

MODUL AJAR TEORI GRAF DAN OTOMATA

DAFTAR ISI

DAFTAR ISI.....	1
BAB 1. DASAR-DASAR GRAF	4
1.1. Pengertian Graf	4
1.2. Graf Sederhana (<i>Simple Graph</i>).....	4
1.3. Graf dan Sub-graf.....	5
1.4. Graf Tak-berarah dan Graf Berarah.....	6
1.5. Representasi Graf	6
1.6. Isomorfisme (<i>isomorphism</i>)	8
1.7. Problem: Lintasan terpendek	8
1. Algoritma Dijkstra.....	8
2. Algoritma Bellman-Ford	9
1.8. Latihan Soal	11
BAB 2. POHON (TREE).....	13
2.1. Pengertian Pohon.....	13
2.2. Pohon Merentang (<i>Spanning Tree</i>)	14
2.3. Formula Cayley's	15
2.4. Problem: Pohon Merentang Minimum	16
1. Algoritma Prim's.....	16
2. Algoritma Kruskal's	17
2.5. Latihan Soal	19
BAB 3. LINTASAN DAN SIRKUIT	20
3.1. Pengertian Lintasan dan Sirkuit	20
3.2. Lintasan dan Sirkuit Euler	20
3.3. Lintasan dan Sirkuit Hamilton	22
3.4. Problem.....	22
1. Permasalahan Tukang Pos Cina (Chinese Postman Problem/CPP)	22
2. Permasalahan Perjalanan Sales (Traveling Salesman Problem/TSP).....	23
3.5. Latihan Soal	25
BAB 4. PLANARITAS.....	28
4.1. Graf Planar	28
4.2. Graf Dual	29
4.3. Problem.....	29
1. Teorema Kurawtoski	30
2. Teorema Fáry's.....	31
4.4. Latihan Soal	31
BAB 5. PEWARNAAN DAN PENCOCOKAN GRAF.....	32

5.1. Pewarnaan Graf.....	32
5.2. Bilangan Kromatis (<i>chromatic number</i>).....	32
5.3. Polinomial Kromatik (<i>Chromatic Polynomial</i>).....	33
5.4. Pencocokan (<i>matching</i>)	34
5.5. Graf Bipartite	35
5.6. Problem.....	36
1. Penjadwalan (<i>Scheduling</i>).....	36
2. Permasalahan Penugasan Person (<i>Personnel Assignment Problem</i>)	37
5.7. Latihan Soal	41
BAB 6. KONEKTIVITAS	42
6.1. Konsep Keterhubungan (<i>Connectivity</i>).....	42
6.2. Komponen Terhubung.....	43
6.3. <i>Cut Edge (Bridge)</i>	43
6.4. Cut Vertex.....	43
6.5. Aliran pada Jaringan (<i>Flow in Network</i>).....	44
6.7. Aliran Maksimum (<i>Maximum Flow</i>) pada Jaringan.....	45
6.8. Cut Set	46
6.9. Teorema Max-Flow Min-Cut.....	47
6.10. Problem.....	48
Maximum Bipartite Matching sebagai Permasalahan Aliran Maksimum.....	48
6.11. Latihan Soal	49
BAB 7. TEORI BAHASA & OPERASI MATEMATIS PENDUKUNGNYA	51
7.1. Pentingnya Belajar Teori Komputasi	51
7.2. Terminologi Bahasa & Operasi Matematis yang Mendukungnya	52
7.3. Mesin-mesin yang Berasosiasi dengan Klas Bahasa	55
7.4. Latihan Soal	60
BAB 8. REGULAR LANGUAGES	61
8.1. Regular Expression.....	61
8.2. Finite Automata.....	64
8.3. Regular Language & Nonregular Language	67
8.4. Finite Automata with Output	72
8.5. Latihan Soal	76
BAB 9. NONDETERMINISM.....	77
9.1. Determinism vs Nondeterminism.....	77
9.2. Nondeterministic Finite Automata (NFA).....	77
9.3. Konversi NFA menjadi DFA	79
9.4. Konversi RE menjadi NFA.....	82
9.5. Latihan Soal	83
BAB 10. CONTEXT-FREE LANGUAGE	85

10.1. Derivasi dan Parsing	86
10.2. Klasifikasi Grammar.....	88
10.3. Penyederhanaan Context-Free Language.....	91
10.4. Chomsky Normal Form dan Greibach Normal Form.....	93
10.5. Latihan Soal	95
BAB 11. PUSHDOWN AUTOMATA DAN MESIN TURING	98
11.1. Pushdown Automata (PDA).....	98
11.2. Membentuk PDA dari CFG	103
11.3. Mesin Turing.....	104
11.4. Latihan Soal	105

BAB 1. DASAR-DASAR GRAF

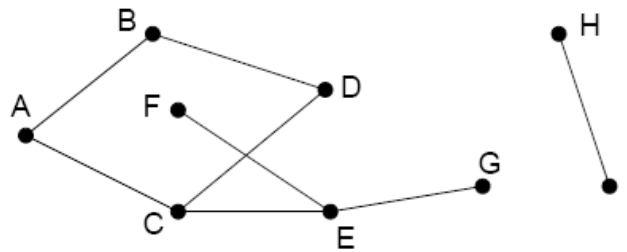
1.1. Pengertian Graf

Secara sederhana graf didefinisikan sebagai kumpulan titik yang dihubungkan oleh garis. Secara matematis, graf adalah pasangan himpunan (V, E) dimana V adalah himpunan tak kosong yang memiliki elemen disebut simpul (*vertices*) dan E adalah kumpulan dari dua elemen subsets V yang disebut busur (*edges*).

Simpul direpresentasikan dengan titik dan busur direpresentasikan dengan garis. Gambar 1.1 adalah contoh graph (V, E) dimana:

$V = \{A, B, C, D, E, F, G, H, I\}$, dan

$E = \{\{A, B\}, \{A, C\}, \{B, D\}, \{C, D\}, \{C, E\}, \{E, F\}, \{E, G\}, \{H, I\}\}$.



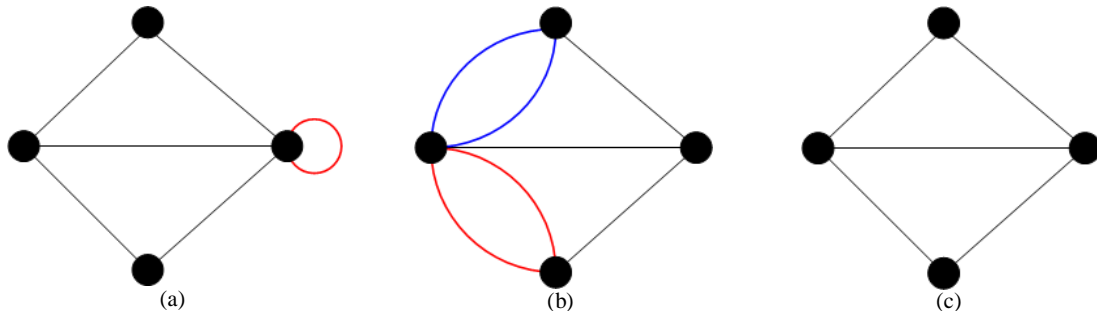
Gambar 1.1. Contoh graf

Sebuah busur selalu memiliki dua *endpoint*, misalnya busur $\{H, I\}$ memiliki *endpoint* H dan I . Graf biasanya digunakan untuk memodelkan objek-objek diskrit dan hubungan antar objek-objek tersebut.

1.2. Graf Sederhana (*Simple Graph*)

Graf sederhana adalah graf yang tidak mengandung *loops* atau *multiple edges*. *Loops* adalah busur yang memiliki *endpoint* sama, sedangkan *multiple edges* adalah busur yang memiliki pasangan *endpoint* sama.

Contoh graf sederhana dan bukan graf sederhana dapat dilihat pada Gambar 1.2. Gambar 1.2.a bukan graf sederhana karena memiliki *loop*. Gambar 1.2.b bukan graf sederhana karena memiliki *multiple edges*. Gambar 1.2.c merupakan graf sederhana karena tidak memiliki *loops* atau *multiple edges*.

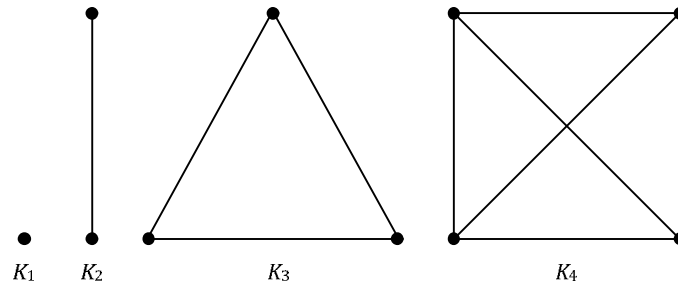


Gambar 1.2. (a) Bukan graf sederhana karena memiliki *loop*. (b). Bukan graf sederhana karena memiliki *multiple edges*. (c) Graf sederhana.

Beberapa jenis graf khusus yang termasuk graf sederhana diantaranya:

1. Graf Komplit (*Complete Graph K_n*)

Adalah graf dimana setiap pasang *vertices* selalu memiliki sebuah busur. Graf komplit dapat diamati pada Gambar 1.3. Simbol dari graf komplit adalah K_n , dimana n menyatakan jumlah simpul dari graf komplit.



Gambar 1.3. Graf komplit K_n untuk $n=1,2,3,4$.

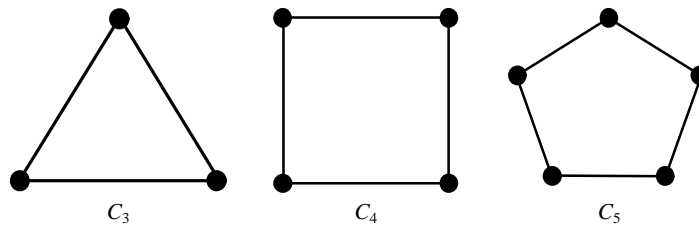
2. Cycle (C_n)

Graf *cycle* adalah graf $C = (V, E)$ dengan bentuk

$$V = \{v_1, v_2, v_3, v_4, \dots, v_n\} \text{ dan}$$

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \dots, \{v_n, v_1\}\},$$

dimana $n \geq 3$ dan $v_1, v_2, v_3, v_4, \dots, v_n$ adalah simpul yang berbeda. Graf *cycle* disimbolkan dengan C_n dimana n adalah banyaknya simpul. Gambar 1.4 menunjukkan graf *cycle* C_n untuk $n=3, 4$, dan 5 .



Gambar 1.4. Graf *cycle* C_n untuk $n=3,4,5$

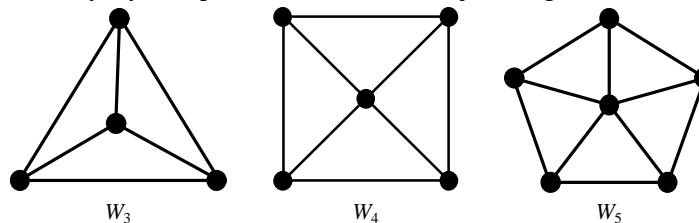
3. Wheel (W_n)

Graf *wheel* adalah graf *cycle* yang ditambahi sebuah simpul baru (v_m) dimana v_m terhubung ke seluruh simpul yang ada. Sehingga graf $W = (V, E)$ dengan bentuk

$$V = \{v_1, v_2, v_3, v_4, \dots, v_n, v_m\} \text{ dan}$$

$$E = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \dots, \{v_n, v_1\}, \{v_1, v_m\}, \{v_2, v_m\}, \{v_3, v_m\}, \dots, \{v_n, v_m\}\},$$

dimana $n \geq 3$ dan $v_1, v_2, v_3, v_4, \dots, v_n, v_m$ adalah simpul yang berbeda. Graf *wheel* disimbolkan dengan W_n dimana $n+1$ adalah banyaknya simpul. Gambar 1.5 menunjukkan graf *wheel* W_n untuk $n=3, 4$, dan 5 .

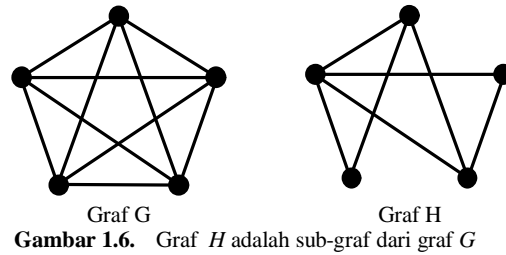


Gambar 1.5. Graf *wheel* W_n untuk $n=3,4,5$

1.3. Graf dan Sub-graf

Jika $V(G)$ dan $E(G)$ adalah himpunan simpul dan busur pada graf G , serta $V(H)$ dan $E(H)$ adalah himpunan simpul dan busur pada graf H . Maka graf H disebut sub-graf dari graf G jika dan hanya jika $V(H) \subseteq V(G)$ dan $E(H) \subseteq E(G)$.

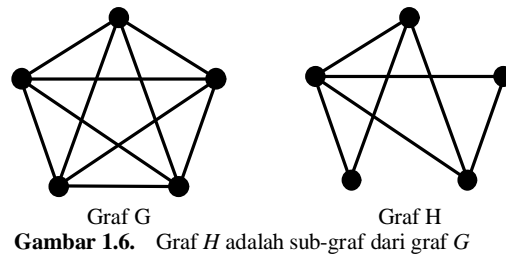
Jadi, jika e adalah busur pada graph H yang menghubungkan simpul v dan u maka e juga merupakan busur pada graph G yang menghubungkan simpul v dan u di graf G . Gambar 1.6 menunjukkan graf H yang merupakan sub-graf dari graf G .



1.4. Graf Tak-berarah dan Graf Berarah

Graph tak-berarah (*undirected graph*) adalah graf yang tidak memiliki orientasi arah pada setiap busur yang dimiliki. Penulisan busur tidak memperhatikan urutan. Penulisan busur $e = (u,v)$, dimana busur e adalah busur yang menghubungkan simpul u dan v sama saja dengan penulisan $e = (v,u)$. Semua graf yang ditampilkan pada Gambar 1.1 sampai dengan Gambar 1.6 merupakan graf tak-berarah.

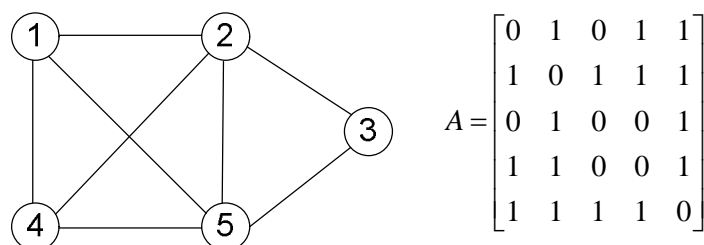
Graf berarah (*directed graph/ digraph*) adalah graf yang memiliki orientasi arah pada setiap busur yang dimiliki. Sehingga, penulisan busur $e = (u,v)$ untuk busur e yang menghubungkan simpul u dan v berbeda maknanya dengan penulisan busur $e = (v,u)$ yang menghubungkan simpul v dan u . Setiap busur pada digraph biasa juga disebut dengan *arc*.



1.5. Representasi Graf

Graf memiliki kemudahan pemahaman ketika direpresentasikan dalam bentuk visual. Akan tetapi, ketika domain permasalahan graf dibawa pada pemrograman maka graf harus dapat direpresentasikan dalam struktur data. Secara umum terdapat dua cara yang dapat merepresentasikan graf, yaitu representasi matriks dan representasi list.

1. Matriks Ketetanggaan (*Adjacency matrix*)

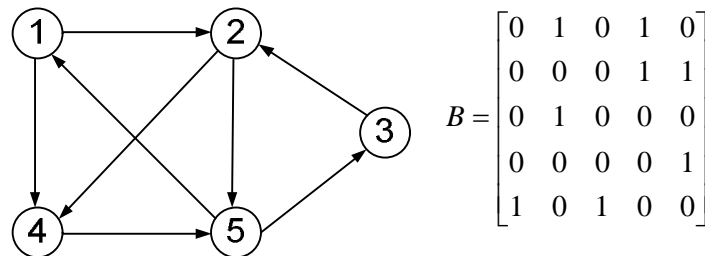


Gambar 1.7. Graf G yang memiliki *adjacency matrix* A

Matriks keketanggaan dari sebuah graf G dengan n simpul adalah berupa matriks $A = \{a_{ij}\}$ dengan ukuran $n \times n$. a_{ij} bernilai 1 apabila terdapat sebuah busur yang menghubungkan simpul i dan simpul j , serta a_{ij} bernilai 0 apabila tidak terdapat sebuah busur yang menghubungkan simpul i dan simpul j .

