalam daftar keturunan dari *v* di *IGST-f*. Selanjutnya, kita melakukan *pre-order traversal* dan menggabungkan daftar *node* yang telah dikunjungi ke dalam *list L* (lihat gambar 2.9). Kemudian, mudah untuk memeriksa bahwa setiap *node v* akan berkorespondensi ke *subrange* dalam *list L*, sehingga dokumen *count(v)* yang terkait akan sesuai persis dengan *count(v)* yang berbeda dokumen pada *subrange*. Sebagai contoh, perhatikan anak kiri dari akar di *IGST-f* pada gambar 2.9, itu berkorespondensi dengan *subrange* $L\,[1...4]$, yang merupakan rangkaian dari *sublist* tereduksi di semua turunannya (termasuk dirinya sendiri). Kita melihat bahwa dokumen terkait-1, 2, 3-akan sesuai dengan dokumen yang berbeda di $L\,[1...4]$. (Hon dkk., 2010).

Jadi, jika kita menggunakan indeks Muthukrishnan untuk menyimpan *L*, dan untuk setiap *node*, kita menyimpan posisi awal dan akhir di *L* untuk *subrange* terkait, langkah 2 juga dapat dijawab secara optimal dalam *O(k')* waktu seperti dalam *Heuristic I* tetapi dengan kebutuhan ruang yang lebih kecil yaitu $O\,(D \log D)$. (Hon dkk., 2010).

Langkah 3 dapat diselesaikan dengan cara yang sama seperti pada langkah 2. Kita amati bahwa setiap *k* dokumen dengan setidaknya *f* kemunculan *P*, bersama-sama dengan *k'* dokumen yang diperoleh dari langkah 2, harus berisi kumpulan *k* dokumen yang kita inginkan untuk *top document(P,k)*. Jadi pada langkah 3, kita akan secara sewenang-wenang memilih satu set *k* dokumen dengan setidaknya *f* kemunculan *P*, yang mana selanjutnya kita pilih *k-k'* dokumen yang tidak diperoleh pada langkah 2. Cara sederhana untuk menyelesaikan bagian terakhir adalah dengan mengurutkan, yang mana membutuhkan $O(min\,\{m, k \log k\})$ waktu. Untuk mempercepat, kita pertahankan *bit-vector* dari *m*

bit, di mana bit ke-*i* berkorespondensi dengan dokumen T_i, dengan semua bit diset menjadi 0 pada awalan. Ketika langkah 2 selesai, kita tandai setiap bit yang berkorespondensi dengan *k'* dokumen dengan 1. Lalu, pada langkah 3, ketika sebuah dokumen diperiksa pada prosedur akhir, kita bisa periksa *bit-vector* untuk melihat apakah dokumen tersebut sudah diperoleh pada langkah 2, sehingga kita bisa dengan mudah mengambil set *k-k'* dokumen yang kita inginkan. Setelah langkah 3, kita bisa me-*reset bit vector* dalam *O(k)* waktu dengan mengacu pada *final output* atau hasil akhir. Dengan demikian, kita telah menyelesaikan permasalahan *top document query*. Sebuah indeks yang memerlukan $O(D \log D)$ ruang untuk Δ dapat dipertahankan, sehingga untuk semua masukan pola *P* dan semua masukan bilangan bulat *k* dalam *top document(P,k)* dapat dijawab dalam $O(|P| + \log D \log\log D + k)$ waktu. (Hon dkk., 2010).

G. Konstruksi Algoritma Pengindeksan

Kita bisa menghitung nilai *count* lebih efisien dengan menggunakan penghitungan masalah *color set size*. Langkah pertama adalah membuat *LeafList* dan menghitung *leafcount()* dari pohon *T*. Selanjutnya hitung nilai *lastleaf()* dengan memanfaatkan *LeafList*. Untuk setiap warna *c* kita simpan indeks daun berwarna *c* paling terakhir yang kita temui. Nilai awalnya adalah *Nil* dan ketika kita bertemu dengan daun berwarna *c* kita simpan indeksnya dan mengubah nilai *lastleaf()* menjadi indeks daun tersebut. (Chi dan Hui, 1992).

