BAB III

LANDASAN TEORI

A. Text Detection

Text detection dapat diartikan sebagai suatu proses pencarian teks yang terdapat pada sebuah citra. Proses *text detection* dalam sebuah citra merupakan salah satu masalah yang sering dihadapi dan selalu menjadi pusat perhatian karena proses ini sangat dibutuhkan pada aplikasi *computer vision* seperti pencarian teks pada gambar (Neumann & Matas, 2012).

Pada penelitian ini proses *text detection* akan dibantu dengan menggunakan *library* OpenCV. OpenCV adalah suatu library yang terdiri dari fungsi-fungsi *computer vision* dan API *(Aplication Programming Interface)* untuk melakukan pengolahan citra baik secara *high level* maupun *low level* dan sebagai optimasi dalam menjalankan aplikasi secara *realtime*. OpenCV sangat disarankan pada pengembangan aplilkasi *computer vision*, karena *library* ini mampu menciptakan aplikasi yang handal, kuat di bidang *digital vision* dan mempunyai kemampuan yang mirip dengan cara pengolahan visual pada manusia. Karena OpenCV bersifat *open source*, maka *library* ini dapat digunakan secara gratis.

Proses *text detection* pada OpenCV dikembangkan berdasarkan teori yang dikemukakan oleh Lukas Neumann dan Jiri Matas (2012). Proses *text detection* pada OpenCV terdiri dari dua tahapan. Tahap pertama, secara bertahap sistem akan menghitung *descriptor*, berupa: area, parameter, bounding box, dan nilai euler. Hasi dari perhitungan *descriptor* akan menghasilkan sejumlah *region* r

dimana setiap region tersebut akan dijadikan ciri (*features*) dalam melakukan proses pendeteksian teks. Hanya *External Regions r* (ERs) yang memenuhi kondisi tertentu yang akan digunakan pada tahap selanjutnya. Pada tahap ke-dua, ERs yang dipilih akan diklasifikasikan ke dalam kategori karakter dan bukan karakter. Proses ini membutuhkan komputasi yang lebih banyak dibandingkan komputasi sebelumnya. Proses yang digunakan yaitu: *Hole area ratio, convex hull ratio*, dan menghitung jumlah batasan terluar dari infleksi poin. Setelah penyaringan ERs selesai dilakukan, maka langkah selanjutnya adalah melakukan pengelompokan ERs ke dalam *high-level text blocks*, seperti: kata, kalimat, maupun paragraf.

B. Thresholding

Saif et al., (2013) dalam jurnalnya menjelaskan bahwa *thresholding* adalah suatu metode yang digunakan untuk memisahkan antara objek dan *background*-nya. *Thresholding* digunakan untuk mengatur jumlah derajat keabuan (*grayscale*) yang ada pada citra (Gupta et al., 2012). Citra dari hasil *thresholding* biasanya sering dipakai lebih lanjut untuk proses ekstraksi fitur serta pengenalan objek.

Teknik *thresholding* dapat dibedakan atas 2 (dua) macam, yaitu: *Global Thresholding* dan *Local Adaptive Thresholding*. Pada *Global Thresholding*, proses dilakukan dengan mengelompokan warna piksel menggunakan sebuah nilai ambang (*threshold*) yang berlaku untuk seluruh bagian pada citra. Sedangkan pada *Local Adaptive Thresholding*, proses dilakukan dengan membagi citra

menjadi beberapa sub citra, kemudian pada setiap sub citra akan dilakukan proses segmentasi menggunakan nilai *threshold* yang berbeda (Al-amri et al., 2010).

Metode *thresholding* yang digunakan pada penelitian ini adalah metode *Global Thresholding*. Pada *Global Thresholding*, metode yang digunakan adalah dengan menentukan suatu nilai ambang (*threshold*) yang digunakan untuk mengelompokan warna piksel. Seluruh piksel pada citra akan dikonversikan menjadi hitam atau putih, dengan ketentuan dimana piksel yang levelnya lebih tinggi dari level *threshold* akan diubah menjadi putih, dan sebaliknya piksel yang levelnya berada di bawah dari level *threshold* akan diubah menjadi hitam. Secara umum, persamaan untuk mengubah citra *grayscale* menjadi citra biner dapat ditulis sebagai berikut:

$$g(x,y) = \begin{cases} 1 \ iff(x,y) > T \\ 0 \ iff(x,y) \leq T \end{cases} \qquad [3,1]$$

dengan *g(x,y)* adalah citra biner dari citra *grayscale f(x,y)*, dan T menyatakan nilai ambang (*threshold*). Nilai T memegang peranan sangat penting dalam proses pengambangan. Kualitas hasil citra biner sangat tergantung dari nilai T yang digunakan.