A adalah matriks ketetanggaan dari graf G pada Gambar 1.7. Graf G memiliki 5 simpul sehingga A adalah matriks yang berukuran 5×5 . Elemen $a_{1,2}$ bernilai 1 karena terdapat sebuah busur yang menghubungkan simpul 1 dan 2. Sedangkan elemen $a_{1,3}$ bernilai 0 karena tidak terdapat sebuah busur yang menghubungkan simpul 1 dan 3.

B adalah matriks ketetanggaan graf H pada Gambar 1.8. Jika dibandingkan antara matriks A dan matriks B , dapat dilihat bahwa matriks ketetanggaan dari graf tak berarah (A) bersifat simetris dan graf berarah (B) belum tentu memiliki matriks ketetanggaan yang simetris.

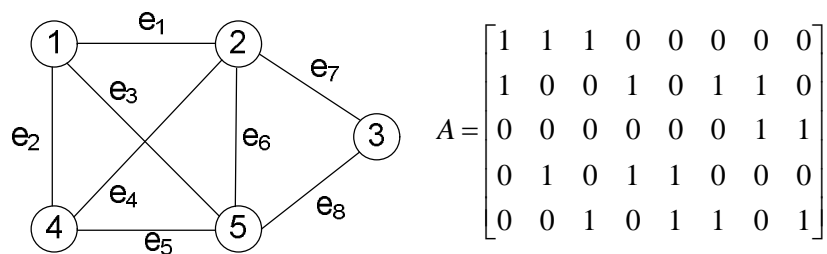


Gambar 1.8. Graf H yang memiliki adjacency matrix B

2. Matriks Insiden (Incidence matrix)

Matriks insiden dari sebuah graf G dengan m simpul dan n busur adalah berupa matriks $A = \{a_{ij}\}$ dengan ukuran $m \times n$. a_{ij} bernilai 1 apabila simpul i merupakan salah satu *end point* dari busur j dan sebaliknya, a_{ij} bernilai 0 apabila simpul i bukan merupakan *end point* dari busur j .

A adalah matriks insiden dari graf G pada Gambar 1.9. Graf G memiliki 5 simpul dan 8 busur, sehingga A adalah matriks yang berukuran 5×8 . Elemen $a_{1,1}$ dan $a_{2,1}$ bernilai 1 karena simpul 1 dan simpul 2 adalah *end point* busur e_1 . Matriks insiden hanya bisa untuk merepresentasikan graf tak berarah, sedangkan graf berarah tidak bisa menggunakan matriks insiden karena definisi *end point* hanya terdapat pada graf tak berarah.



Gambar 1.9. Graf G yang memiliki adjacency matrix A

A adalah matriks insiden dari graf G pada Gambar 1.9. Graf G memiliki 5 simpul dan 8 busur, sehingga A adalah matriks yang berukuran 5×8 . Elemen $a_{1,1}$ dan $a_{2,1}$ bernilai 1 karena simpul 1 dan simpul 2 adalah *end point* busur e_1 . Matriks insiden hanya bisa untuk merepresentasikan graf tak berarah, sedangkan graf berarah tidak bisa menggunakan matriks insiden karena definisi *end point* hanya terdapat pada graf tak berarah.

3. Daftar Ketetanggaan (Adjacency list)

Selain dengan matriks, graf dapat direpresentasikan dengan daftar ketetanggaan. Setiap simpul yang bertetangga ditulis dalam daftar. Daftar ketetanggaan matriks tak berarah G pada Gambar 1.7 dapat

dilihat pada Gambar 1.10(a), sedangkan daftar ketetanggaan matriks berarah G pada Gambar 1.8 dapat dilihat pada Gambar 1.10(b).

1: 2, 4, 5	1: 2, 4
2: 1, 3, 4, 5	2: 4, 5
3: 2, 5	3: 2
4: 1, 2, 5	4: 5
5: 1, 2, 3, 4	5: 1, 3

Gambar 1.10. Daftar Ketetanggaan (a) graf G pada Gambar 1.8. (b) graf G pada Gambar 1.9.

1.6. Isomorfisme (*isomorphism*)

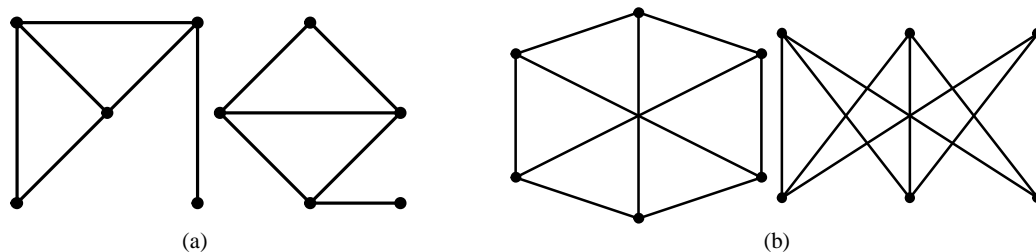
Diberikan dua graf $G_1(V_1, E_1)$ dan $G_2(V_2, E_2)$. Sebuah fungsi $f: v_1 \rightarrow v_2$ dinyatakan isomorfis jika:

1. f adalah fungsi korespondensi satu-satu
2. untuk semua $a, b \in V_1$, $\{a, b\} \in E_1$ jika dan hanya jika $\{f(a), f(b)\} \in E_2$.

Graf G_1 dan G_2 yang isomorfis dinotasikan dengan $G_1 \cong G_2$. Kondisi perlu dari dua graf yang isomorfis adalah:

1. Keduanya harus memiliki jumlah simpul yang sama
2. Keduanya harus memiliki jumlah busur yang sama
3. Keduanya harus memiliki sejumlah simpul dengan derajat yang sama
4. Simpul pada keduanya harus memiliki urutan derajat yang sama dan vektor sirkuit yang sama (c_1, \dots, c_n), dimana c_i adalah jumlah sirkuit dengan panjang i .

Contoh pasangan graf yang isomorfis dapat dimati pada Gambar 1.11(a) dan (b).



Gambar 1.11. Pasangan graf isomorfis.

1.7. Problem: Lintasan terpendek

1. Algoritma Dijkstra

Ditemukan oleh seorang ilmuwan berkebangsaan Belanda Edsger Dijkstra pada tahun 1956 dan diperkenalkan ke khalayak ramai pada tahun 1959. Algoritma ini termasuk dalam jenis algoritma *greedy* dan dipakai untuk menemukan jarak terpendek (shortest path) dalam sebuah graf dengan bobot sisi bernilai tidak negatif.

Algoritma Dijkstra merupakan salah satu varian bentuk algoritma populer dalam pemecahan persoalan yang terkait dengan masalah optimasi. Sifatnya sederhana dan lempang (*straightforward*). Sesuai dengan kata *greedy* yang secara harfiah berarti rakus, algoritma *greedy* ini hanya memikirkan solusi terbaik yang akan diambil pada setiap langkah tanpa memikirkan konsekuensi ke depan. Prinsipnya, ambillah apa yang bisa Anda dapatkan saat ini (*take what you can get now!*), dan keputusan yang telah diambil pada setiap langkah tidak akan bisa diubah kembali.

Algoritma Dijkstra memiliki dua himpunan simpul S dan C . Pada setiap tahap himpunan S berisi simpul yang telah dipilih dan C berisi semua simpul lainnya, dimana $V = S \cup C$. Ketika algoritma

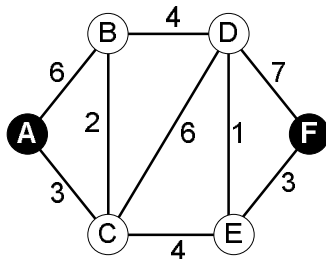
dimulai S hanya berisi simpul sumber dan ketika menghentikan algoritma, S berisi semua simpul graf dan masalah terpecahkan. Pada setiap algoritma langkah memilih simpul di C yang memiliki jarak terpendek ke sumber dan menambahkannya ke S .

DIJKSTRA (G, w, s)

1. INITIALIZE SINGLE-SOURCE (G, s)
2. $S \leftarrow \{ \}$ // S will ultimately contains vertices of final shortest-path weights from s
3. Initialize priority queue Q i.e., $Q \leftarrow V[G]$
4. while priority queue Q is not empty do
5. $u \leftarrow \text{EXTRACT_MIN}(Q)$ // Pull out new vertex
6. $S \leftarrow S \cup \{u\}$
 // Perform relaxation for each vertex v adjacent to u
7. for each vertex v in $\text{Adj}[u]$ do
8. Relax (u, v, w)

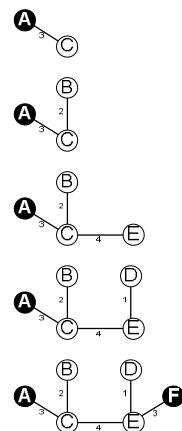
Contoh:

Kerjakan graf dibawah ini dengan algoritma Dijkstra dalam kasus mencari biaya minimum dari simpul A ke F . Bobot pada busur menyatakan biaya untuk menghubungkan *endpoint* pada masing-masing busur.



Jawab:

	B	C	D	E	F	
A	6 A	3 A	∞ -	∞ -	∞ -	$S = \{A\}$ Pilih C (terpendek)
C	5 C	3 A	9 C	7 C	∞ -	$S = \{A, C\}$ Pilih B (terpendek)
B	5 C	3 A	9 C	7 C	∞ -	$S = \{A, C, B\}$ Pilih E (terpendek)
E	5 C	3 A	8 E	7 C	10 E	$S = \{A, C, B, E\}$ Pilih D (terpendek)
D	5 C	3 A	8 E	7 C	10 E	$S = \{A, C, B, E, D\}$ Pilih F (terpendek)
F	5 C	3 A	8 E	7 C	10 E	$S = \{A, C, B, E, D, F\}$ Selesai



Karena F yang merupakan simpul tujuan telah terpilih, maka algoritma ini berhenti. Jadi, jalur terpendek dari simpul A ke F adalah 10 dengan lintasan $A - C - E - F$.

2. Algoritma Bellman-Ford

Algoritma Bellman-Ford merupakan modifikasi Algoritma Dijkstra untuk menghitung lintasan terpendek pada graf yang mengandung bobot negatif.

Algoritma Bellman-Ford dimulai dengan memberikan bobot tak terhingga (∞) kepada semua simpul kecuali simpul awal (0). Lakukan pengulangan sebanyak jumlah simpul kali jumlah busur. Jadikan setiap nilai baru yang lebih rendah dibanding nilai sebelumnya sebagai bobot simpul.

```
procedure BellmanFord(list vertices, list edges, vertex source)
  // This implementation takes in a graph, represented as lists of vertices
  // and edges, and modifies the vertices so that their distance and
  // predecessor attributes store the shortest paths.

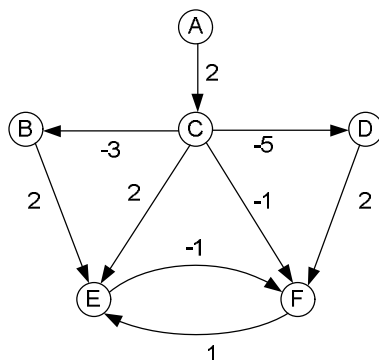
  // Step 1: initialize graph
  for each vertex v in vertices:
    if v is source then v.distance := 0
    else v.distance := infinity
    v.predecessor := null

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
    for each edge uv in edges: // uv is the edge from u to v
      u := uv.source
      v := uv.destination
      if u.distance + uv.weight < v.distance:
        v.distance := u.distance + uv.weight
        v.predecessor := u

  // Step 3: check for negative-weight cycles
  for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if u.distance + uv.weight < v.distance:
      error "Graph contains a negative-weight cycle"
```

Contoh:

Kerjakan graf dibawah ini dengan algoritma Bellman-Ford dalam kasus mencari biaya minimum dari simpul A. Bobot pada busur menyatakan biaya untuk menghubungkan *endpoint* pada masing-masing busur.



Jawab:

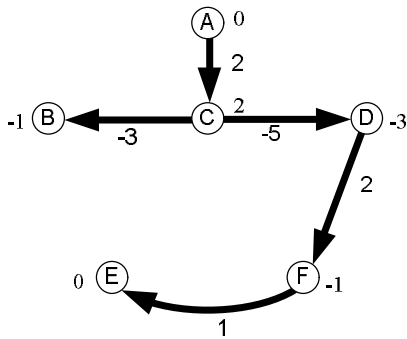
Dimisalkan, prioritas urutan pengecekan busur adalah:

AC, BE, CB, CD, CE, CF, DF, EF, FE.

Maka nilai optimum yang disimpan pada tiap vertex adalah sbb:

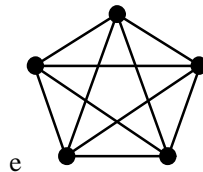
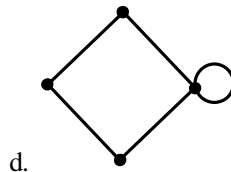
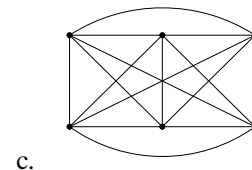
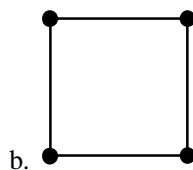
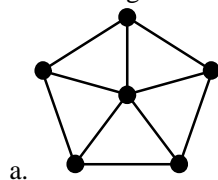
i	A	B	C	D	E	F												
1	<table><tr><td>0</td></tr><tr><td>A</td></tr></table>	0	A	<table><tr><td>∞</td></tr><tr><td>-</td></tr></table>	∞	-	<table><tr><td>∞</td></tr><tr><td>-</td></tr></table>	∞	-	<table><tr><td>∞</td></tr><tr><td>-</td></tr></table>	∞	-	<table><tr><td>∞</td></tr><tr><td>-</td></tr></table>	∞	-	<table><tr><td>∞</td></tr><tr><td>-</td></tr></table>	∞	-
0																		
A																		
∞																		
-																		
∞																		
-																		
∞																		
-																		
∞																		
-																		
∞																		
-																		
2	<table><tr><td>0</td></tr><tr><td>A</td></tr></table>	0	A	<table><tr><td>-1</td></tr><tr><td>C</td></tr></table>	-1	C	<table><tr><td>2</td></tr><tr><td>A</td></tr></table>	2	A	<table><tr><td>-3</td></tr><tr><td>C</td></tr></table>	-3	C	<table><tr><td>0</td></tr><tr><td>F</td></tr></table>	0	F	<table><tr><td>-1</td></tr><tr><td>D</td></tr></table>	-1	D
0																		
A																		
-1																		
C																		
2																		
A																		
-3																		
C																		
0																		
F																		
-1																		
D																		

Bentuk akhir graf yang memiliki lintasan terpendek adalah sebagai berikut (biaya total dari simpul A ke masing-masing simpul dinyatakan dengan bobot di samping simpul):

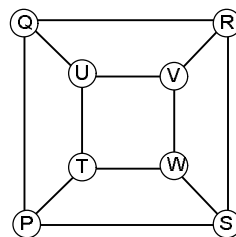
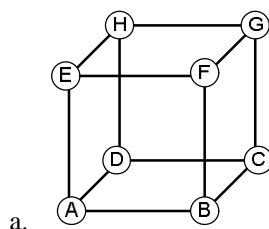


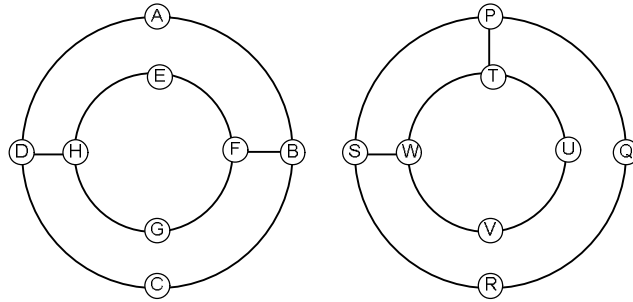
1.8. Latihan Soal

1. Termasuk graf sederhana atau bukan graf-graf di bawah ini:

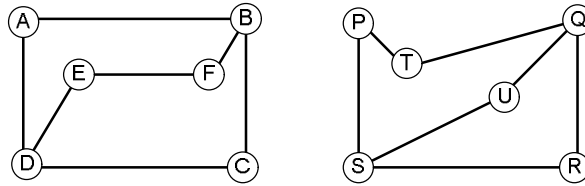


2. Tunjukkan dengan fungsi pemetaan simpul bahwa pasangan graf di bawah ini isomorfik atau tidak isomorfik:



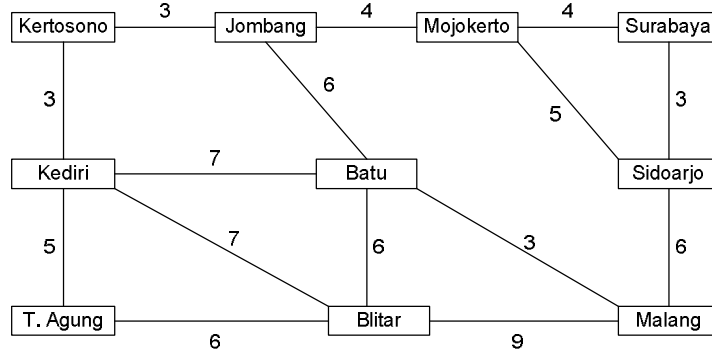


b.



c.