Selanjutnya kita hitung *CPLCount()* dengan menginisialisasi nilainya dengan 0. Untuk setiap daun *x*, kita hitung *u = lca(x, lastleaf(x))* dan menambahkan nilai *CPLCount(u)* dengan 1. Terakhir, kita gunakan *post-order traversal* untuk menghitung semua nilai *duplicate()* dengan hubungan rekursif $duplicate(u) = CPLCount(u) + \sum_{w \in W} duplicate(w)$ di mana *W* adalah

himpunan anak dari *u*. Setelah mendapatkan nilai *leafcount()* dan *duplicate()* maka menghitung nilai *count* adalah dengan mengurangkan nilai tersebut. (Chi dan Hui, 1992).

Penerapannya dilakukan misal ada sebuah IGST-*f* tertentu dengan *sublist* tereduksi yang dibuat. Tahap *pre-processing* yang dilakukan adalah:

1. Di dalam setiap *node*, tuliskan jumlah dokumen di dalam *sublist* yang tereduksi tersebut.

2. Lakukan *bottom-up traversal* di IGST-*f*, sehingga setiap *node v* memperoleh jumlah total n_v dokumen di dalam *sublist* tereduksi dari semua turunannya (termasuk dirinya sendiri).

Nilai n_v pada setiap *node v* setelah langkah ke-2 pada tahap *pre-processing* di atas berkaitan erat dengan nilai *count* yang diinginkan. Ketika nilai *count* menghitung setiap dokumen berbeda dalam *sublist* turunan yang tereduksi sebanyak satu kali, nilai n_v dapat menghitung sebuah dokumen beberapa kali. Untuk memperbaiki hitungan, ada beberapa hal yang harus diperhatikan. Misalkan dokumen *i* muncul di *sublist* tereduksi dari *node y*. Keturunan *y* ini dapat dimisalkan dengan $u_1, u_2, ..., u_y$ terurut berdasarkan kemunculan pertamanya pada *pre-order traversal*. Lebih lanjut, misalkan dokumen *i* muncul di dalam *sublist* tereduksi persis di j_i keturunan dari *node v* (termasuk dirinya sendiri), maka kita memiliki:

1. Terdapat bilangan bulat *z* sedemikian rupa sehingga keturunan j_i dari *v* sama dengan $u_{z+1}, u_{z+2}, ..., u_{z+j_i}$.

2. Nenek moyang terendah atau *lowest common ancestor* dari u_g dan u_{g+1} ada di subpohon *v* jika dan hanya jika $g \in \{z+1, z+2, ..., z+j_i-1\}$.

Implikasi dari hal-hal di atas adalah misalkan j_i' menyatakan jumlah g sedemikian rupa sehingga *lowest common ancestor* dari u_g dan u_{g+1} ada di subpohon *v*. Maka $j_i - j_i' = 1$ ketika $j_i \geq 1$ dan $j_i - j_i' = 0$ ketika $j_i = 0$. Dengan kata lain, $j_i - j_i'$ selalu mengindikasikan apakah ada tidaknya dokumen *i* dalam subpohon *v*. (Hon dkk., 2010).

Ingat bahwa n_v adalah jumlah total dokumen dalam *sublist* tereduksi di semua turunan *v*. Berdasarkan pernyataan di atas, kita dapat melihat bahwa nilai hitungan untuk *node v* yang merupakan jumlah dokumen yang berbeda dalam *list* yang direduksi di subpohon dari *v* dapat dihitung dengan $\sum_i (j_i - j_i') = \sum_i j_i - \sum_i j_i' = n_v - \sum_i j_i'$ di mana *i* mencakup semua dokumen. (Hon dkk., 2010).