Berikut merupakan contoh citra *grayscale* yang diubah ke dalam bentuk citra biner dengan menggunakan teknik *thresholding*:

Gambar 3.1 Contoh hasil Thresholding

Dikutip Dari: Ramadijanti N., Setiawardhana,& Mahsun H. (2009). Social Network (http://digilib.its.ac.id/public/ITS-NonDegree-7548-7405040015-bab2.pdf)

Gambar sebelah kiri merupakan gambar *original* dan gambar sebelah kanan adalah gambar hasil dari *thresholding*

C. Segmentasi Citra

Segmentasi citra *(image segmentation)* merupakan bagian dari proses pengolahan citra. Proses segmentasi citra merupakan suatu proses pra pengolahan pada sistem pengenalan objek dalam citra. Segmentasi citra mempunyai arti membagi suatu citra menjadi wilayah-wilayah yang homogen berdasarkan kriteria tertentu antara tingkat keabuan suatu piksel dengan tingkat keabuan piksel–piiksel tetangganya. Hasil dari proses segmentasi ini akan digunakan untuk proses tingkat lebih lanjut yang dapat dilakukan pada suatu citra, misalnya proses klasifikasi citra ataupun proses identifikasi suatu obyek (Saif et al., 2013 ; Al-amri et al., 2010 ; Gupta et al., 2012).

Saif et al. (2013) berpendapat bahwa segmentasi citra adalah suatu proses pembagian daerah citra menjadi beberapa bagian yang lebih kecil berdasarkan tata letak piksel dan intensitasnya yang saling berdekatan. Segmentasi merupakan

suatu bagian yang sangat penting dalam analisis citra secara otomatis, karena pada prosedur ini obyek hasil segmentasi akan digunakan untuk proses selanjutnya, misalnya: pada pengenalan pola.

Beberapa teknik segmentasi citra diantaranya: *thresholding*, *Connected Component Labeling* (CCL) dan segmentasi berbasis *Clusterring* seperti Iterasi, *K-means*, *fuzzy C-means* dan *SOM*. Pada penelitian ini, metode segmentasi yang digunakan adalah *Connected Component Labeling* (CCL). Metode ini sangat cocok digunakan dalam proses pemisahan karakter yang terdapat pada sebuah citra. Pada penelitian ini, proses segmentasi karakter digunakan pada saat melakukan pemisahan karakter yang terdapat pada ekspresi matematika.

D. Connected Component Labeling

Salah satu metode dari segmentasi adalah *Connected Component Labeling*. *Connected Component Labeling* adalah sebuah algoritma pengelompokan sederhana yang bertujuan untuk mengisolasi, mengukur, dan mengidentifikasi potensi daerah obyek dalam citra (Rajaraman & Chokkalingam, 2013). Metode ini akan menghasilkan sebuah citra dengan label baru yang sudah terkait satu sama lainnya antar sesama komponen. Dengan metode ini, akan dilakukan segmentasi untuk memisahkan setiap karakter yang terdapat pada citra.

Operasi pelabelan dari daerah obyek akan memberikan nama atau nomor yang unik ke semua piksel bernilai 1 yang termasuk dalam daerah tersebut. Hasil pelabelan adalah komponen individu yang dapat diekstraksi. Algoritma *Connected Component Labeling* dapat bekerja pada citra biner dengan

menggunakan metode *4-connectivity* atau *8-connectivity* (Asano & Tanaka, 2010). Proses pelabelan dilakukan dengan menggunakan sebuah penanda yang berfungsi untuk mencari titik p yang menunjukkan piksel tempat label akan diberikan pada daerah *foreground*. Ketika kondisi bernilai benar maka akan dilakukan pengecekan ke seluruh titik tetangga dari p (tergantung dari jumlah *n-connectivity*). Jika semua tetangga adalah *background* maka akan diberikan label baru pada p. Tetapi apabila hanya satu tetangga yang bernilai *foreground* maka label p sama dengan label tetangga. Dengan begitu, setiap karakter akan diberi label yang berbeda sehingga karakter yang satu dengan karakter yang lain dapat dipisahkan berdasarkan label yang dimilikinya.