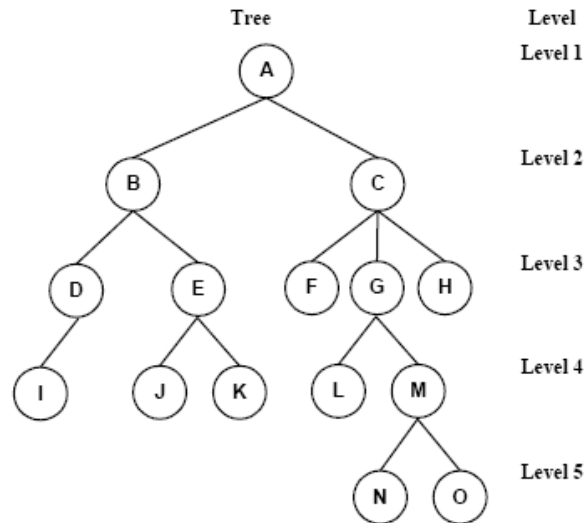
3. Andi adalah seorang pengusaha yang berdomisili di Surabaya dan memiliki beberapa kantor cabang di kota lain yaitu: Sidoarjo, Malang, Batu, Mojokerto, Jombang, Kertosono, Kediri, Blitar dan Tulung Agung . Pada suatu saat (tidak secara bersamaan), Andi harus berkunjung ke kantor-kantor cabangnya. Biaya transportasi yang dikenakan dari kota ke kota digambarkan sebagai bobot pada setiap edge (dalam ribuan rupiah) pada gambar di bawah ini. Tentukan biaya minimal dan rute dari Surabaya ke T. Agung dengan menggunakan algoritma Dijkstra!



BAB 2. POHON (TREE)

2.1. Pengertian Pohon

Pohon adalah graf tak-berarah terkoneksi yang tidak berisi sirkuit. Sebuah graf tak-berarah terkoneksi merupakan pohon jika dan hanya jika ada tepat satu lintasan sederhana antar tiap simpulnya.



Gambar 2.1. Pohon

Properti Pohon

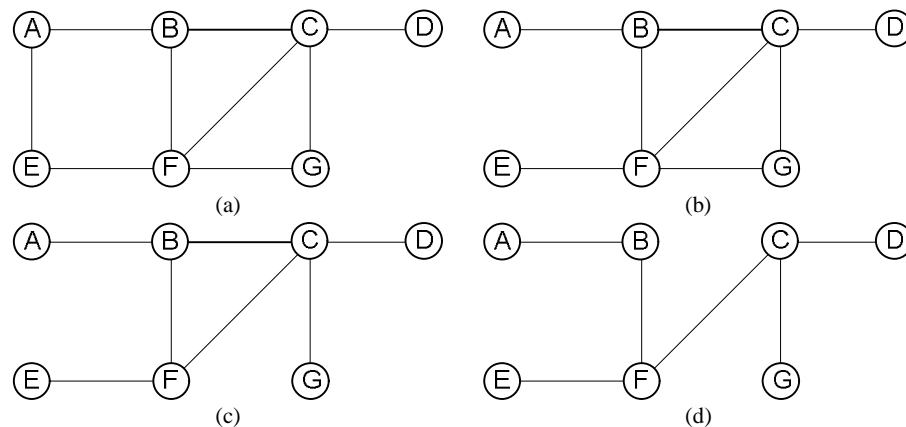
Penggambaran pohon pada ilmu komputer sedikit berbeda dengan penggambaran pohon pada dunia nyata. Pohon dalam ilmu komputer digambarkan seperti pohon pada dunia nyata yang terbalik. Akar digambarkan pada posisi paling atas, sedangkan daun tergambar di bawahnya. Untuk memahami properti pohon dalam ilmu komputer, akan digunakan Gambar 2.1 sebagai contoh.

- **Simpul (node)**, adalah elemen tree yang berisi informasi/ data dan penunjuk pencabangan. Simpul pada pohon di Gambar 2.1 berjumlah 15, yaitu: A, B, C, ..., O.
- **Akar (root)**, adalah level terendah dalam sebuah pohon. Biasanya letak akar dalam pohon adalah pada posisi teratas. Akar pada pohon di Gambar 2.1 adalah simpul A.
- **Tingkat (level)**. Tingkat suatu simpul ditentukan dari akar yang didefinisikan sebagai level 1. Apabila simpul dinyatakan sebagai tingkat N , maka simpul-simpul yang merupakan anaknya berada pada tingkat $N+1$.
- **Derajat (degree)**, menyatakan banyaknya anak/ turunan di simpul tersebut. Misalnya, simpul A memiliki derajat 2 (B dan C), simpul yang memiliki derajat 0 (nol) disebut leaf (daun) seperti : I, J, K, L, N, dan O.
- **Daun (leaf)**, adalah simpul yang memiliki derajat 0 (nol). Daun pada pohon di Gambar 2.1 antara lain: simpul I, J, K, L, N, dan O.
- **Tinggi (height)** atau **kedalaman (depth)**, adalah nilai maksimum dari tingkat suatu pohon dikurangi 1. Contoh pada pohon di Gambar 2.1 mempunyai tinggi 4.
- **Ancesor** suatu simpul adalah semua simpul yang terletak dalam satu jalur dengan simpul tersebut, dari akar sampai simpul yang ditinjaunya. Contoh, *Ancesor* simpul L adalah A, C dan G.
- **Predecessor** adalah simpul yang berada di atas simpul yang ditinjau. Contoh, *Predecessor* simpul D adalah B.
- **Successor** adalah simpul yang berada di bawah simpul yang ditinjau. Contoh, *Successor* simpul D adalah I.
- **Descendant** adalah seluruh simpul yang terletak sesudah simpul tertentu dan terletak pada jalur yang sama. Contoh, *Descendant* simpul E adalah J dan K.

- **Sibling** adalah simpul-simpul yang memiliki parent yang sama dengan simpul yang ditinjau. Contoh, *Sibling* simpul *J* adalah *K*
- **Parent** adalah simpul yang berada satu level di atas simpul yang ditinjau. Contoh, *Parent* simpul *J* adalah *E*

2.2. Pohon Merentang (*Spanning Tree*)

Misalkan G adalah graf tak-berarah sederhana, pohon merentang dari graf G adalah subgraf yang dapat dibentuk dari G berupa pohon T dimana setiap simpul di T merupakan simpul di G . Ilustrasi pembuatan pohon merentang dapat diamati pada Gambar 2.4. Pohon merentang dari sebuah graf tak-berarah sederhana G yang bukan pohon (berarti memiliki beberapa sirkuit) dapat dibentuk dengan cara memutuskan sirkuit-sirkuit yang ada. Pilih sebuah sirkuit kemudian hapus satu buah busur dari sirkuit tersebut. Lakukan secara berulang untuk setiap sirkuit yang ada di G sehingga sirkuit-sirkuit tersebut hilang dari G .

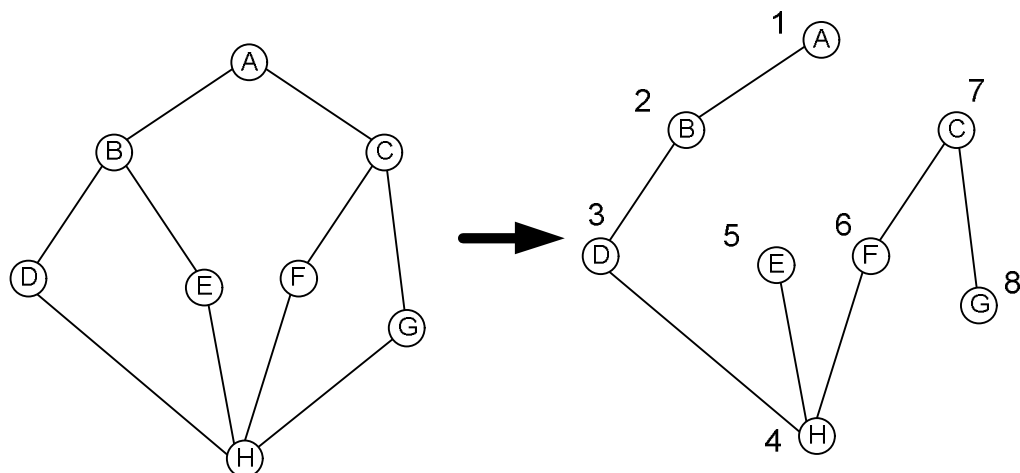


Gambar 2.4. (a)-(d) Tahap-tahap membuat pohon merentang dari graf yang bukan pohon.

Algoritma yang biasa digunakan untuk membuat pohon merentang pada sebuah graf G antara lain:

- **Depth-First Search (DFS)**

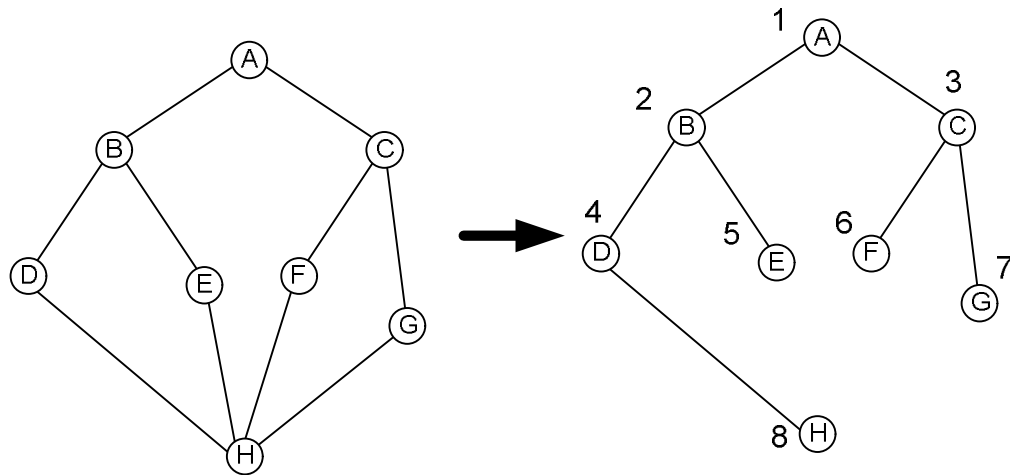
Penelusuran graf yang arah penelusurannya mendahulukan ke arah kedalaman graf tersebut. Ilustrasi algoritma DFS dapat diamati pada Gambar 2.5. Angka menunjukkan urutan pemilihan simpul berikutnya yang terpilih.



Gambar 2.5. Algoritma menentukan pohon merentang dengan DFS

- **Breadth-First Search (BFS)**

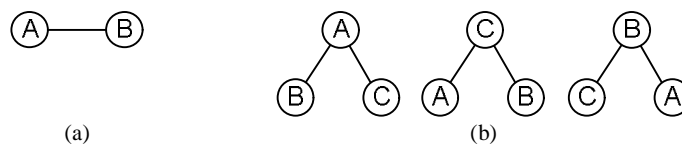
Penelusuran graf yang arah penelusurannya mendahulukan ke arah lebar graf tersebut. Ilustrasi algoritma BFS dapat diamati pada Gambar 2.6. Angka menunjukkan urutan pemilihan simpul berikutnya yang terpilih.

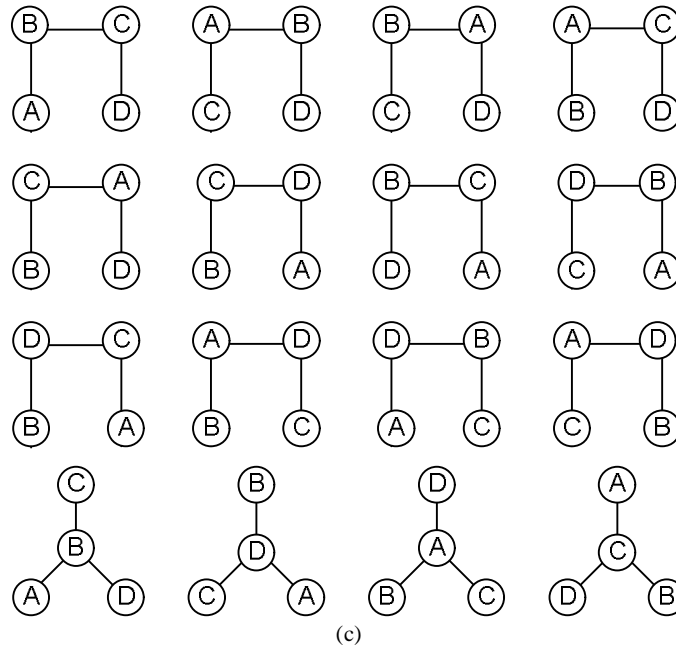


Gambar 2.6. Algoritma menentukan pohon merentang dengan BFS

2.3. Formula Cayley's

Untuk setiap bilangan bulat positif n , banyaknya kemungkinan pohon merentang yang dapat dibuat pada graf berlabel dengan jumlah simpul n adalah n^{n-2} . Contoh penerapan formula Cayley's dapat diamati pada Gambar 2.7. Gambar 2.7(a) kondisi ketika jumlah simpul n bernilai 2, kemungkinan bentuk pohon merentang yang bisa dibentuk adalah $(2^{2-2}) = 1$. Gambar 2.7(b) kondisi ketika jumlah simpul n bernilai 3, kemungkinan bentuk pohon merentang yang bisa dibuat adalah $(3^{3-2}) = 3$. Gambar 2.7(c) kondisi ketika jumlah simpul n bernilai 4, kemungkinan bentuk pohon merentang yang bisa dibuat adalah $(4^{4-2}) = 16$.





Gambar 2.6. Penerapan formula Cayley's pada nilai n yang berbeda-beda. (a) nilai $n=2$. (b) nilai $n=3$. (c) nilai $n=4$.

2.4. Problem: Pohon Merentang Minimum

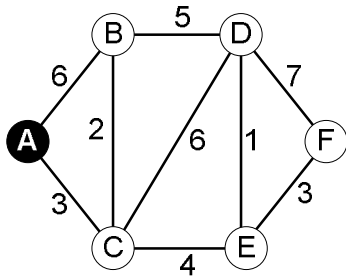
1. Algoritma Prim's

Algoritma ini pertama kali diusulkan pertama kali oleh Vojtěch Jarník pada 1930, tetapi dinamai Prim's karena ditemukan kembali oleh Robert C. Prim pada 1957. Algoritma Prim's dimulai dari sebuah simpul sembarang yang kemudian dijadikan akar. Pada setiap tahap, ditambah busur baru yang bernilai minimum dan tidak membentuk sirkuit dari setiap simpul yang telah terpilih. Algoritma Prim's merupakan jenis algoritma greedy karena pada setiap langkah ditambah dengan busur yang minimum diantara semua busur kandidat yang tersedia.