Nilai perbaikan dari $\sum_i j_i'$ untuk setiap *node v* pada IGST-*f* dapat diperoleh dengan cara yang sama seperti ketika mencari nilai n_v. Detailnya adalah sebagai berikut:

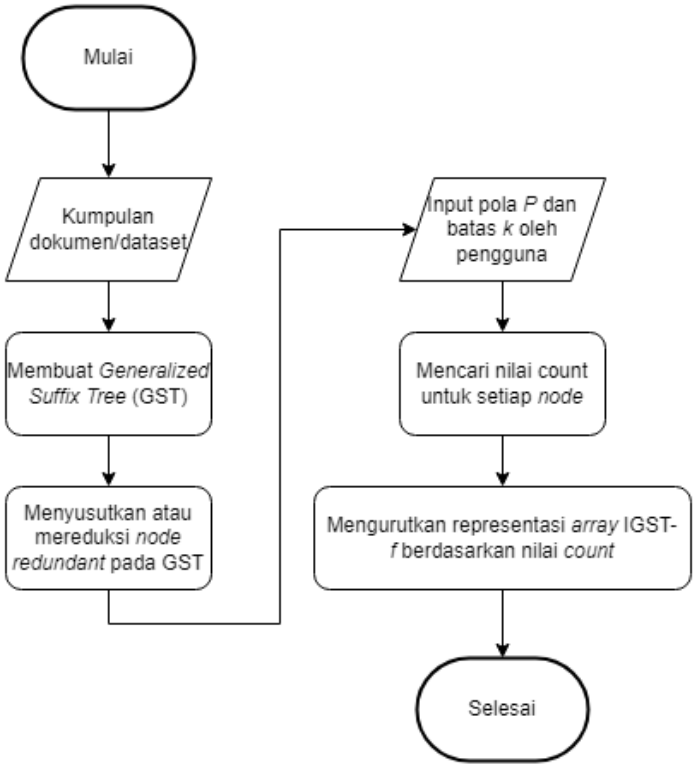
1. Tetapkan penghitung atau *counter* untuk setiap *node*, yang diinisialisasi ke 0.
2. Proses setiap dokumen *i* seperti di bawah ini:
   (a) Temukan *node* $u_1, u_2, ..., u_y$ yang berisi dokumen *i* dalam *sublist* tereduksinya, di mana *node* diurutkan berdasarkan kemunculan pertama dalam *pre-order traversal*.
   (b) Tambahkan 1 ke penghitung atau *counter* di *lowest common ancestor* dari u_g dan u_{g+1}, untuk semua g.
3. Lakukan *bottom-up traversal* pada IGST-*f*, sehingga untuk setiap *node v* memiliki jumlah waktu ketika turunan *v* menjadi *lowest common ancestor* dari beberapa pasangan pada langkah ke-2.

Nilai yang disimpan di setiap *node v* setelah prosedur di atas adalah nilai yang diinginkan untuk $\sum_i j'_i$. Pada akhirnya, dengan mengurangkan nilai n_v dengan $\sum_i j'_i$ yang berhubungan pada setiap *node v*, kita bisa mendapatkan nilai *count* yang kita inginkan untuk setiap *node v*. (Hon dkk., 2010).