Gambar 3.2 dan 3.3 akan menunjukan contoh hasil dari penerapan metode *connected component labeling* pada sebuah citra.

0	1	1	0	1	1	1	0
0	1	1	0	1	1	1	0
0	1	1	0	1	1	1	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0
0	0	0	0	1	1	1	0
0	0	0	0	1	1	1	0

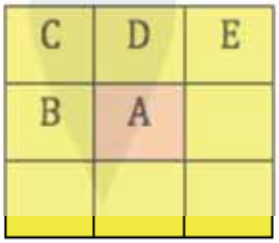
Gambar 3.2 Contoh citra biner

Dikutip dari : Rajaraman (2013). Connected Components Labeling and Extraction Based Interphase Removal from Chromosome Images

0	1	1	0	2	2	2	0
0	1	1	0	2	2	2	0
0	1	1	0	2	2	2	0
0	0	0	0	0	0	0	0
0	0	0	0	3	3	3	0
0	0	0	0	3	3	3	0
0	0	0	0	3	3	3	0

Gambar 3.3 Contoh citra hasil proses *connected component labeling*
Dikutip dari : Rajaraman (2013). Connected Components Labeling
and Extraction Based Interphase Removal from Chromosome Images

Salah satu metode yang dapat digunakan untuk proses *connected component labeling* adalah *8-Connectivity Sequential Connected Component Labeling*. Hasil dari proses ini berupa matrix label yang berasal dari citra biner. Algoritma *8-connectivity sequential connected component labeling* merupakan jenis algoritma *two-pass* dari *connected component labeling*. Algoritma ini akan menelusuri citra sebanyak dua kali *(two-pass)*. Berikut merupakan alur dari metode *connected component labeling*:



Gambar 3.4 *8-connectivity neighbourhood*

1. Telusuri citra dari kiri ke kanan, atas ke bawah
2. Fase *scanning*: (*pass* pertama), Jika piksel (A) adalah *foreground* maka cek piksel tetangganya.
    a. Jika B adalah *foreground* maka labeli A sama seperti label B
    b. Selain itu jika C adalah *foreground* maka labeli A sama seperti label C
    c. Selain itu jika D adalah *foreground* maka labeli A sama seperti label D
    d. Jika E adalah *foreground* maka:
        1) Jika A belum berlabel maka labeli A sama seperti label E
        2) Jika A sudah berlabel maka catat hal ini dalam *Equivalent Map*
    e. Jika B,C,D dan E adalah *background* maka labeli A dengan label baru dan tambahkan jumlah objek yang ditemukan
3. Tahap analisis: Dari *Equivalent Map* disiapkan *remapping array* untuk proses perbaikan pada matrix label.
4. Fase *Labeling*: (*pass* kedua), Perbaiki label pada matrix menggunakan *remapping array*.

Matrix label kemudian akan ditelusuri untuk mendapatkan daftar dari obyek-obyek yang ada pada gambar beserta informasi lokasi dan ukuran dari setiap label.

E. Ekstraksi Fitur

Choudhary & Rishi (2011) dan Kader & Deb (2012) berpendapat bahwa proses ekstraksi fitur merupakan proses menandai dan menyimpan semua fitur dari hasil segmentasi karakter. Fitur adalah suatu bentuk informasi atau karakteristik dari segmen yang bisa dijadikan sebagai tanda pengenal dari bentuk segmen karakter tersebut, oleh karena itu fitur harus unik untuk tiap bentuk karakter.

Ekstraksi fitur akan mengubah huruf / angka yang terdapat dalam citra menjadi susunan kode angka (yaitu angka 0 dan 1). Ekstraksi fitur berfungsi untuk mengubah suatu pola menjadi bit-bit digital sehingga dapat dimengerti oleh komputer.