Input: A non-empty connected weighted graph with vertices V and edges E (the weights can be negative).
Initialize: $V_{\text{new}} = \{x\}$, where x is an arbitrary node (starting point) from V , $E_{\text{new}} = \{\}$
Repeat until $V_{\text{new}} = V$:
 Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
 Add v to V_{new} , and (u, v) to E_{new}
Output: V_{new} and E_{new} describe a minimal spanning tree

Contoh:

Carilah pohon merentang minimum pada graf dibawah ini dengan algoritma Prim's dimana simpul A adalah simpul awal. Bobot pada busur menyatakan biaya untuk menghubungkan *endpoint* pada masing-masing busur.



Jawab:

Step	U	Edge(u,v)	$V - U$	Graf
0	{ }	-	{A, B, C, D, E, F}	-
1	{A}	(A-B) = 6 (A-C) = 3 \checkmark	{ B, C, D, E, F }	
2	{A, C}	(A-B) = 6 (C-B) = 2 \checkmark (C-D) = 6 (C-E) = 4	{ B, D, E, F }	
3	{A, C, B}	(A-B) = 6 cycle (B-D) = 5 (C-D) = 6 (C-E) = 4 \checkmark	{ D, E, F }	
4	{A, C, B, E}	(A-B) = 6 cycle (B-D) = 5 (C-D) = 6 (E-D) = 1 \checkmark (E-F) = 3	{ D, F }	
5	{A, C, B, E, D}	(A-B) = 6 cycle (B-D) = 5 cycle (C-D) = 6 cycle (E-F) = 3 \checkmark (D-F) = 7	{ F }	
6	{A, C, B, E, D, F}	(A-B) = 6 cycle (B-D) = 5 cycle (C-D) = 6 cycle (D-F) = 7 cycle	{ }	

Total biaya minimum untuk membuat pohon merentang adalah $3+2+4+1+3 = 13$.

2. Algoritma Kruskal's

Algoritma Kruskal's diusulkan oleh Joseph Kruskal pada tahun 1956. Algoritma Kruskal's dimulai dengan mengurutkan seluruh busur dari bobot terkecil hingga terbesar. Bobot terkecil yang tidak membuat sirkuit diambil hingga pohon merentang terbentuk. Algoritma Kruskal's dapat dilihat pada fungsi di bawah ini.

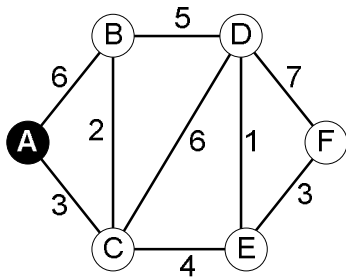
```

function Kruskal( $G = \langle N, A \rangle$ : graph; length:  $A \rightarrow R^+$ ): set of edges
  Define an elementary cluster  $C(v) \leftarrow \{v\}$ 
  Initialize a priority queue  $Q$  to contain all edges in  $G$ , using the weights as keys.
  Define a forest  $T \leftarrow \emptyset$  //  $T$  will ultimately contain the edges of the MST
  //  $n$  is total number of vertices
  while  $T$  has fewer than  $n-1$  edges do
    // edge  $u,v$  is the minimum weighted route from  $u$  to  $v$ 
     $(u,v) \leftarrow Q.\text{removeMin}()$ 
    // prevent cycles in  $T$ . add  $u,v$  only if  $T$  doesn't already contain a path between  $u-v$ .
    // the vertices has been added to the tree.
    Let  $C(v)$  be the cluster containing  $v$ , and let  $C(u)$  be the cluster containing  $u$ .
    if  $C(v) \neq C(u)$  then
      Add edge  $(v,u)$  to  $T$ .
      Merge  $C(v)$  and  $C(u)$  into one cluster, that is, union  $C(v)$  and  $C(u)$ .
  return tree  $T$ 

```

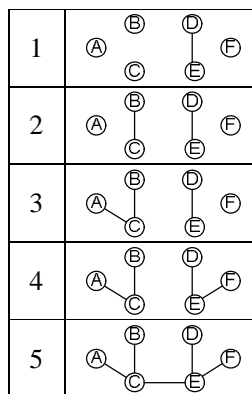
Contoh:

Carilah pohon merentang minimum pada graf dibawah ini dengan algoritma Kruskal's. Bobot pada busur menyatakan biaya untuk menghubungkan *endpoint* pada masing-masing busur.



Jawab:

Edge(u,v)	Bobot	Keterangan
(D,E)	1	Pilih
(B,C)	2	Pilih
(A,C)	3	Pilih
(E,F)	3	Pilih
(C,E)	4	Pilih
(B,D)	5	Cycle
(A,B)	6	Cycle
(C,D)	6	Cycle
(D,F)	7	Cycle



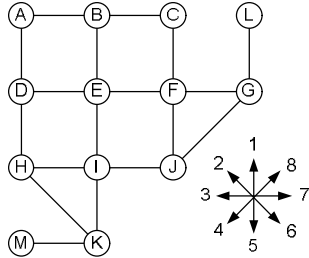
Total biaya minimum untuk membuat pohon merentang adalah $1+2+3+3+4 = 13$.

2.5. Latihan Soal

1. Tentukan pohon merentang dari graf gambar 2 dimana simpul E sebagai akar dengan arah sesuai prioritas yang ditunjukkan oleh gambar panah menggunakan metode :

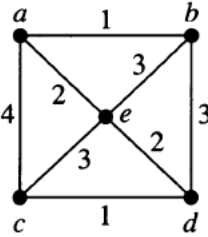
a. BFS

b. DFS

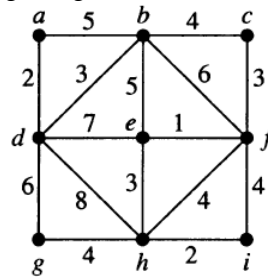


2. Carilah pohon merentang minimum dengan algoritma Prim's dan Kruskal's dari graf di bawah ini:

a.



b.



BAB 3. LINTASAN DAN SIRKUIT

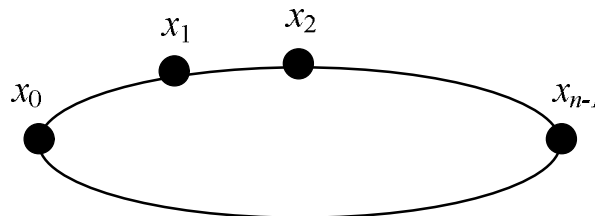
3.1. Pengertian Lintasan dan Sirkuit

Lintasan (*path/ open walk*) dalam graf G adalah e_1, e_2, \dots, e_n di mana $f(e_1) = \{x_0, x_1\}$, $f(e_2) = \{x_1, x_2\}$, \dots , $f(e_n) = \{x_{n-1}, x_n\}$. Panjang lintasan dihitung berdasarkan banyaknya busur pada sebuah jalur, yaitu bernilai n . Definisi ini dapat diilustrasikan pada Gambar 3.1.



Gambar 3.1. Contoh lintasan yang menghubungkan $x_0 - x_n$

Sirkuit (*circuit/ cycle/ closed walk*) dalam graf G adalah lintasan yang memiliki bentuk $\{x_0, x_1\}, \{x_1, x_2\}, \dots, \{x_{n-1}, x_0\}$. Definisi ini dapat diilustrasikan pada Gambar 3.2. Panjang sirkuit dihitung berdasarkan banyaknya busur pada sebuah sirkuit, yaitu bernilai n . Lintasan atau sirkuit disebut sederhana (*simple*) jika tidak berisi busur yang sama lebih dari satu kali.

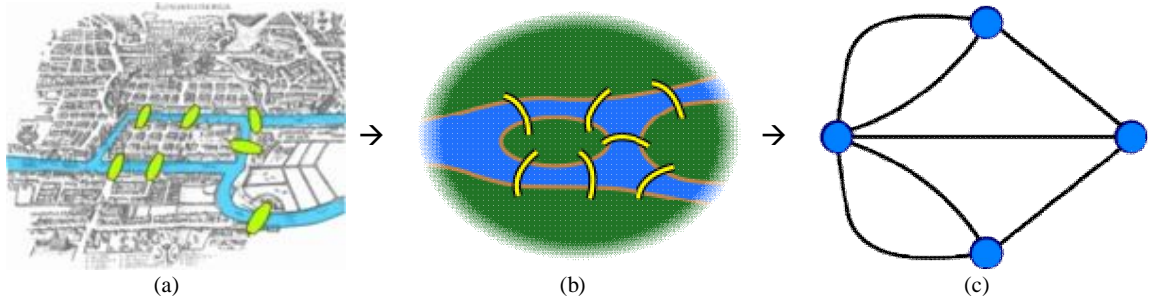


Gambar 3.2. Contoh sirkuit, lintasan berawal dan berakhir di x_0

3.2. Lintasan dan Sirkuit Euler

Lintasan Euler (*Euler path*) adalah suatu jalur sederhana yang melewati semua busur tepat satu kali. Sirkuit Euler (*Euler Circuit*) adalah suatu sirkuit sederhana yang melewati semua busur tepat satu kali. Graf yang mempunyai lintasan Euler dinamakan dengan graf semi-Euler (*semi-Eulerian graph*), sedangkan graf yang mempunyai sirkuit Euler disebut dengan graf Euler (*Eulerian graph*).

Teori mengenai lintasan/ sirkuit Euler merupakan teori graf tertua karena diilhami dari keingintahuan Euler atas 7 jembatan di kota Königsberg pada tahun 1735. Kota Königsberg, sebagaimana terlihat petanya pada Gambar 3.3 (a), dialiri sungai dan memiliki 7 jembatan yang memisahkan antar wilayah berbeda. Euler melemparkan sebuah pertanyaan tentang kemungkinan lintasan yang harus ditempuh agar dia dapat berangkat dari suatu wilayah dan kembali lagi ke wilayah yang sama dengan melewati semua jembatan tepat satu kali. Aliran sungai dan 7 jembatan Kota Königsberg disederhanakan menjadi Gambar 3.3 (b) dan dimodelkan dalam graf pada Gambar 3.3 (c). Pemodelan graf dilakukan dengan memisalkan wilayah sebagai simpul dan jembatan sebagai busur. Akhirnya Euler menarik kesimpulan bahwa seseorang tidak mungkin dapat berangkat dari suatu wilayah dan kembali lagi ke wilayah yang sama dengan melewati semua jembatan tepat satu kali.

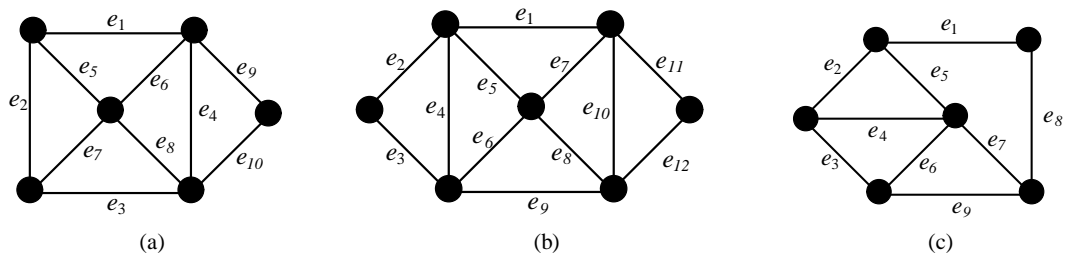


Gambar 3.3. (a) Denah kota Königsberg. (b) sketsa sungai dan jembatan kota Königsberg. (c) Pemodelan sungai dan jembatan kota Königsberg pada graf.

Teorema mengenai lintasan/ sirkuit Euler adalah sebagai berikut:

1. Graf tak-berarah memiliki lintasan Euler jika dan hanya jika terhubung dan memiliki 2 simpul berderajat ganjil atau tidak ada simpul berderajat ganjil sama sekali.
2. Graf tak-berarah G memiliki sirkuit Euler jika dan hanya jika tiap simpul berderajat genap.
3. Graf berarah G memiliki lintasan Euler jika dan hanya jika G terhubung dan setiap simpul memiliki derajat-masuk dan derajat-keluar sama kecuali dua simpul, yang pertama memiliki derajat-keluar satu lebih besar derajat-masuk, dan yang kedua memiliki derajat-masuk satu lebih besar dari derajat-keluar.
4. Graf berarah G memiliki sirkuit Euler jika dan hanya jika G terhubung dan setiap simpul memiliki derajat-masuk dan derajat-keluar sama.

Graf tak-berarah yang memiliki lintasan Euler dicontohkan pada Gambar 3.4(a). Salah satu kemungkinan lintasan Euler pada graf tersebut adalah $e_1, e_9, e_{10}, e_4, e_6, e_8, e_3, e_7, e_5, e_2$. Sedangkan graf tak-berarah yang memiliki sirkuit Euler dicontohkan pada Gambar 3.4(b). Salah satu kemungkinan sirkuit Euler pada graf tersebut adalah $e_1, e_{11}, e_{12}, e_{10}, e_7, e_8, e_9, e_6, e_5, e_4, e_3, e_2$. Graf tak-berarah pada Gambar 3.4(c) tidak memiliki lintasan ataupun sirkuit Euler karena tidak memiliki properti yang disyaratkan pada teorema di atas.



Gambar 3.4. Graf tak-berarah. (a) Graf semi-Euler. (b) Graf Euler. (c) Bukan Graf semi-Euler ataupun Euler.

Graf berarah yang memiliki lintasan Euler dicontohkan pada Gambar 3.5(a). Salah satu kemungkinan lintasan Euler pada graf tersebut adalah e_1, e_4, e_3, e_2, e_5 . Sedangkan graf berarah yang memiliki sirkuit Euler dicontohkan pada Gambar 3.5(b). Salah satu kemungkinan sirkuit Euler pada graf tersebut adalah $e_1, e_2, e_7, e_4, e_3, e_6, e_8, e_5$. Graf berarah pada Gambar 3.5(c) tidak memiliki lintasan ataupun sirkuit Euler karena tidak memiliki properti yang disyaratkan pada teorema di atas.

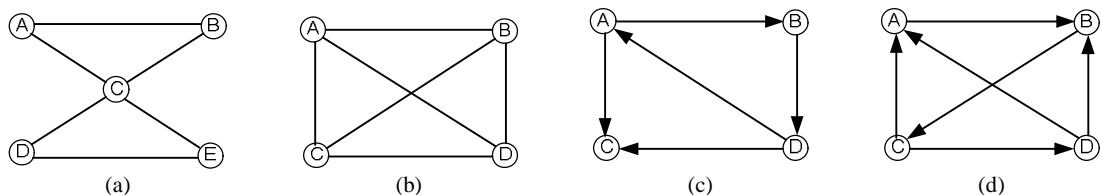
(a) (b) (c)
Gambar 3.5. Graf berarah (a) Graf semi-Euler. (b) Graf Euler. (c) Bukan Graf semi-Euler ataupun Euler.

3.3. Lintasan dan Sirkuit Hamilton

Pada graf $G = (V, E)$, lintasan Hamilton (*Hamiltonian path*) merupakan suatu lintasan sederhana yang melewati semua simpul di G tepat satu kali. Sedangkan dalam graf $G = (V, E)$ sirkuit Hamilton (*Hamiltonian circuit*) merupakan suatu sirkuit sederhana yang melewati semua simpul di G tepat satu kali. Graf yang mempunyai lintasan Hamilton dinamakan dengan graf semi-Hamilton (*semi-Hamiltonian graph*), sedangkan graf yang mempunyai sirkuit Hamilton disebut dengan graf Hamilton (*Hamiltonian graph*).

Contoh lintasan hamilton dapat diamati pada graf tak-berarah Gambar 3.6(a) dan graf berarah Gambar 3.6(c). Salah satu lintasan Hamilton pada graf tak-berarah Gambar 3.6(a) adalah: A-B-C-E-D. Sedangkan salah satu lintasan Hamilton pada graf berarah Gambar 3.6(c) adalah: A-B-D-C.

Contoh sirkuit hamilton dapat diamati pada graf tak-berarah Gambar 3.6(b) dan graf berarah Gambar 3.6(d). Salah satu sirkuit Hamilton pada graf tak-berarah Gambar 3.6(b) adalah: A-B-D-C-A. Sedangkan salah satu sirkuit Hamilton pada graf berarah Gambar 3.6(d) adalah: A-B-C-D-A.