BAB III

METODOLOGI PENELITIAN

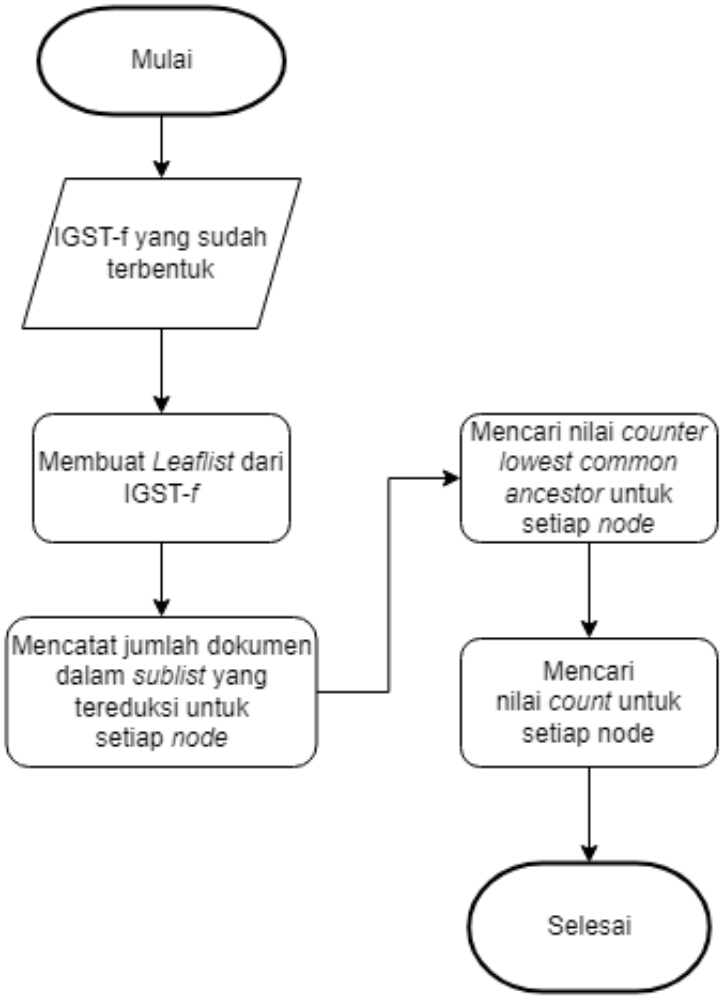
A. *Flowchart* Algoritma



Gambar 3.1: *Flowchart* Algoritma Indeks Efisien Untuk *Top-K Document Retrieval Problem*

Diagram alir dari algoritma pengindeksan dimulai masukan berupa kumpulan dokumen yang akan dibuat menjadi GST. Langkah selanjutnya yaitu membuat GST dari kumpulan dokumen tersebut. Seperti yang dijelaskan pada subbab 2D yaitu *Generalized Suffix Tree*, GST menjadi struktur data awal dari kumpulan dokumen. Konstruksi dari GST juga dijelaskan pada subbab tersebut. Dari GST tersebut kemudian dilakukan penyusutan atau reduksi untuk *node* yang redundan seperti

yang sudah dijelaskan pada subbab 2E sehingga akan terbentuk pohon yang terinduksi untuk frekuensi *f* yang bernama *Induced Generalized Suffix Tree-f* atau IGST-*f*. IGST-*f* sendiri seperti yang dijelaskan pada bagian akhir subbab 2B mengenai *Document Retrieval* sebelumnya, menjadi kunci atau komponen utama dari pengindeksan. Langkah selanjutnya adalah program menerima masukan berupa pola kata dan batas *k* untuk dicari pada kumpulan dokumen. Dari sini, kita akan mencari nilai *count* dari setiap *node*.



Gambar 3.2: *Flowchart* Pencarian Nilai *Count*

Pertama-tama kita harus membuat *Leaflist* dari IGST-*f* yang sudah terbentuk.

Kemudian, mencatat jumlah dokumen dalam *sublist* yang tereduksi untuk setiap *node*. Selanjutnya, mencari nilai *counter lowest common ancestor* dari setiap *node*. Dari sini, kita bisa mendapatkan nilai *count* yang merupakan jumlah dokumen berbeda yang memiliki kemunculan pola *P* untuk setiap *node*.

Langkah terakhir yang dijelaskan pada subbab 2F mengenai Indeks Efisien Untuk *Top-k Document Retrieval Problem*, yaitu mengurutkan representasi *array* IGST-f yang sudah memiliki nilai *count* dan mengembalikan hasil top-k dokumen yang memiliki pola *P*.