F. Learning Vector Quantization

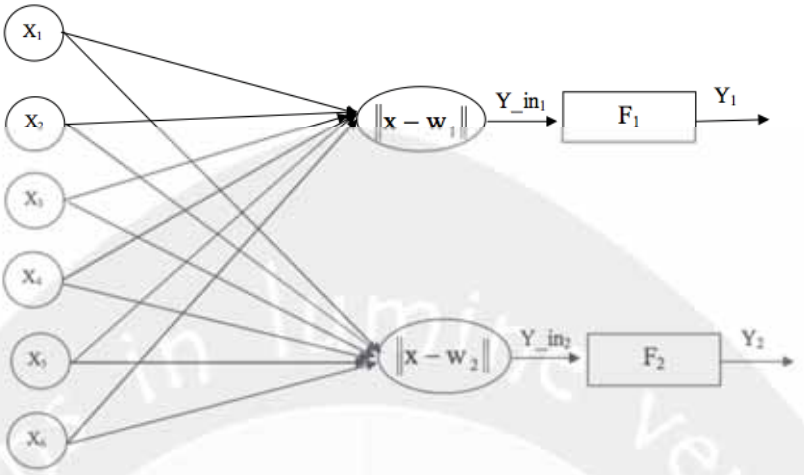
Menurut Soleiman & Fetanat (2014), *Learning Vector Quantization* (LVQ) adalah suatu metode yang melakukan pembelajaran pada lapisan kompetitif yang terawasi. Lapisan kompetitif akan secara otomatis belajar untuk mengklasifikasikan vektor-vektor input. Kelas-kelas yang diperoleh sebagai hasil dari lapisan kompetitif ini hanya tergantung pada jarak antara vektor-vektor input. Apabila beberapa vektor input memiliki jarak yang sangat berdekatan, maka vektor-vektor input tersebut akan dikelompokan dalam kelas yang sama. Metode yang digunakan untuk menghitung jarak vektor pada jaringan LVQ adalah *Euclidian Distance*, dengan demikian waktu yang diperlukan untuk melakukan proses pengenalan pola relatif cepat. Hal ini sangat cocok diterapkan pada aplikasi

berbasis *mobile* yang membutuhkan operasi perhitungan yang sangat cepat, sehingga aplikasi yang dibangun dapat berjalan dengan optimal.

1. Arsitektur Jaringan LVQ

Metode LVQ dapat digunakan untuk proses pengelompokan dimana jumlah kelompok telah ditentukan sesuai dengan rancangan arsitekturnya (target/kelas sudah ditentukan). Menurut Chen, Tsai & Yang (2010), LVQ merupakan salah satu metode pembelajaran yang handal dalam melakukan pengklasifikasian. Soleiman & Fetanat (2014) berpendapat bahwa pada jaringan LVQ, lapisan kompetitif akan belajar untuk mengklasifikasikan vektor input terhadap lapisan input yang terdapat pada jaringan LVQ.

Umer & Khiyal (2007) berpendapat bahwa arsitektur jaringan LVQ dapat dikelompokan atas tiga bagian yaitu: *input layer, competitive layer (hidden layer)*, dan *output layer*. Pada *input layer* berisi vektor-vektor input yang mendeskripsikan fitur / ciri dari sekumpulan pola yang akan dikenali. Pada *competitive layer*, setiap lapisan unit akan melakukan *clustering* / pengelompokan terhadap vektor input. Hasil *clustering* diperoleh dari perhitungan jarak (*euclidian disctance*) antara vektor input dengan lapisan unit. Vektor-vektor input yang memiliki jarak *euclidian* yang sangat berdekatan dengan salah satu lapisan unit, maka vektor input dan lapisan unit tersebut akan dikelompokan ke dalam kelas (*class*) yang sama. Sedangkan pada *output layer* berisi jumlah target yang merepresentasikan jumlah kelas yang terdapat pada jaringan LVQ.



Gambar 3.5 Contoh Arsitektur Jaringan LVQ

Dikutip dari Kusumadewi (2004). Jaringan Syaraf Tiruan. Hal 296

dimana:

X = Vektor Masukan (x_1,…,x_2,…,x_n)

F = Lapisan kompetitif

Y_in = Masukan kelapisan kompetitif

Y = Keluaran (*Output*)

W = Vektor bobot untuk unit keluaran

||x - w|| = Selisih nilai jarak *Euclidean* antara vektor *input* dengan vektor bobot untuk unit *Output*