Gambar 3.6. (a) Graf tak-berarah semi-Hamilton. (b) Graf tak-berarah Hamilton. (c) Graf berarah semi-Hamilton. (d) Graf berarah Hamilton.

Hingga saat ini belum ada teorema yang menyatakan syarat perlu dan syarat cukup yang sederhana untuk menentukan suatu graf memiliki lintasan/ sirkuit Hamilton. Akan tetapi, terdapat beberapa teorema umum tentang keberadaan lintasan/ sirkuit Hamilton dengan menggunakan beberapa syarat cukup (bukan syarat perlu), yaitu:

1. **Teorema Dirac.** Jika G adalah graf sederhana dengan n buah simpul ($n \geq 3$) sedemikian sehingga derajat setiap simpul paling sedikit $n/2$ (yaitu, $d(v) \geq n/2$ untuk setiap simpul v di G), maka G adalah graf Hamilton.
2. **Teorema Ore.** Jika G adalah graf sederhana dengan n buah simpul ($n \geq 3$) sedemikian sehingga $d(v) + d(u) \geq n$ untuk setiap pasang simpul tidak bertetangga u dan v , maka G adalah graf Hamilton.
3. Setiap graf lengkap adalah graf Hamilton
4. Di dalam graf lengkap G dengan n buah simpul ($n \geq 3$) terdapat sebanyak $(n-1)!/2$ sirkuit Hamilton
5. Di dalam graf lengkap G dengan n buah simpul ($n \geq 3$ dan n ganjil), terdapat $(n-1)/2$ sirkuit Hamilton yang saling lepas (tidak ada sisi yang beririsan). Jika n genap dan $n \geq 4$, maka di dalam G terdapat $(n-2)/2$ sirkuit Hamilton yang saling lepas.

3.4. Problem

1. Permasalahan Tukang Pos Cina (Chinese Postman Problem/CPP)

Pada tahun 1962, Permasalahan Tukang Pos Cina (CPP) pertama kali dikemukakan oleh ahli matematika bernama Kuan Mei-Ko yang berasal dari Cina. Ia tertarik pada tukang pos dan mengemukakan masalahnya adalah sebagai berikut:

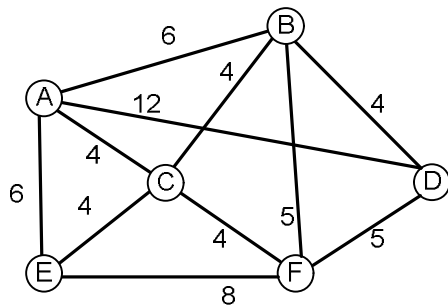
Seorang tukang pos akan mengantarkan surat ke alamat sepanjang jalan di suatu daerah. Ia merencanakan rute perjalanannya dari titik awal (kantor pos) dan melewati semua jalan dengan biaya yang minimum dan kembali lagi ke tempat awal keberangkatannya.

Persoalan ini tak lain adalah persoalan untuk menentukan sirkuit Euler dalam graf. Jika peta jalan tempat tukang pos bekerja memenuhi syarat sebagai sirkuit Euler, maka persoalan ini sudah terselesaikan. Jika tidak, maka tukang pos harus melewati kembali sejumlah jalan yang telah dilaluinya, tentunya dengan biaya minimum (rute terpendek).

Algoritma populer yang digunakan untuk menyelesaikan CPP adalah Algoritma Fleury. Algoritma Fleury adalah algoritma yang menyelesaikan masalah transportasi pada graf berbobot, dengan lebih dahulu menyelidiki graf berarah tersebut merupakan graf Euler atau bukan. Jika graf tersebut bukan graf berarah Euler, maka graf tersebut dibentuk menjadi graf Euler dengan mencari jalur terpendek. Algoritma Fleury akan menduplikasi busur, sehingga diperoleh graf Euler. Pada graf Euler, jarak minimum adalah jumlah seluruh bobot pada semua busur. Sedangkan pada graf bukan Euler, nilai jarak minimum adalah jumlah seluruh bobot busur ditambah busur yang diduplikasi.

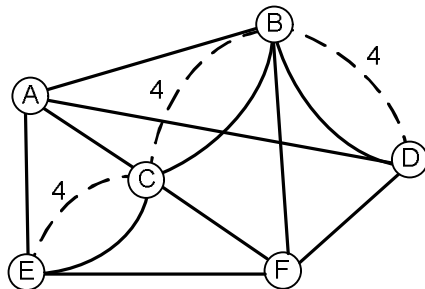
Contoh:

Tentukan nilai bobot minimum CPP pada graf berikut. Jika A sebagai kantor pos (simpul awal), tentukan salah satu rute yang menjadi solusi CPP.



Jawab:

- Graf di atas ternyata tidak memiliki sirkuit Euler, karena terdapat dua simpul berderajat ganjil (E dan D). Total bobot semua busur = 62
- Cari jalur terpendek antara simpul-simpul yang berderajat ganjil (E dan D). Pencarian bisa menggunakan algoritma penentuan jalur terpendek. Akhirnya dengan algoritma Dijkstra, didapatkan jalur terpendek simpul E dan D adalah 12 dengan rute E-C-B-D.
- Gandakan jalur terpendek yang ditemukan sehingga semua simpul berderajat genap



- Jumlah bobot solusi minimum CPP adalah total bobot semua busur graf awal (62) ditambah jumlah bobot jalur yang digandakan (12), sehingga hasilnya adalah 74.
- Salah satu rute yang mungkin dilewati oleh tukang pos adalah: A-B-D-B-F-D-A-C-F-E-C-B-C-E-A

2. Permasalahan Perjalanan Sales (Traveling Salesman Problem/TSP)

Permasalahan Perjalanan Sales (TSP) pertama kali diperkenalkan oleh Rand pada tahun 1948. TSP adalah masalah seorang sales yang harus melakukan kunjungan ke sejumlah kota dalam menjajakan produknya. Rangkaian kota-kota yang dikunjungi harus membentuk suatu jalur sedemikian sehingga kota-kota tersebut hanya boleh dilewati tepat satu kali dan kemudian kembali lagi ke kota awal dengan

jarak tempuh minimum. Dari penjelasan di atas, dapat diamati bahwa TSP merupakan implementasi dari sirkuit Hamilton.

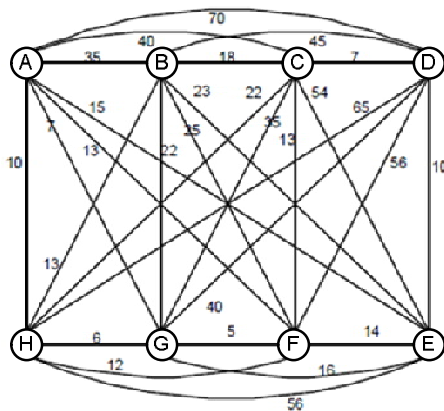
Penyelesaian secara eksak terhadap persoalan ini melibatkan algoritma yang mengharuskan untuk mencari kemungkinan semua solusi yang ada. Sebagai akibatnya, kompleksitas waktu dari eksekusi algoritma ini akan menjadi eksponensial terhadap ukuran dari masukan yang diberikan. Metode baru yang diusulkan untuk optimasi TSP dari sisi hasil dan kompleksitas hingga saat ini masih terus berkembang.

Salah satu algoritma klasik yang digunakan untuk memecahkan TSP adalah prinsip greedy heuristik. Berikut adalah langkah-langkah yang dilakukan:

1. Cari pohon merentang minimum (MST) pada graf $G = (V, E)$, disarankan gunakan algoritma Kruskal.
2. Carilah simpul dengan derajat ganjil. Buat jalur pasangan busur berderajat ganjil yang memiliki bobot minimum, sehingga graf semua berderajat genap.
3. Periksa setiap simpul yang dikunjungi lebih dari 1 kali dan perbaiki solusi TSP dengan menerapkan pertidaksamaan: $l(a, b) < l(a, c) + l(c, b)$

Contoh:

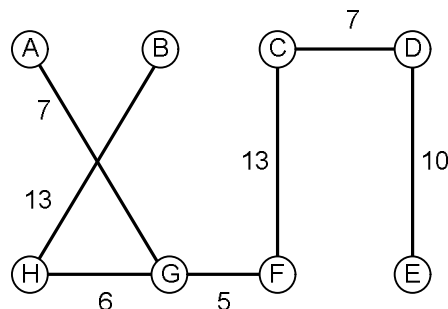
Tentukan nilai bobot minimum TSP pada graf G berikut ini. Jika A sebagai simpul awal, tentukan salah satu rute yang menjadi solusi TSP.



Jawab:

Langkah 1

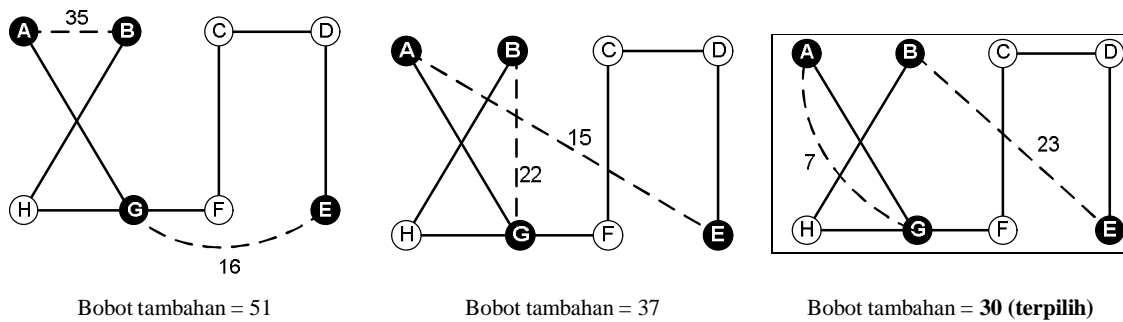
Cari pohon merentang minimum (MST) pada graf $G = (V, E)$, disarankan gunakan algoritma Kruskal. Hasil pencarian MST dengan Kruskal's adalah graf di bawah ini.



Langkah 2

Carilah simpul dengan derajat ganjil. Buat jalur pasangan simpul berderajat ganjil yang memiliki bobot minimum, sehingga graf semua berderajat genap.

Semua kemungkinan pasangan simpul berderajat ganjil adalah sbb:



Dipilih graf dengan bobot tambahan minimum yaitu 30

Langkah 3

Periksa setiap simpul yang dikunjungi lebih dari 1 kali dan perbaiki solusi TSP dengan menerapkan pertidaksamaan: $l(a,b) < l(a,c) + l(c,b)$.

Dalam hal ini, simpul G dilewati 2 kali, sehingga dilakukan pengecekan.

Perjalanan A ke H diperbaiki dari A-G-H menjadi A-H karena:

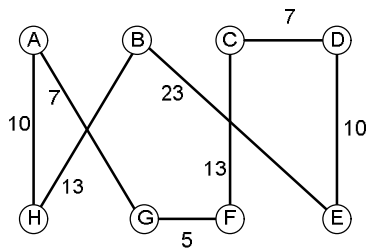
$$l(A,H) = 10$$

$$l(A,G) = 6$$

$$l(G,H) = 7.$$

$$l(A,H) < l(A,G) + l(G,H).$$

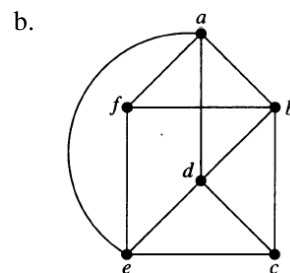
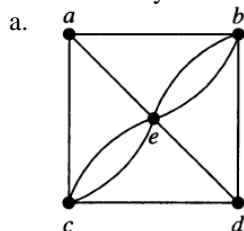
Sehingga, hasil akhir MST adalah sbb:

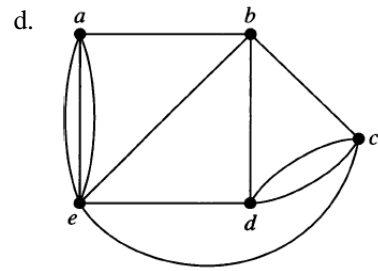
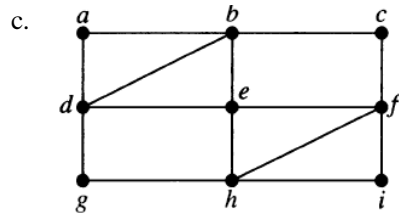


Total bobot untuk MST tersebut adalah 88.

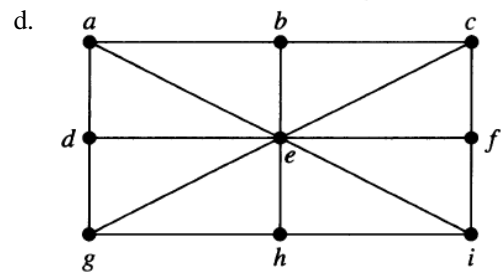
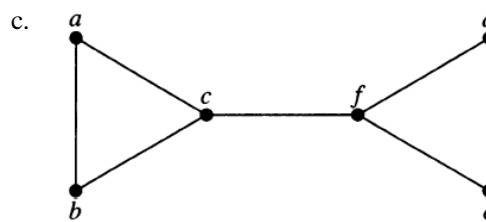
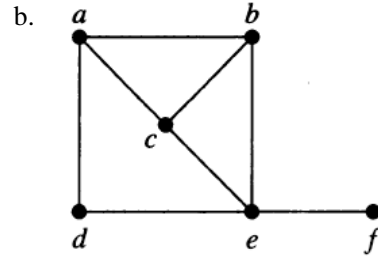
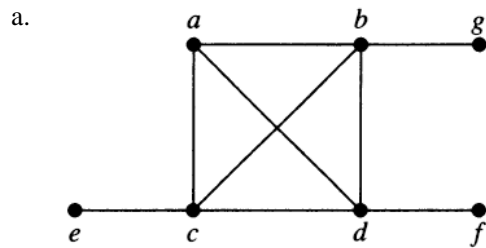
3.5. Latihan Soal

1. Tunjukkan apakah graf yang diberikan di bawah ini termasuk graf euler, graf semi-euler, atau bukan keduanya.

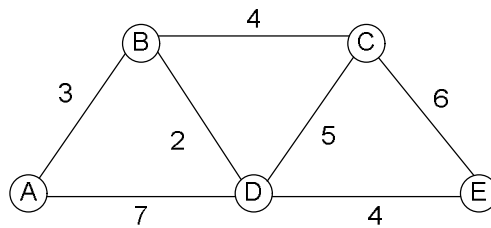




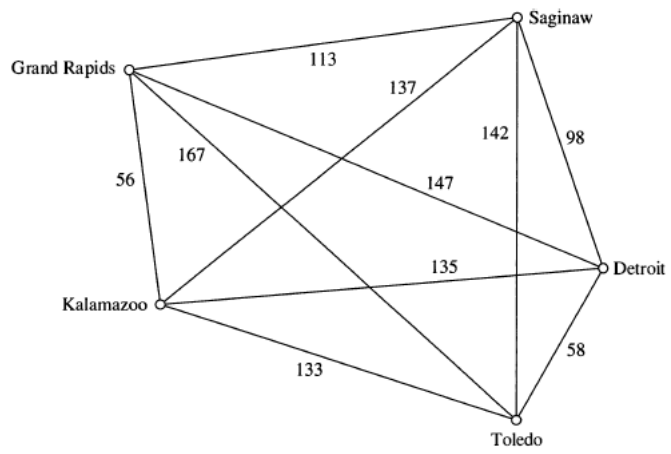
2. Tunjukkan apakah graf yang diberikan di bawah ini termasuk graf hamilton, graf semi-hamilton, atau bukan keduanya.



3. Tentukan nilai bobot minimum CPP pada graf berikut. Jika A sebagai kantor pos (simpul awal), tentukan salah satu rute yang menjadi solusi CPP.



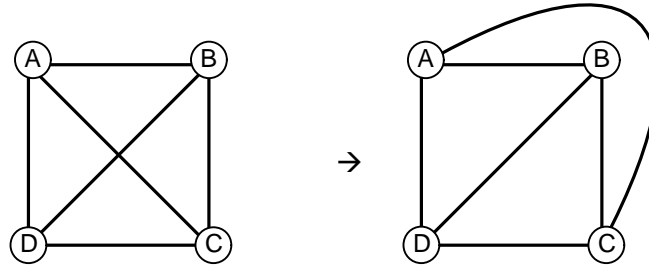
4. Tentukan nilai bobot minimum TSP pada graf berikut ini. Jika Kalamazoo dijadikan sebagai simpul awal, tentukan salah satu rute yang menjadi solusi TSP.