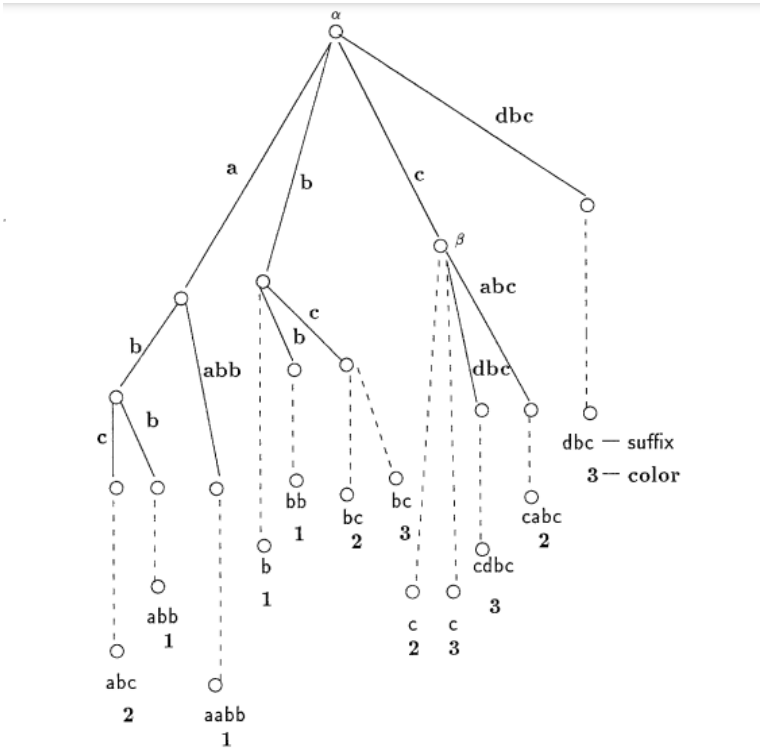
1. *Pseudocode* Algoritma

Di bawah ini merupakan *pseudocode* dari *flowchart* algoritma di atas

Algorithm 1 *Efficient Index For Retrieving Top-k Most Frequent Document* Algorithm

$p = string(patternP)$
$k = integer(limit)$
$D = documents$
$gst = []$
$nv = integer$
$lcaCounter = 0$
$countValue = 0$
$result = []$
for all document in D **do**
    add documents to gst
end for
for all document in gst **do**
    if $document$ is redundant **then**
        contracting document ▷ This is a process for making IGST
    end if
end for
for all document in gst **do**
    count nv
    if $document$ is lowest common ancestor **then**
        $lcaCounter+ = 1$
    end if
    $countValue = nv - lcaCounter$
    add document to $result$
end for
sort result
for i in range (0, k) **do**
    return result[i]
end for

B. Contoh Konstruksi *Generalized Suffix Tree*



Gambar 3.3: Contoh Konstruksi *Generalized Suffix Tree*

Pada pohon tersebut terdapat 3 warna dan sufiks-sufiks yang tersusun dari pola-pola *string* yang ada. Terdapat pola *aabb* dengan warna 1, *cabc* dengan warna 2, *cdbc* dengan warna 3. Sufiks *bc* dan *c* merupakan sufiks yang paling sering ditemukan untuk warna 2 dan 3. Nantinya, GST akan memuat seluruh sufiks dari seluruh dokumen dan pengindeks atau *indexer* akan mencari pola kata kunci yang diberikan di dalam GST ini.

C. Alat dan Bahan Penelitian

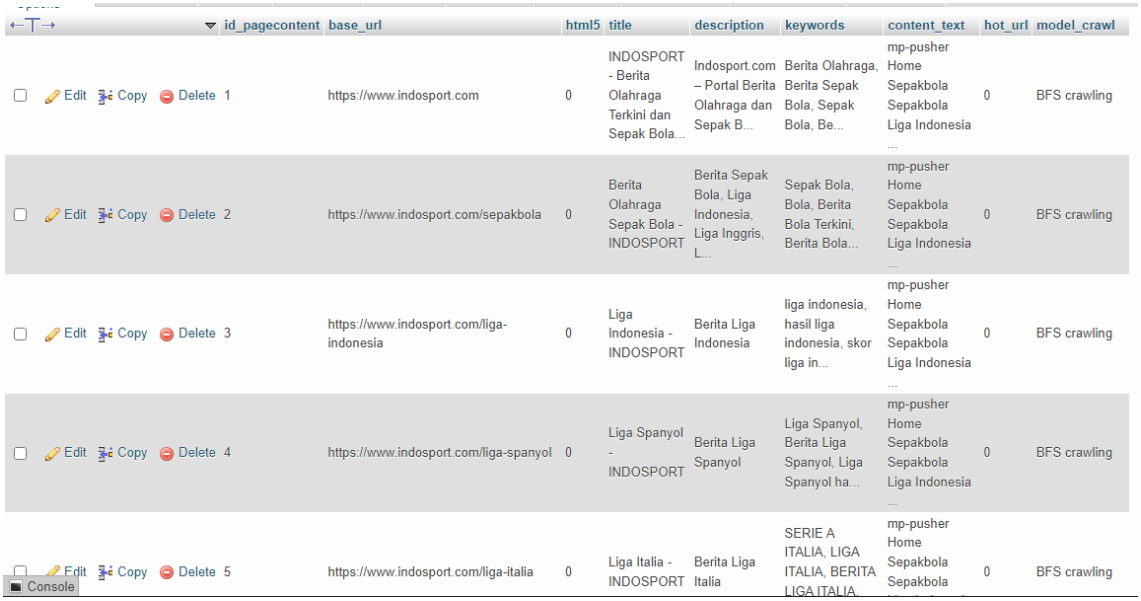
Dalam penelitian ini, beberapa alat berupa perangkat keras yang digunakan untuk menunjang pembuatan modul pengindeks adalah sebagai berikut:

1. Laptop ASUS X550D dengan CPU AMD A8 dan RAM 10GB
2. Koneksi berbasis *Wi-Fi*

Selain perangkat keras, perangkat lunak yang dipakai untuk penelitian ini adalah sebagai berikut:

1. Windows 10 64-bit Operating System
2. Visual Studio Code sebagai *code editor*
3. XAMPP untuk menjalankan *MySQL database*
4. Python 3 untuk menjalankan program Python

D. Dataset



	id_pagecontent	base_url	html5	title	description	keywords	content_text	hot_url	model_crawl
Edit Copy Delete	1	https://www.indosport.com	0	INDOSPORT - Berita Olahraga Terkini dan Sepak Bola...	Indosport.com – Portal Berita Olahraga dan Sepak B...	Berita Olahraga, Berita Sepak Bola, Sepak Bola, Be...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
Edit Copy Delete	2	https://www.indosport.com/sepakbola	0	Berita Olahraga Sepak Bola - INDOSPORT	Berita Sepak Bola, Liga Indonesia, Liga Inggris, L...	Sepak Bola, Bola, Berita Bola Terkini, Berita Bola...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
Edit Copy Delete	3	https://www.indosport.com/liga-indonesia	0	Liga Indonesia - INDOSPORT	Berita Liga Indonesia	liga indonesia, hasil liga indonesia, skor liga in...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
Edit Copy Delete	4	https://www.indosport.com/liga-spanyol	0	Liga Spanyol - INDOSPORT	Berita Liga Spanyol	Liga Spanyol, Berita Liga Spanyol, Liga Spanyol ha...	mp-pusher Home Sepakbola Sepakbola Liga Indonesia ...	0	BFS crawling
Edit Copy Delete	5	https://www.indosport.com/liga-italia	0	Liga Italia - INDOSPORT	Berita Liga Italia	SERIE A ITALIA, LIGA ITALIA, BERITA LIGA ITALIA,	mp-pusher Home Sepakbola Sepakbola	0	BFS crawling

Console

Gambar 3.4: 5 Data Pertama Dataset pada Tabel *Page Information*

Pada penelitian ini, dataset yang akan digunakan sebagai kumpulan dokumen adalah tabel *page information* khususnya kolom *content text*. Dataset ini disusun oleh

Muhammad Fathan Qoriiba dalam penelitiannya yang berjudul PERANCANGAN *CRAWLER* SEBAGAI PENDUKUNG PADA *SEARCH ENGINE*. Dataset ini disusun menggunakan *tools crawling* yang telah dikembangkan. Situs yang digunakan untuk proses *crawling* adalah situs *indosport.com* dan *curiouscuisiniere.com*. Total data yang tersimpan di dalam dataset tersebut berukuran 176 MiB atau sekitar 185 MB.

Ada 8 tabel yang tersimpan di dalam dataset ini dengan rincian sebagai berikut:

1. Tabel *page information* berisi 1060 baris halaman web dengan 8 atribut.

Tabel 3.1: Deskripsi Tabel *Page Information*

Atribut	**Deskripsi**
base url	Pranala dari halaman web
html5	Menyatakan halaman tersebut tersusun dari HTML5 atau bukan
title	Judul halaman web
description	Deskripsi halaman web
keywords	Kata kunci dari halaman web
content text	Isi konten dari halaman web
hot url	Menyatakan halaman web tersebut termasuk *hot url* atau tidak
model crawl	Menyatakan cara *crawling* yang digunakan pada halaman web

2. Tabel *Linking* berisi 131.581 baris *linking* dengan 3 atribut.

Tabel 3.2: Deskripsi Tabel *Linking*

Atribut	**Deskripsi**
crawl id	ID dari *crawl*
url	Pranala atau *link*
outgoing link	Pranala lain yang terhubung dengan atribut *url*.