Gambar 3.5 adalah arsitektur jaringan LVQ (Learning *Vector Quantization)* yang terdiri dari 6 *node* lapisan masukan (Input *layer),* 2 *node* lapisan tersembunyi (Hidden *layer)* serta 2 *node* lapisan keluaran (Output *layer).* Lapisan *input* memiliki 6 *node* yang disimbolkan dengan nilai x_1, x_2, x_3, x_4, x_5,

dan x_6. Pada tiap lapis masukan terlebih dahulu diberikan dua nilai bobot yang berbeda yaitu w_1 dan w_2. Dimana tujuan dari pemberian bobot tersebut adalah agar nilai tiap masukan memiliki penimbang (bobot) yang kemudian akan dihitung nilainya dan diselesaikan dengan persamaan yang dimiliki oleh metode LVQ (*LearningVector Quantization).* Sehingga jaringan memiliki dua buah kelas yang berbeda, yaitu kelas 1 dan kelas 2. Kelas 1 memiliki bobot w_1 dan kelas dua memiliki bobot w_2. Keluaran dari lapisan ini akan menjadi masukan bagi lapisan tersembunyi (hidden *layer)* sebanyak 2 *node* yaitu Y_in1 dan Y_in2. Dengan menggunakan prinsip bahwa nilai paling kecil yang dihasilkan adalah pemenang dan merupakan kelas dari input tersebut maka pada lapisan keluaran (Output *layer)* digunakan sebuah fungsi pembanding yang berguna membandingkan dua nilai tersebut untuk dicari nilai terkecilnya. Dalam jaringan diatas fungsi pembanding tersebut dituliskan dengan simbol F_1 dan F_2.

2. Algoritma Jaringan LVQ

Dalam buku Kusumadewi (2004) dijelaskan algoritma dari metode LVQ (*Learning Vector Quantization*). Metode ini terdiri dari 2 algoritma yaitu algoritma pelatihan dan algoritma pengujian. Berikut merupakan alur dari setiap algoritma:

a. Algoritma Pelatihan:

1) Tetapkan:

a) Bobot awal variable input ke-j menuju ke kelas (*cluster*) ke-i: Wij, dengan i = 1,2,…, K; dan j = 1,2,…, m

b) Maksimum *epoh:*MaxEpoh

c) Parameter *learning rate:*α

d) Penguranan *learning rate:* Decα

e) Minimal *learning rate* yang diperbolehkan: Minα

2) Masukan:

a) Data input: X_{ij}, dengan i= 1,2,…n; dan j = 1,2,…,m

b) Target berupa kelas: T_k; dengan k= 1,2,…,n.

3) Tetapkan kondisi awal: *eoph* = 0

4) Kerjakan jika: (*epoh* ≤ *MaxEpoh*) dan ($\alpha \geq Min\alpha$)

a) *Epoh = epoh +1;*

b) Kerjakan untuk i = 1 sampai n

(1) Tentukan J sedemikian hingga $|X_i - W_j|$ minimum; dengan j = 1,2,…,K

(2) Perbaiki W dengan ketentuan:

(a) Jika T = C_j, maka:

$W_j = W_j + \alpha (X_i - W_j)$

(b) Jika T ≠ C_j, maka:

$W_j = W_j - \alpha (X_i - W_j)$

c) Kurangi nilai α.

(Pengurangan α bias dilakukan dengan: $\alpha = \alpha - Dec\alpha$; atau dengan cara: $\alpha = \alpha - \alpha * Dec\alpha$)

Setelah dilakukan pelatihan, akan diperoleh bobot-bobot akhir (W). Bobot-

bobot ini nantinya akan digunakan untuk melakukan simulasi atau pengujian.

b. Algoritma Pengujian

1) Masukan data yang akan diuji, missal: X_{ij}; dengan i = 1,2,…,np; dan j = 1,2,…,m
2) Kerjakan untuk i = 1 sampai np
   a) Tentukan J sedemikian hingga $Q\%X_i - W_jQ\%$ minimum; dengan j = 1,2,…,K
   b) J adalah kelas untuk X_i

G. *Infix* dan *Postfix*

Infix adalah suatu metode penulisan notasi aritmatika dimana posisi operator berada di antara dua operand. Dalam hal ini pemakaian tanda kurung () sangat menentukan hasil operasi. Contoh penulisan *infix* adalah: **(A+B)*(C-D)**. Sedangakan *Postfix* adalah suatu metode penulisan notasi aritmatika dimana posisi operator berada setelah operand. Contoh penulisan postfix adalah: **AB+CD-***.