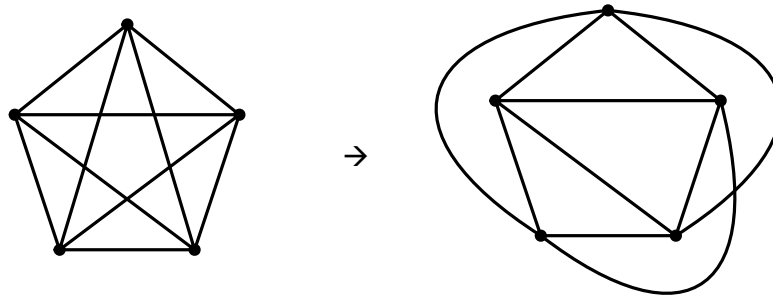
BAB 4. PLANARITAS

4.1. Graf Planar

Graf planar adalah graf yang dapat digambarkan pada bidang datar dengan busur tidak saling berpotongan. Contoh graf planar dapat dilihat di Gambar 4.1. Sedangkan Gambar 4.2 menunjukkan graf non-planar karena dimodifikasi bagaimanapun tetap ada busur yang saling berpotongan.

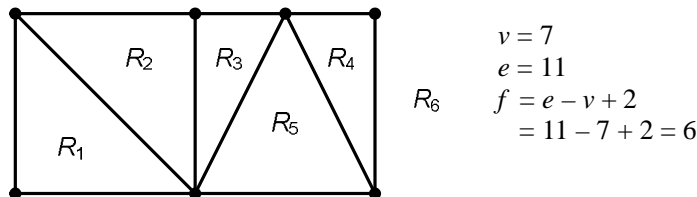


Gambar 4.1. Graf planar karena dapat digambarkan tanpa ada busur yang berpotongan



Gambar 4.2. Bukan graf planar karena tidak bisa dibentuk tanpa ada busur yang berpotongan

Edge pada graf planar membagi bidang menjadi beberapa wilayah (*region*) atau muka (*face*). Region adalah daerah yang dibatasi oleh busur-busur pada sebuah graf. Region pada graf planar diilustrasikan pada Gambar 4.3. Jumlah region pada graf tersebut adalah 6. Pada contoh graf tersebut, region R_6 merupakan region yang berada di luar graf.



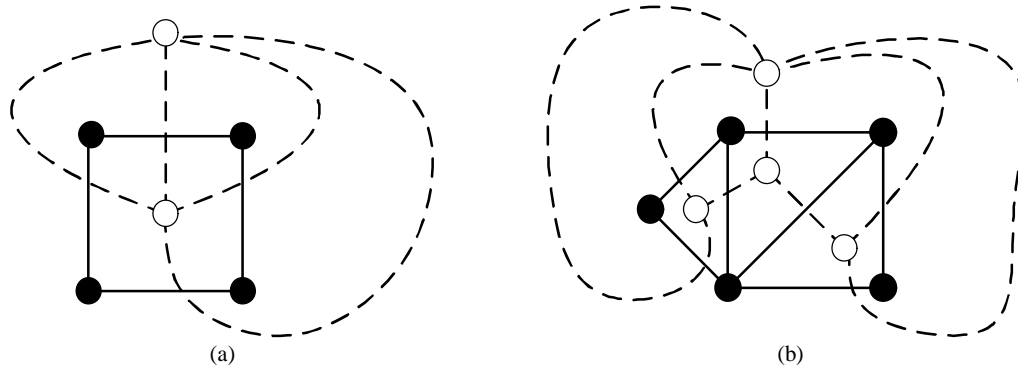
Gambar 4.3. Contoh graf planar dengan jumlah region 6

Formula Euler tentang region adalah $v - e + f = 2$, dimana v adalah jumlah simpul, e adalah jumlah busur, dan f adalah jumlah *face*/region. Sebagai contoh dapat diterapkan pada graf di Gambar 4.3. Jumlah simpul $v=7$, jumlah busur $e = 11$, sehingga jumlah region $f = e - v + 2 = 6$.

Graf sederhana disebut planar maksimum (*maximal planar*) jika berbentuk planar tetapi dengan adanya penambahan busur akan menjadikan graf tersebut menjadi tidak planar.

4.2. Graf Dual

Misalkan terdapat sebuah graf planar G , maka dapat dibuat graf dual dari graf G yaitu graf G' yang memiliki simpul dalam setiap region G dan busur untuk setiap busur di graf G yang menghubungkan 2 region bertetangga.

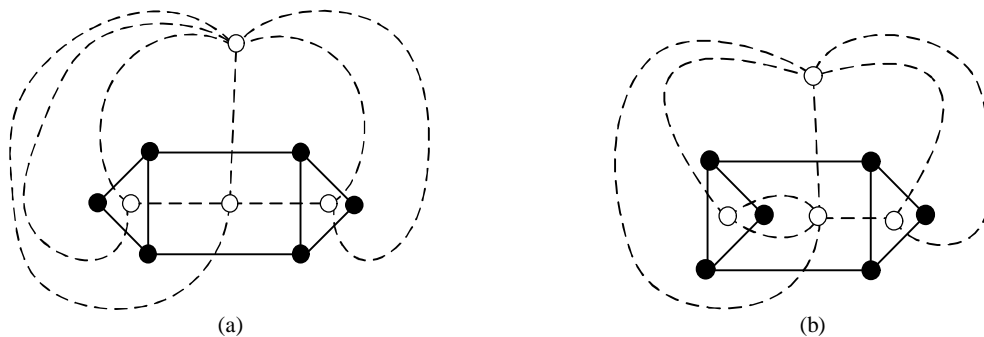


Gambar 4.4. (a) Graf G adalah graf dengan simpul hitam dan busur solid. Graf G' yang merupakan graf dual dari graf G adalah graf dengan simpul putih dan busur putus-putus. (b) Graf H adalah graf dengan simpul hitam dan busur solid. Graf H' yang merupakan graf dual dari graf H adalah graf dengan simpul putih dan busur putus-putus.

Contoh graf dual dapat diamati pada graf G' di Gambar 4.4 (a) dan graf H' di Gambar 4.4 (b). Graf G' dan H' yang memiliki simpul warna putih dan busur berupa garis putus-putus secara berurutan merupakan graf dual dari graf planar G dan H .

Adapun properti yang dimiliki oleh graf dual adalah sebagai berikut:

1. Graf dual dari suatu graf planar adalah graf planar
2. Memiliki sifat simetris, jika G graf planar dan G' adalah graf dual dari graf G maka graf G adalah graf dual dari graf G' . Misalnya pada graf G dan G' di gambar 4.4 (a), graf G' merupakan graf dual dari graf G dan sebaliknya, graf G merupakan graf dual dari graf G'
3. Tidak *unique*, graf yang isomorfis dapat menghasilkan graf dual yang tidak isomorfis. Misalnya graf G yang memiliki simpul hitam pada Gambar 4.5(a) isomorfis dengan graf H yang memiliki simpul hitam pada Gambar 4.5(a). Graf G dan H berturut-turut memiliki graf dual berupa graf G' dan H' yang memiliki simpul putih di Gambar 4.5(a) dan (b). Graf G' dan H' bukanlah graf yang isomorfis walaupun G dan H adalah graf yang isomorfis.

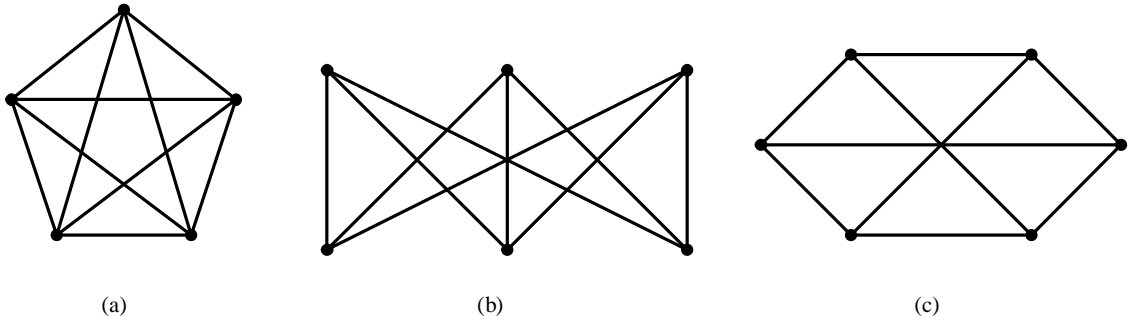


Gambar 4.5. (a) Graf G adalah graf dengan simpul hitam dan busur solid. Graf G' yang merupakan graf dual dari graf G adalah graf dengan simpul putih dan busur putus-putus. (b) Graf H adalah graf dengan simpul hitam dan busur solid. Graf H' yang merupakan graf dual dari graf H adalah graf dengan simpul putih dan busur putus-putus. Graf G isomorfis dengan graf H , namun graf G' tidak isomorfis dengan graf H' .

4.3. Problem

1. Teorema Kurawtoski

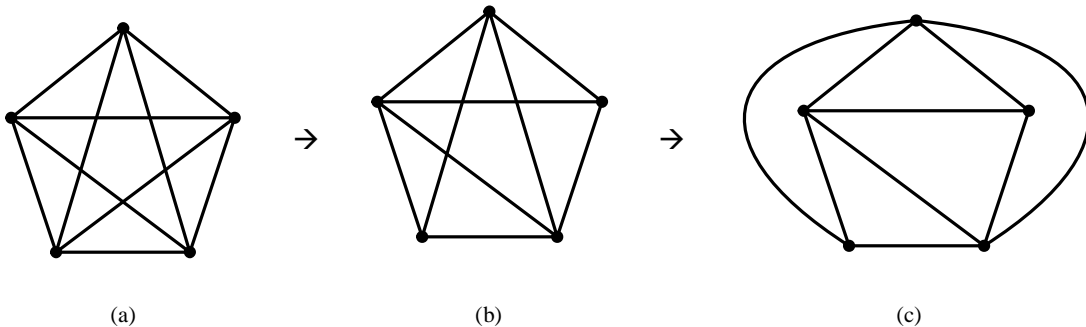
Teorema Kuratowski digunakan untuk menentukan apakah sebuah graf termasuk graf planar atau bukan. Terdapat dua jenis graf Kuratowski. Graf Kuratowski pertama adalah graf K_5 seperti yang terlihat pada Gambar 4.6(a). Sedangkan graf Kuratowski kedua ada;ah graf yang memiliki 6 simpul yaitu graf $K_{3,3}$ dan K_6 seperti yang ditunjukkan pada Gambar 4.6 (b) dan (c). Seluruh jenis graf Kuratowski yang terdapat pada Gambar 4.6 adalah bukan graf planar.



Gambar 4.6. (a). Graf Kuratowski jenis pertama. (b)(c). Graf Kuratowski jenis kedua.

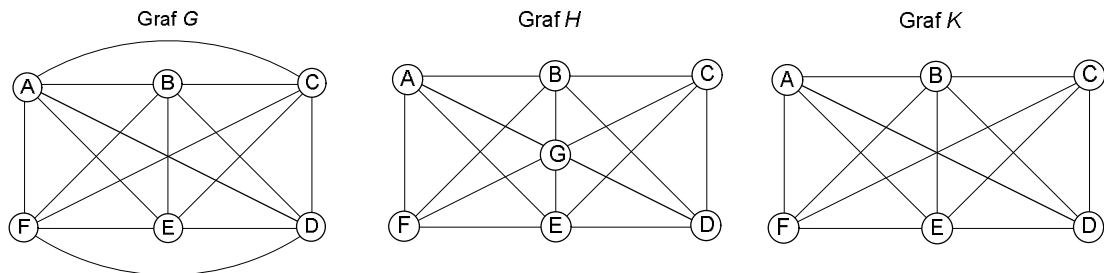
Adapun sifat-sifat dari graph Kuratowski adalah sebagai berikut:

1. Kedua graf Kuratowski adalah graph teratur
2. Kedua graf Kuratowski adalah graph tidak-planar
3. Penghapusan simpul atau busur dari graf Kuratowski menyebabkannya menjadi graph planar. Contoh penghapusan simpul ditunjukkan pada Gambar 4.7. Gambar 4.7(b) adalah graf Kuratowski di Gambar 4.7(a) yang diholangkan salah satu busurnya. Ternyata Gambar 4.7(b) adalah graf planar sebagaimana ditunjukkan oleh Gambar 4.7(c).



Gambar 4.7. (a). Graf Kuratowski (b). dilakukan penghapusan salah satu busur di (a) (c). bentuk planar dari graf (b)

Teorema Kuratowski berbunyi, suatu graf bersifat planar jika dan hanya jika tidak mengandung sub-graf yang sama dengan salah satu graf Kuratowski atau homeomorfik (*homeomorphic*) dengan salah satu dari keduanya.



(a) (b) (c)
Gambar 4.8. (a). Graf G (b). Graf H (c). Graf Kuratowski K yang merupakan sub-graf dari graf G dan homeomorfik dengan graf H

Graf G pada Gambar 4.8(a) bukan merupakan graf planar karena merupakan sub-graf graf Kuratowski K (graf Kuratowski jenis ke-2). Sedangkan graf H pada Gambar 4.8(b) bukan merupakan graf planar karena homeomorfik dengan graf Kuratowski K .

2. Teorema Fáry's

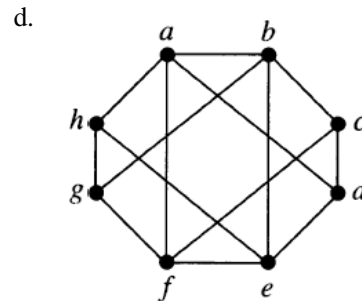
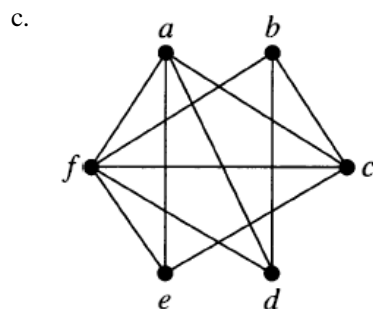
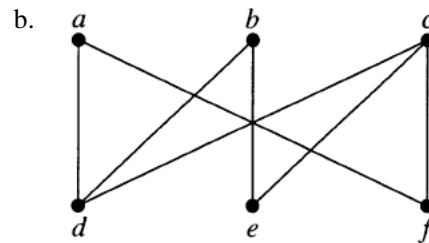
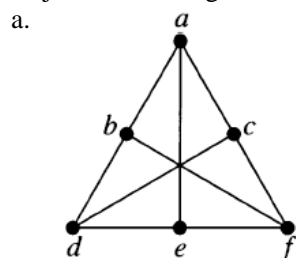
Teorema Fáry's menyatakan bahwa graf planar sederhana dapat dibuat tanpa persilangan yang menyebabkan salah satu/beberapa busurnya berbentuk kurva. Alasan terbentuknya teorema itu ialah, karena jika didalam graph terdapat kurva, maka graph tersebut tidak bisa dikembangkan menjadi graph yang lebih besar.

Induksi langkah untuk membuktikan Teorema Fáry's adalah sebagai berikut:

1. Misal G adalah graf planar sederhana dengan n simpul, busur dapat ditambahkan jika diperlukan sehingga graf planar G maksimal.
2. Semua region di graf G akan menjadi segitiga, dapat ditambahkan busur ke setiap region sambil menjaga planarity, bertentangan dengan asumsi graf planar maksimum.
3. Pilih sekitar tiga simpul a, b, c untuk membentuk region segitiga G . Terbukti dengan cara induksi pada n bahwa terdapat garis lurus *embedding* dari G di mana segitiga abc adalah region luar *embedding*.
4. jika $n = 3$ dan a, b , dan c adalah satu-satunya simpul di G maka hasilnya adalah trivial. Jika tidak, semua simpul pada G memiliki lebih dari atau sama dengan tiga tetangga.