3. Tabel *Style Resource* berisi 7.530 baris *css* dengan 2 atribut.

Tabel 3.3: Deskripsi Tabel *Style Resource*

Atribut	Deskripsi
base url	Pranala dari halaman web
style	*CSS* dari halaman web

4. Tabel *Script Resource* berisi 29.732 baris *js* dengan 2 atribut.

Tabel 3.4: Deskripsi Tabel *Script Resource*

Atribut	Deskripsi
base url	Pranala dari halaman web
script	*Script JS* dari halaman web

5. Tabel *forms* berisi 2.476 baris *form* dengan 2 atribut.

Tabel 3.5: Deskripsi Tabel *Forms*

Atribut	Deskripsi
base url	Pranala dari halaman web
form	*Script form* dari halaman web

6. Tabel *images* berisi 38.644 baris *image* dengan 2 atribut.

Tabel 3.6: Deskripsi Tabel *Images*

Atribut	Deskripsi
base url	Pranala dari halaman web
image	*Script image* dari halaman web

7. Tabel *List* berisi 68.341 baris *list* dengan 2 atribut.

Tabel 3.7: Deskripsi Tabel *List*

Atribut	Deskripsi
base url	Pranala dari halaman web
list	*Script list* dari halaman web

8. Tabel *Tables* berisi 75 baris *table* dengan 2 atribut.

Tabel 3.8: Deskripsi Tabel *Tables*

Atribut	Deskripsi
base url	Pranala dari halaman web
table	*Script table* dari halaman web

Nantinya akan dilakukan sedikit pembersihan pada data dengan menghilangkan isi-isi yang kurang berkaitan dengan artikel seperti teks iklan dan rekomendasi judul artikel lain di tengah isi konten. Data konten halaman web yang sudah bersih kemudian akan dijadikan acuan sebagai kumpulan dokumen yang akan digunakan untuk proses penelitian ini.

E. Tahapan Pengembangan

Pada penelitian ini, penulis akan membuat modul pengindeks atau *indexer* yang menerima *input string* berupa kata kunci atau pola kata dan *integer input* berupa batasan berapa dokumen yang dikembalikan sebagai hasil. *Indexer* akan mengembalikan beberapa dokumen sesuai dengan batasan yang diberikan dan dengan relevansi paling tinggi yang mengandung kata kunci tersebut. Algoritma yang akan digunakan adalah algoritma *Efficient index for retrieving top-k most frequent documents*.

Penelitian ini merupakan penelitian pendukung dari keseluruhan penelitian *search engine*. Penelitian induk dari rangkaian penelitian *search engine* antara lain PERANCANGAN *CRAWLER* SEBAGAI PENDUKUNG PADA *SEARCH ENGINE* oleh Muhammad Fathan Qoriiba dan PERANCANGAN ARSITEKTUR *SEARCH ENGINE* DENGAN MENGINTEGRASIKAN *WEB CRAWLER*, *ALGORITMA PAGE RANKING*, DAN *DOCUMENT RANKING* oleh Lazuardy Khatulistiwa.

Pembuatan modul pengindeks pada penelitian ini akan dilakukan bertahap sesuai dengan setiap proses pada *flowchart* 3.1. Waktu dari pembuatan setiap langkah atau proses diestimasikan 1-2 minggu dan paling maksimal 3 minggu tergantung kompleksitas dari proses tersebut. Jika penelitian ini selesai lebih dulu dibandingkan penelitian induknya yaitu PERANCANGAN ARSITEKTUR *SEARCH ENGINE* DENGAN MENGINTEGRASIKAN *WEB CRAWLER*, *ALGORITMA PAGE RANKING*, DAN *DOCUMENT RANKING* yang dibuat oleh Lazuardy Khatulistiwa, maka penelitian ini akan berjalan secara otonom dan hasil akhirnya adalah modul pengindeks. Namun, jika penelitian induknya selesai lebih dulu dibandingkan penelitian ini, maka struktur direktori serta *design class* dan fungsi dari penelitian induk akan diikuti skemanya atau dengan kata lain modul pengindeks akan dimasukkan ke dalam arsitektur *search engine* besar.