Pada umumnya penggunaan notasi *infix* lebih sering digunakan pada perhitungan aritmatika jika dibandingkan dengan penggunaan notasi *postfix*. Akan tetapi pada mesin kompilasi notasi *postfix* merupakan notasi yang digunakan dalam melakukan perhitungan. Hal ini dikarenakan dengan menggunakan metode *postfix* mesin kompolasi dapat dengan mudah melakukan pengkodean dan

perhitungan. Salah satu metode untuk mengubah notasi *infix* menjadi notasi *postfix* adalah dengan menggunakan operasi *Stack.*

Arianty (2008) dalam jurnalnya membahas tentang cara mengkonversi notasi *infix* menjadi notasi *postfix* menggunakan operasi *stack* pada struktur data. Suatu *stack* pada dasarnya merupakan *array* yang memuat 2 informasi penting yaitu NOEL yang berfungsi untuk mengetahui jumlah tumpukan dan TOP yang berfungsi untuk menentukan posisi puncak dari suatu *stack.*

Stack dapat diartikan sebagai tumpukan, dimana pada konsep ini memori komputer diibaratkan sebagai suatu tumpukan yang memiliki cara kerja, "yang terakhir masuk ke kotak, akan diambil yang pertama kali" atau "*last in first out*". Pada operasi *stack*, data yang ada akan diolah dengan menggunakan operator *Create*, *Isempty*, *Push*, dan *Pop*. Arianty (2008) mengatakan bahwa dalam proses konversi notasi *infix* ke *postfix*, terdapat beberapa aturan yang digunakan diantaranya:

1. Jika ditemukan symbol kurung buka '(', Operasi *push* pada *stack* akan digunakan untuk menyimpan symbol tersebut ke dalam *stack.*
2. Jika ditemukan symbol kurung tutup ')', Operasi *pop* digunakan untuk mengeluarkan operator-operator yang berada di dalam *stack*.
3. Jika terdapat symbol operator pada notasi *infix* maka operasi yang dilakukan pada *stack* adalah sebagai berikut:
    a. Jika TOP(S) dari *stack* tersebut kosong atau berisi simbol '(' maka operasi *push* akan digunakan untuk memasukan operator tersebut pada posisi di TOP(S)

b. Jika operator yang berada di puncak *stack* merupakan elemen yang memiliki tingkat yang sama atau lebih tinggi maka operasi *pop* digunakan utuk mengeluarkan operator tersebut diikuti operasi *push* untuk menyimpan operator hasil *scanning* untai.

c. Jika operator yang berada di puncak *stack* memiliki tingkat yang lebih rendah dari operator yang di-*scan*, maka operator baru akan langsung dimasukan ke dalam *stack* dengan operasi *push*.

4. Jika ditemukan suatu operand, nilai operand yang ada langsung dijadikan *output* dari notasi *postfix*.

Berikut merupakan tingkatan operator yang disusun berdasarkan urutan level tertinggi dapat dilihat pada Tabel 3.1

Tabel 3.1

Level operator dalam *Stack*

Operator	Tingkatan Operator dalam *stack*
** atau pangkat	Tertinggi
* atau /	Menengah
+ atau -	Rendah

Arianty (2008) dalam jurnalnya menjelaskan bahwa algoritma untuk mengubah notasi *infix* menjadi notasi *postix* adalah sebagai berikut:

1. **Read** Panjang suatu untai karakter
2. **Create** Stack
3. n := 0

4. **For K := 1 To** Panjang untai + 1 **do**

    If untai ke k adalah operand **then**

        Keluarkan untai ke K dalam bentuk *Output*

    Else

        While nilai operator dalam *stack* ≥ operator yang dibaca **do**

            Pop isi operator dari dalam *stack*

            Keluarkan operator tersebut dalam bentuk *Output*

        End While

        IF operator ke K = ‘)’ **then**

            Pop isi operator dari *stack*

        Else

            Pushoperator ke dalam *stack*

        End If

    End If

    Panjang untai ;= n

End For

Berikut merupakan contoh perubahan dari notasi *infix* ke *postfix*. Notasi *infix* yang akan diubah adalah: **((A * B) + C / D – E ** F) * G;** dimana hasil konversi ke dalam bentuk notasi *postfix*-nya digambarkan sebagai berikut:

Indeks Urutan Ke-	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Karakter-karakter yang akan dilacak dalam notasi *Infix*	(	(	A	*	B	)	+	C	/	D	-	E	**	F	)	*	G	;
Operator puncak (TOP(S)	(	(	(	*	*	(	+	+	/	/	-	-	**	**		*	*	
Output yg dihasilkan dalam bentuk *postfix*			A		B	*		C		D	/+	E		F	**.		G	*

Gambar 3.6 Proses konversi dari notasi *infix* ke notasi *postfix*

Dikutip dar : Arianti (2008). Social Network.