4.4. Latihan Soal

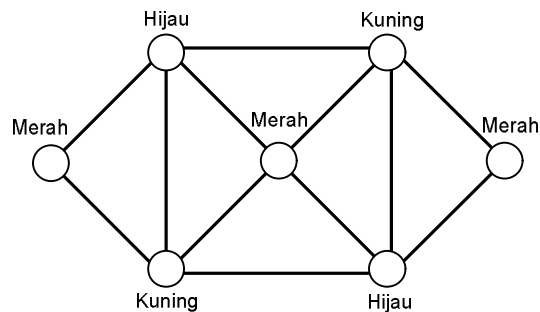
1. Misalkan sebuah graf planar memiliki 20 simpul, masing-masing memiliki derajat 3. Berapakah banyak region yang bisa dibentuk?
2. Tunjukkan bahwa graf di bawah ini termasuk planar atau tidak



BAB 5. PEWARNAAN DAN PENCOCOKAN GRAF

5.1. Pewarnaan Graf

Pewarnaan pada graf G adalah pewarnaan pada setiap simpul sedemikian hingga tidak ada dua simpul yang bertetangga memiliki warna yang sama. Seperti dicontohkan pada graf di Gambar 5.1, tidak simpul-simpul yang berdekatan selalu memiliki warna yang berbeda. Warna yang dipakai oleh suatu simpul boleh dipakai kembali oleh simpul lain yang tidak bertetangga.



Gambar 5.1. Pewarnaan graf. Simpul yang bertetangga memiliki warna yang berbeda

5.2. Bilangan Kromatis (*chromatic number*)

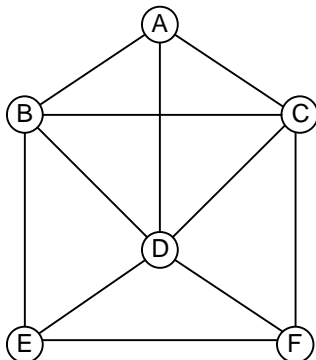
Bilangan kromatis graf G yang dinotasikan dengan $\chi(G)$ adalah jumlah minimum warna berbeda yang diperlukan untuk melakukan pewarnaan graf. Nilai $\chi(G)$ maksimum yang mungkin dimiliki oleh graf planar adalah 4. Nilai tersebut telah dibuktikan oleh Appel dan Haken pada tahun 1976 dimana dibutuhkan ratusan halaman kertas untuk menuliskannya.

Algoritma yang dapat digunakan untuk mewarnai sebuah graf G adalah algoritma Welch-Powell. Akan tetapi, algoritma ini hanya memberikan batas atas untuk nilai $\chi(G)$, artinya algoritma ini tidak selalu memberikan jumlah warna minimum yang diperlukan untuk mewarnai graf G . Algoritma Welch-Powell adalah sebagai berikut:

1. Urutkan simpul-simpul pada graf G dari simpul berderajat tinggi hingga berderajat rendah.
2. Ambil sebuah simpul derajat tertinggi yang belum pernah dikunjungi, jika belum terwarnai maka warnai dengan warna yang unik.
3. Warnai dengan warna yang sama sebuah simpul lain yang tidak bertetangga dengannya dan belum pernah diwarnai
4. Ulangi langkah nomor 2 dan 3 sampai semua simpul terwarnai.

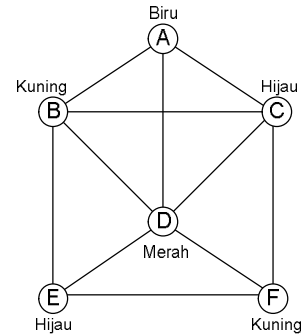
Contoh:

Warnai graf di bawah ini dengan algoritma Welch-Powell!



Jawab:

Simpul	Derajat	Warna	Simpul yg tidak bertetangga
D	5	merah	-
B	4	kuning	F (dipilih)
C	4	hijau	E (dipilih)
A	3	biru	E, F (sudah diwarnai)
E	3	hijau	A, C (sudah diwarnai)
F	3	kuning	A, B (sudah diwarnai)



5.3. Polinomial Kromatik (*Chromatic Polynomial*)

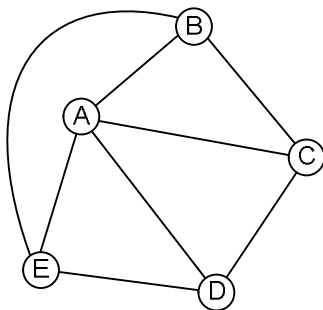
Nilai polinomial kromatik $P_n(\lambda)$ dari graf dengan n simpul adalah banyaknya cara yang benar untuk mewarnai graf dengan warna berjumlah kurang dari atau sama dengan λ . Jika C_i adalah banyaknya cara untuk mewarnai graf dengan warna sejumlah i , maka kombinasi untuk memilih warna yang dipakai sejumlah i dapat dipilih dengan rumus kombinasi $\binom{\lambda}{i}$ dimana $1 \leq i \leq n$. Misalnya disediakan 5 warna berbeda untuk mewarnai sebuah graf, jika dipakai 3 warna saja dari 5 warna yang disediakan maka $\binom{5}{3} = 10$ cara untuk memilih warna yang dipakai. Dari penjelasan tersebut, maka nilai $P_n(\lambda)$ dapat dihitung dengan

$$P_n(\lambda) = \sum_{i=1}^n C_i \binom{\lambda}{i}$$

$$P_n(\lambda) = C_1 \frac{\lambda}{1!} + C_2 \frac{\lambda(\lambda-1)}{2!} + C_3 \frac{\lambda(\lambda-1)(\lambda-2)}{3!} + \dots + C_n \frac{\lambda(\lambda-1)(\lambda-2)\dots(\lambda-n+1)}{n!}$$

Contoh:

Tentukan nilai polinomial kromatik dari graf di bawah ini!



Jawab:

$$P_5(\lambda) = C_1 \frac{\lambda}{1!} + C_2 \frac{\lambda(\lambda-1)}{2!} + C_3 \frac{\lambda(\lambda-1)(\lambda-2)}{3!} + C_4 \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)}{4!} + C_5 \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)(\lambda-4)}{5!}$$

Karena graf tersebut memiliki sub-graf yang berbentuk segitiga, maka minimal dibutuhkan 3 warna untuk mewarnainya. Sehingga $C_1 = C_2 = 0$.

C_3 :

Misalkan terdapat 3 warna x, y dan z . Ketiga warna dapat diberikan pada simpul A, B dan C dengan banyaknya kemungkinan susunan warna $P_3^3 = 3!$ cara yang berbeda. Karena warna yang tersedia hanya 3, maka warna simpul D harus sama dengan simpul B dan simpul E harus sama dengan simpul C (artinya hanya ada 1 cara untuk mewarnai simpul D dan E).

Jadi, $C_3 = 3!$

C_4 :

Misalkan terdapat 4 warna w, x, y dan z . 4 warna dapat diberikan pada 3 simpul yang membentuk segitiga yaitu simpul A, B dan C dengan banyaknya kemungkinan susunan warna $P_3^4 = 4!$. Warna keempat dapat diberikan untuk simpul D atau E , sehingga memiliki 2 cara untuk memberikan warna pada simpul ke-4. Simpul ke-5 hanya ada 1 pilihan, yaitu mengikuti salah satu warna yang dipakai sebelumnya.

Jadi, $C_4 = 4! \cdot 2$

C_5 :

Misalkan terdapat 5 warna untuk mewarnai 5 simpul maka kemungkinannya pasti $P_5^5 = 5!$ cara.

Jadi, $C_5 = 5!$

$$P_5(\lambda) = C_1 \frac{\lambda}{1!} + C_2 \frac{\lambda(\lambda-1)}{2!} + C_3 \frac{\lambda(\lambda-1)(\lambda-2)}{3!} + C_4 \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)}{4!} + C_5 \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)(\lambda-4)}{5!}$$

$$P_5(\lambda) = 0 + 0 + 3! \frac{\lambda(\lambda-1)(\lambda-2)}{3!} + 4! \cdot 2 \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)}{4!} + 5! \frac{\lambda(\lambda-1)(\lambda-2)(\lambda-3)(\lambda-4)}{5!}$$

$$P_5(\lambda) = \lambda(\lambda-1)(\lambda-2) + 2 \lambda(\lambda-1)(\lambda-2)(\lambda-3) + \lambda(\lambda-1)(\lambda-2)(\lambda-3)(\lambda-4)$$

$$P_5(\lambda) = \lambda(\lambda-1)(\lambda-2)(\lambda^2 - 5\lambda + 7)$$

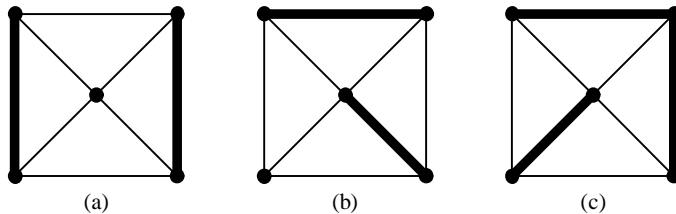
Agar $P_5(\lambda)$ tidak bernilai 0, maka nilai $\lambda \geq 3$ karena adanya faktor $(\lambda-2)$. Sehingga nilai kromatik graf tersebut adalah 3.

Jika diberikan 5 warna berbeda untuk mewarnai graf tersebut maka banyaknya cara untuk mewarnai graf tersebut adalah:

$$P_5(5) = 5(5-1)(5-2)(5^2 - 5 \cdot 5 + 7) = 420 \text{ cara.}$$

5.4. Pencocokan (*matching*)

Pencocokan adalah pemilihan busur (yang memasangkan simpul) tanpa ada busur bertangga yang terpilih. Hasil pencocokan dapat diamati pada graf di Gambar 5.2(a) dan (b), terlihat tidak ada busur bertetangga yang terpilih. Sedangkan graf di Gambar 5.2(c) bukan merupakan hasil pencocokan karena ada busur bertetangga yang terpilih.



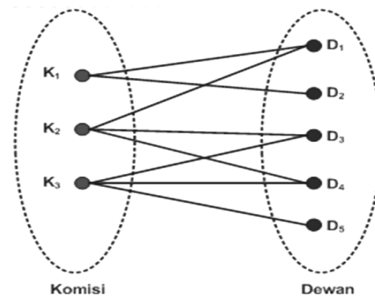
Gambar 5.2. (a)(b) Graf hasil pencocokan. (c) Bukan graf hasil pencocokan.

Sebuah pencocokan maksimal (*maximal matching*) adalah kondisi pencocokan dimana jika dipilih lagi busur maka menjadikan pencocokan menjadi salah. Sebuah graf memang dapat memiliki lebih satu bentuk pencocokan. Pencocokan maksimal yang beranggotakan busur terbanyak disebut pencocokan maksimal terbesar (*largest maximal matching*). Graf di Gambar 5.3 (a) dan (b) merupakan contoh

pencocokan maksimal terbesar, sedangkan graf di Gambar 5.3 (c) dan (b) merupakan contoh pencocokan maksimal tetapi bukan pencocokan maksimal terbesar.

(a) (b) (c)
Gambar 5.3. (a)(b) Graf hasil pencocokan maksimal terbesar. (c) Bukan graf hasil pencocokan maksimal terbesar.

Untuk lebih memahami tentang konsep pencocokan maka akan diberikan sebuah studi kasus. Dimisalkan terdapat 5 anggota dewan yang merupakan anggota dari 3 komisi. Satu anggota dari setiap komisi akan ditunjuk untuk mewakili sebuah kepanitiaan *ad-hoc*. Pemetaan kelima anggota dewan terhadap 3 komisi digambarkan seperti Gambar 5.4. Maka, mungkinkah setiap komisi dapat mengirimkan satu wakilnya ? Jika tidak, berapa jumlah posisi maksimum yang dapat terisi ?



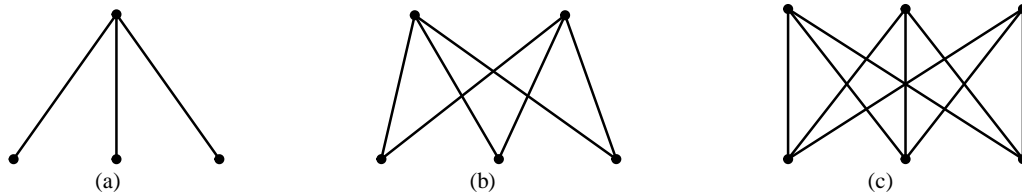
Gambar 5.4. Pemetaan 5 anggota dewan terhadap 3 komisi

Permasalahan di atas merupakan salah satu permasalahan dalam teori pencocokan dari sebuah himpunan simpul ke himpunan simpul lain. Pencocokan yang baik pada sebuah graf akan menghasilkan sub-himpunan busur yang anggotanya saling *disjoint* (tidak ada busur yang bertetangga).

5.5. Graf Bipartite

Pencocokan memang dapat didefinisikan pada sebarang graf, namun dalam banyak aplikasi metode pencocokan ini paling sesuai untuk permasalahan graf *bipartite* (*bigraph*). Graf *bipartite* adalah graf yang simpulnya dapat diuraikan dalam dua himpunan saling lepas V_1 dan V_2 sedemikian hingga setiap busur menghubungkan sebuah simpul dari V_1 ke V_2 dan tidak ada busur yang menghubungkan simpul dalam satu himpunan. Gambar 5.4 merupakan contoh dari graf *bipartite*.

Graf bipartite lengkap adalah graf yang semua simpulnya terhubung oleh busur ke titik-titik lain. Gambar 5.5 merupakan contoh graf bipartite lengkap.



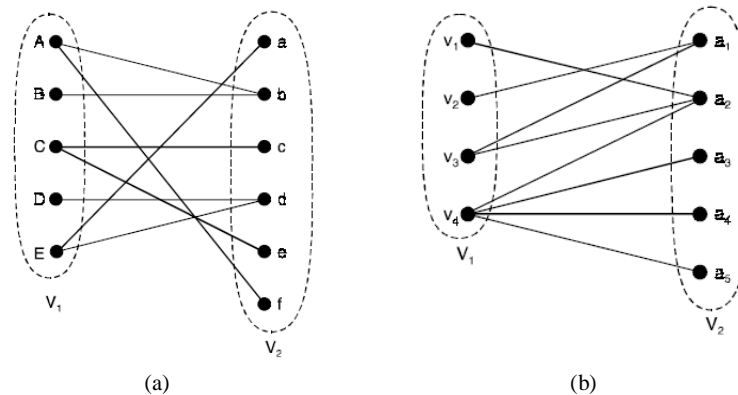
Gambar 5.5. Graf bipartite lengkap (a) $K_{1,3}$. (b) $K_{2,3}$. (c) $K_{3,3}$.

Pada graf *bipartite* yang memiliki partisi himpunan simpul V_1 dan V_2 , dapat ditentukan memiliki pencocokan komplet ketika setiap simpul di V_1 pasti akan terpetakan oleh sebuah busur ke himpunan simpul V_2 . Sebuah pencocokan komplet secara otomatis dapat dianggap sebagai pencocokan maksimal

terbesar. Contoh graf *bipartite* yang memiliki pencocokan komplit dapat dilihat pada Gambar 5.6(a). Pada graf *bipartite* terdapat nilai *deficiency* $d(G)$, yaitu nilai terbesar dari $r - q$ dimana r adalah himpunan bagian simpul di V_1 yang berinsiden dengan himpunan bagian simpul q di V_2 .

Adapun teorema mengenai graf bipartite adalah sebagai berikut:

1. Pencocokan komplit pada graf *bipartite* dapat terbentuk jika terdapat konstanta m sedemikian hingga kondisi berikut terpenuhi :
derajat setiap simpul di $V_1 \geq m \geq$ derajat setiap simpul di V_2 .
2. **Teorema Hall's Marriage.** Sebuah pencocokan komplit dari V_1 ke V_2 dalam graf *bipartite* dapat terjadi jika dan hanya jika setiap subset r simpul di V_1 bertetangga secara kolektif dengan r atau lebih di simpul V_2 untuk semua nilai r . Teorema ini secara implisit menyatakan bahwa pencocokan komplit dapat terbentuk pada graf *bipartite* jika dan hanya jika $d(G) \leq 0$. Contoh graf *bipartite* yang memiliki pencocokan lengkap dapat dilihat pada Gambar 5.6(a). Sedangkan graf pada Gambar 5.6(b) tidak memiliki pencocokan lengkap karena gagal untuk himpunan bagian $r = \{v_1, v_2, v_3\}$ yang memiliki himpunan bagian $s = \{a_1, a_2\}$.