Untuk pengujian dari hasil penelitian ini, akan ada dua poin penting pengujian, yaitu:

1. Waktu pengindeksan atau *indexing time* yaitu kecepatan pengindeksan dari awal proses hingga mengembalikan hasil. Indikator modul pengindeks berjalan dengan baik adalah waktu pengembalian dokumen dalam waktu singkat yaitu hitungan detik.

2. Relevansi pencarian yaitu kesesuaian hasil pencarian yang dihasilkan oleh program dengan pola kata kunci yang diberikan pada saat awal memasukkan

input. Indikator modul pengindeks berjalan dengan baik adalah dokumen yang dikembalikan relevan dengan kata yang dicari oleh pengguna.

Untuk waktu pengindeksan, skenarionya adalah sebagai berikut:

1. *Developer* memasukkan 5 *input* pola kata kunci dengan panjang yang berbeda-beda dan *input* limit *k* sebesar 3, 5, dan 7 sebagai batasan berapa dokumen yang dikembalikan.
2. Program mencari dokumen yang memiliki pola *input*.
3. Program mengembalikan top-*k* dokumen yang memiliki pola *input*.
4. *Developer* mengukur dan membandingkan waktu yang dihabiskan dari mulai awal proses hingga pengembalian dokumen dengan *input* yang berbeda-beda.

Untuk relevansi pencarian dibutuhkan *tester* lain selain developer untuk menilai relevansi dari dokumen yang dikembalikan dengan pola *input* yang diberikan. Skenario eksperimennya adalah sebagai berikut:

1. *Tester* memasukkan 3 *input* pola kata kunci dengan panjang yang berbeda-beda dan *input* limit *k* sebesar 3 sehingga hanya 3 dokumen paling relevan yang dikembalikan.
2. Program mencari dokumen yang memiliki pola *input*.
3. Program mengembalikan top-*k* dokumen yang memiliki pola *input*.
4. *Tester* menilai relevansi hasil pencarian pola pada dokumen.

Tester berjumlah 3 orang dan pola kata kunci masukan atau *input* akan sama untuk setiap *tester*.

DAFTAR PUSTAKA

Al Aziz, M. M., Thulasiraman, P., dan Mohammed, N. (2022). Parallel and private generalized suffix tree construction and query on genomic data. *BMC genomic data*, 23(1):1–16.

Brin, S., Motwani, R., Page, L., dan Winograd, T. (1998). What can you do with a web in your pocket? *IEEE Data Eng. Bull.*, 21(2):37–47.

Brin, S. dan Page, L. (1998). The anatomy of a large-scale hypertextual web search engine.

Chi, L. dan Hui, K. (1992). Color set size problem with applications to string matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 230–243. Springer.

Harman, D. dkk. (2019). Information retrieval: the early years. *Foundations and Trends® in Information Retrieval*, 13(5):425–577.

Hon, W.-K., Patil, M., Shah, R., dan Wu, S.-B. (2010). Efficient index for retrieving top-k most frequent documents. *Journal of Discrete Algorithms*, 8(4):402–417.

Mahdi, M. S. R., Al Aziz, M. M., Mohammed, N., dan Jiang, X. (2021). Privacy-preserving string search on encrypted genomic data using a generalized suffix tree. *Informatics in Medicine Unlocked*, 23:100525.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272.

Muthukrishnan, S. (2002). Efficient algorithms for document retrieval problems. In *SODA*, volume 2, pages 657–666. Citeseer.

Puglisi, S. J. dan Zhukova, B. (2021). Document retrieval hacks. In *19th International Symposium on Experimental Algorithms (SEA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Qoriiba, M. F. (2021). Perancangan crawler sebagai pendukung pada search engine.

Seymour, T., Frantsvog, D., Kumar, S., dkk. (2011). History of search engines. *International Journal of Management & Information Systems (IJMIS)*, 15(4):47–58.

StatCounter (2022). Search engine market share worldwide.

Zaky, A. dan Munir, R. (2016). Full-text search on data with access control using generalized suffix tree. In *2016 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.