(http://isjd.pdii.lipi.go.id/admin/jurnal/13308207214.pdf)

Berdasarkan gambar di atas, dapat dijelaskan sebagai berikut:

1. *Input* yang dibaca adalah ‘(’, maka dimasukan ke dalam *stack*.
2. *Input* yang dibaca adalah ‘(’, maka dimasukan ke dalam *stack.*
3. *Input* yang dibaca adalah operand ‘A’, maka langsung dimasukan ke dalam Output
4. *Input* yang dibaca adalah operator ‘*’, karena posisi teratas pada *stack* adalah ‘(’, maka ‘*’ dimasukan ke dalam *stack*.
5. *Input* yang dibaca adalah operand ‘B’, maka langsung dimasukan ke dalam Output
6. *Input* yang dibaca adalah ‘)’, maka keluarkan semua operator yang ada di *stack*s ampai menemukan simbol ‘)’, kemudian masukan ke dalam Output

(simbol ‘(’ dan ‘)’ tidak perlu dimasukan ke dalam Output). Sehingga operator * yang ada di *stack* dipindahkan ke Output.

7. *Input* yang dibaca adalah operand ‘+’, karena posisi teratas pada *stack* adalah ‘(’ maka ‘+’ dimasukan ke dalam *stack*.
8. *Input* yang dibaca adalah operand ‘C’, maka langsung dimasukan ke dalam Output
9. *Input* yang dibaca adalah operator ‘/’, karena posisi teratas pada *stack* adalah ‘+’ dan tingkat operator ‘/’ lebih tinggi dari ‘+’, maka ‘/’ dimasukan ke dalam *stack*.
10. *Input* yang dibaca adalah operand ‘D’, maka langsung dimasukan ke dalam Output
11. *Input* yang dibaca adalah operator ‘-’, karena posisi teratas pada *stack* adalah ‘/’ dan tingkat operator ‘-’ lebih rendah dari ‘/’, maka ‘/’ dipindahkan ke dalam Output. Kemudian cek lagi di *stack*,karena posisi teratas pada *stack* adalah ‘+’ dan tingkat operator ‘-’ sama dengan ‘+’, maka ‘+’ dipindahkan ke dalam Output. Karena posisi teratas pada *stack* adalah ‘(’, maka ‘-’ dimasukan ke dalam *stack*.
12. *Input* yang dibaca adalah operand ‘E’, maka langsung dimasukan ke dalam Output
13. *Input* yang dibaca adalah operator ‘**’, karena posisi teratas pada *stack* adalah ‘-’ dan tingkat operator ‘**’ lebih tinggi dari ‘-’, maka ‘**’ dimasukan ke dalam *stack*.

14. *Input* yang dibaca adalah operand ‘F’, maka langsung dimasukan ke dalam Output
15. *Input* yang dibaca adalah ‘)’, maka keluarkan semua operator yang ada di *stack*sampai menemukan simbol ‘)’, kemudian masukan ke dalam Output. Sehingga operator ‘**’ dan ‘-’ yang ada di *stack* dipindahkan ke Output.
16. *Input* yang dibaca adalah operator ‘*’, karena *stack* masih kosong, maka ‘*’ dimasukan ke dalam *stack*.
17. *Input* yang dibaca adalah operand ‘G’, maka langsung dimasukan ke dalam Output
18. *Input* yang dibaca adalah ‘;’, maka semua operator yang ada di *stack* dipindahkan ke Output. Sehingga operator ‘*’ yang ada di *stack* dipindahkan ke Output.

Dengan demikian, hasil konversi dari notasi *infix* berupa **((A * B) + C / D – E ** F) * G** ke notasi *postfix* adalah **A B * C D / + E F ** - G ***