Gambar 5.6. (a) Graf bipartite dengan pencocokan komplit. (b) Graf bipartite yang tidak memiliki pencocokan komplit.

3. Jumlah maksimal simpul di V_1 yang dapat terpetakan ke V_2 pada graf *bipartite* adalah jumlah simpul di $V_1 - d(G)$.

5.6. Problem

1. Penjadwalan (*Scheduling*)

Masalah penjadwalan adalah aplikasi pada pewarnaan graf. Biasanya aplikasi pewarnaan graf ini dilakukan untuk mengetahui shift minimum yang harus dibuat agar jadwal peserta tidak ada yang berbenturan.

Contoh:

Diketahui suatu universitas akan menjadwalkan UAS 7 mata kuliah. Pasangan mata kuliah yang mahasiswanya beririsan adalah: A dan B, A dan C, A dan D, A dan G, B dan C, B dan D, B dan E, B dan G, C dan D, C dan F, C dan G, D dan E, D dan F, E dan F, E dan G, serta F dan G. Manakah jadwal ujian yang dapat dijadwalkan bersamaan?

Jawab:

Dimisalkan setiap mata kuliah adalah sebuah simpul. Mata kuliah yang memiliki mahasiswa yang beririsan berarti adalah simpul yang bertetangga. Simpul yang bertetangga tidak boleh diberikan warna yang sama, artinya mata kuliah yang memiliki irisan mahasiswa peserta tidak boleh dijadwalkan bersamaan. Simpul yang bertetangga adalah:

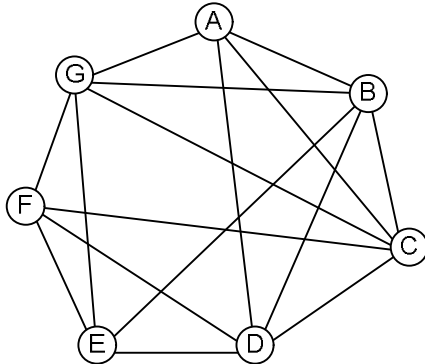
Simpul A: A-B, A-C, A-D, A-G

Simpul B: B-C, B-D, B-E, B-G

Simpul C: C-D, C-F, C-G

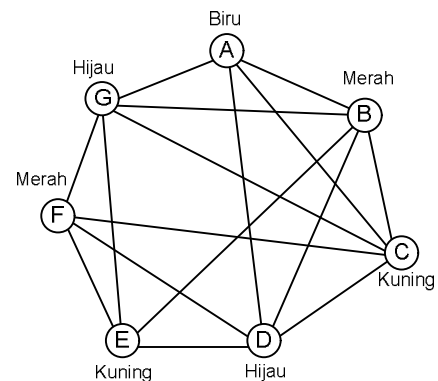
Simpul D: D-E, D-F
 Simpul E: E-F, E-G
 Simpul F: F-G

Sehingga bentuk graf adalah sebagai berikut:



Dikerjakan dengan Algoritma Welch-Powel maka didapatkan:

Simpul	Derajat	Warna	Simpul yg tidak bertetangga
B	5	merah	F (dipilih)
C	5	kuning	E (dipilih)
D	5	hijau	G (dipilih)
G	5	hijau	D (sudah diwarnai)
A	4	biru	E,F (sudah diwarnai)
E	4	kuning	A,C (sudah diwarnai)
F	4	merah	A,B (sudah diwarnai)



Jadwal ujian yang dapat dilaksanakan bersama adalah mata kuliah: B dan F, C dan E, D dan G, serta A harus diadakan sendiri. Sehingga ujian tersebut dijadwalkan minimal 4 *shift*.

2. Permasalahan Penugasan Person (*Personnel Assignment Problem*)

Pada 1955, Harold Kuhn, seorang matematikawan Amerika mempublikasikan algoritma Hungarian. Algoritma Hungarian adalah sebuah algoritma kombinasional untuk optimasi yang dapat digunakan untuk menemukan solusi optimal dari permasalahan penugasan person. Terdapat dua macam interpretasi dari algoritma ini, yaitu dengan matriks dan graf *bipartite*.

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & \dots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \dots & c_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \dots & c_{nn} \end{pmatrix}$$

Permasalahan penugasan person banyak ditemui dalam kehidupan sehari-hari. Misalkan pada matriks di atas, C_{ij} menyatakan upah yang harus diberikan kepada pekerja ke- i untuk pekerjaan ke- j . Sebuah perusahaan ingin memperkerjakan n orang pekerja untuk menyelesaikan n buah pekerjaan dengan jumlah upah yang minimal. Untuk itu, perusahaan tersebut harus memilih tepat 1 elemen dari tiap baris dan tiap kolom, di mana jumlah keseluruhan dari elemen-elemen yang dipilih seminimal mungkin. Apabila matriks yang terbentuk bukanlah matriks bujur sangkar, yaitu jumlah pekerja \neq

jumlah pekerjaan, maka harus dibentuk sebuah matriks perluasan dengan cara menambah baris atau kolom bernilai 0.

Algoritma Hungarian dengan representasi matriks adalah sebagai berikut

1. Cari elemen tiap baris dalam matriks dengan nilai minimum. Kurangkan semua elemen dalam baris tersebut dengan elemen baris bernilai minimum.
2. Cari elemen tiap kolom dalam matriks dengan nilai minimum. Kurangkan semua elemen dalam baris tersebut dengan elemen kolom bernilai minimum.
3. Buat garis yang melingkupi semua elemen bernilai nol.
4. Lakukan pengecekan :
 - a. Jika jumlah garis = n , maka pilih kombinasi dari elemen-elemen matriks, dimana jumlah kombinasi tersebut = 0.
 - b. Jika jumlah garis < n , lakukan langkah ke-5.
5. Cari elemen dengan nilai minimum yang tidak terlingkupi oleh garis. Kurangkan elemen yang bernilai minimum dengan elemen-elemen yang tidak terlingkupi garis lainnya. Tambahkan elemen yang bernilai minimum dengan elemen-elemen yang terlingkupi garis horisontal dan garis vertikal. Apabila hasil pengurangan memberikan hasil negatif, maka elemen dengan nilai minimum tidak perlu dikurangkan. Kembali ke langkah ke-3.

Catatan :

Apabila persoalan yang dihadapi adalah persoalan maksimasi, kalikan matriks **C** dengan (-1).

Contoh:

Sebuah perusahaan konstruksi bangunan memiliki empat buah bulldoser. Keempat buah bulldoser itu memiliki jarak yang berbeda-beda terhadap lokasi pembangunan seperti terlihat dalam tabel berikut :

		Jarak ke Lokasi Pembangunan (km)			
		1	2	3	4
Bulldoser	1	90	75	75	80
	2	35	85	55	65
	3	125	95	90	105
	4	45	110	95	115

Tempatkan masing-masing bulldoser pada sebuah lokasi pembangunan dimana jumlah keseluruhan jarak ke lokasi pembangunan dibuat seminimal mungkin.

1. Bentuk matriks berukuran 4 x 4, dan kemudian cari elemen dengan nilai minimum pada tiap baris.

90	75	75	80
35	85	55	65
125	95	90	105
45	110	95	115

Kurangkan elemen dengan nilai minimum pada tiap baris dengan semua elemen pada baris tersebut.

15	0	0	5
0	50	20	30
35	5	0	15
0	65	50	70

2. Cari elemen dengan nilai minimum pada tiap kolom.

15	0	0	5
0	50	20	30
35	5	0	15
0	65	50	70

Kurangkan elemen dengan nilai minimum pada tiap kolom dengan semua elemen pada kolom tersebut.

15	0	0	0
0	50	20	25
35	5	0	10
0	65	50	65

3. Buat garis yang melingkupi semua elemen bernilai nol.

15	0	0	0
0	50	20	25
35	5	0	10
0	65	50	65

4. Lakukan pengecekan :

(jumlah garis =3) < (n = 4), maka, lanjutkan ke langkah ke-5.

5. Cari elemen dengan nilai minimum yang tidak terlingkupi oleh garis. Dalam matriks ini, elemen tersebut adalah elemen (2,3) dengan nilai 20.

15	0	0	0
0	50	20	25
35	5	0	10
0	65	50	65

Kurangkan elemen dengan nilai minimum yang tidak terlingkupi garis tersebut dengan elemen-elemen yang tidak terlingkupi garis lainnya. Tambahkan elemen bernilai minimum dengan elemen-elemen yang terlingkupi garis vertikal dan horisontal (yaitu elemen (1,1) dan (3,1)).

35	0	0	0
0	30	0	5
55	5	0	10
0	45	30	45

Kembali ke langkah ke-3.

35	0	0	0
0	30	0	5
55	5	0	10
0	45	30	45

Lakukan pengecekan :
(jumlah garis = 3) < (n = 4)

Lakukan kembali langkah ke-5 :

Cari elemen dengan nilai minimum yang tidak terlingkupi oleh garis. Dalam matriks ini, elemen tersebut adalah (2,4) dengan nilai 5.

35	0	0	0
0	30	0	5
55	5	0	10
0	45	30	45

Kurangkan elemen dengan nilai minimum dengan elemen-elemen yang tidak terlingkupi garis lainnya. Tambahkan elemen dengan nilai minimum dengan elemen-elemen yang terlingkupi garis vertikal dan garis horizontal (yaitu elemen (1,1) dan (1,3)).

40	0	5	0
0	25	0	0
55	0	0	5
0	40	30	40

Kembali ke langkah ke-3.

40	0	5	0
0	25	0	0
55	0	0	5
0	40	30	40

Lakukan pengecekan :
(jumlah garis = 4) = (n = 4)

40	0	5	0
0	25	0	0
55	0	0	5
0	40	30	40

Posisi dari elemen-elemen yang bernilai 0 adalah nilai optimum yang dipilih. Pada kasus ini, didapatkan lebih dari 1 kemungkinan yang menghasilkan jumlah nilai optimal yang sama. Salah satu kemungkinan adalah

Bulldoser 1 : Lokasi pembangunan 4 (80 km)

Bulldoser 2 : Lokasi pembangunan 3 (55 km)

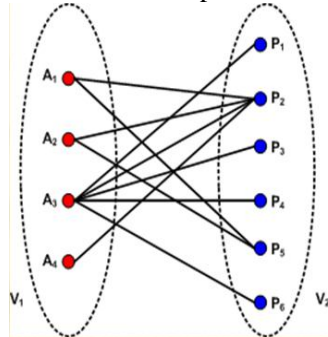
Bulldoser 3 : Lokasi pembangunan 2 (95 km)

Bulldoser 4 : Lokasi pembangunan 1 (45 km)

Jumlah jarak = (80 + 55 + 95 + 45) km
= 275 km

5.7. Latihan Soal

1. Suatu organisasi memiliki 6 komite yang rutin mengadakan rapat pada tiap akhir bulan dimana jadwal rapat per komite adalah sehari dalam sebulan. Komite tersebut adalah: K1 beranggotakan Agus, Budi, Zaenal; K2 beranggotakan Budi, Lilis, Ratih; K3 beranggotakan Agus, Ratih, Zaenal; K4 beranggotakan Lilis, Ratih, Zaenal; K5 beranggotakan Agus, Budi; serta K6 beranggotakan Budi, Ratih, Zaenal. Jika jumlah ruang rapat jumlahnya tidak dibatasi:
 - a. Berapa jumlah hari yang paling sedikit dibutuhkan untuk menjadwalkan rapat untuk seluruh komisi?
 - b. Komite mana yang dapat menyelenggarakan rapat dalam hari yang sama?
2. Terdapat 4 pelamar untuk 6 lowongan pekerjaan yang tersedia. Kualifikasi pelamar terhadap jenis pekerjaan yang ada dapat digambarkan melalui pemetaan pada graf di bawah ini. Apakah mungkin menerima semua pelamar untuk mengisi lowongan yang ada?



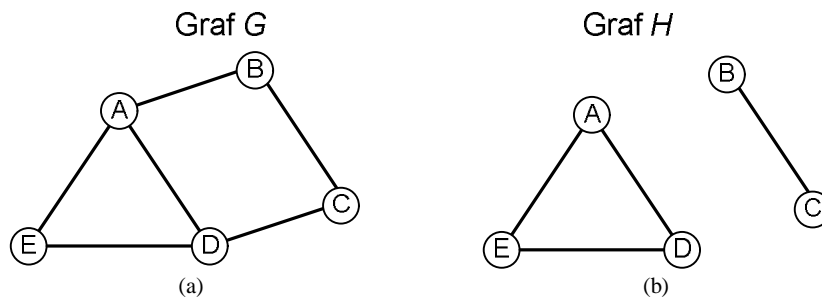
3. Terdapat 3 anak laki laki b_1 , b_2 , dan b_3 dan 4 anak perempuan g_1 , g_2 , g_3 , g_4 . Jika:
 - b_1 adalah sepupu dari g_1 , g_3 , dan g_4
 - b_2 adalah sepupu dari g_2 dan g_4
 - b_3 adalah sepupu dari g_2 dan g_3 ,
 dapatkah setiap anak laki laki menikahi anak perempuan yang merupakan sepupunya?

BAB 6. KONEKTIVITAS

6.1. Konsep Keterhubungan (*Connectivity*)

1. Keterhubungan pada Graf Tak-berarah

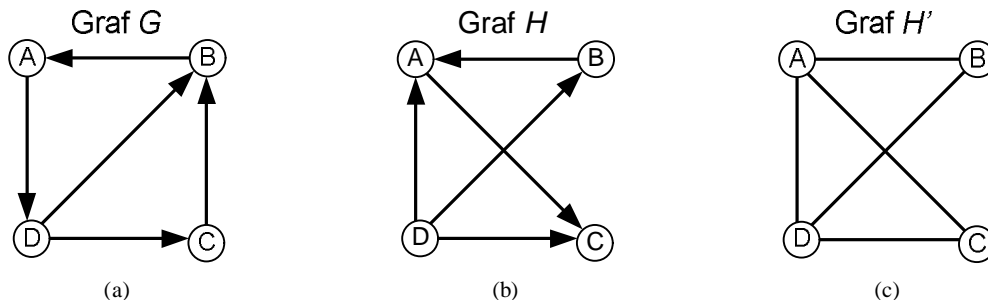
Sebuah graf tak-berarah disebut terhubung (*connected*) jika terdapat lintasan antar tiap pasangan simpul yang berbeda. Jika terdapat satu atau lebih pasangan yang tidak memiliki lintasan, maka graf tersebut disebut terputus (*disconnected*). Graf terhubung ditunjukkan pada graf G di Gambar 6.1(a). Sedangkan graf terputus ditunjukkan pada graf H di Gambar 6.1(b). Salah satu pasangan yang tidak memiliki lintasan pada graf H adalah simpul A dan B .



Gambar 6.1. (a) Graf terhubung G . (b) Graf terputus H

2. Keterhubungan pada Graf Berarah

Graf berarah memiliki penggolongan keterhubungan yang berbeda dengan graf tak-berarah. Terdapat dua jenis keterhubungan pada graf berarah, yaitu keterhubungan kuat (*strong connection*) dan keterhubungan lemah (*weak connection*).



Gambar 6.2. (a) Graf berarah G dengan keterhubungan kuat. (b) Graf berarah H dengan keterhubungan lemah. (c) Graf H' yang merupakan bentuk graf tak-berarah dari graf berarah H (*underlying*).

Graf berarah memiliki keterhubungan kuat jika terdapat lintasan untuk setiap simpul u dan v , dan sebaliknya terdapat pula lintasan untuk setiap simpul v dan u . Graf G pada Gambar 6.2 adalah graf berarah dengan keterhubungan kuat. Terdapat lintasan untuk setiap pasang simpul yang ada.

Graf berarah memiliki keterhubungan lemah jika *underlying*-nya merupakan graf terhubung. *Underlying* adalah bentuk graf tak-berarah dari sebuah graf berarah. Graf H' pada Gambar 6.2 merupakan *underlying* dari graf berarah H pada Gambar 6.2(c). Graf H pada Gambar 6.2(b) merupakan graf dengan koneksi lemah karena *underlying*-nya merupakan graf terhubung.

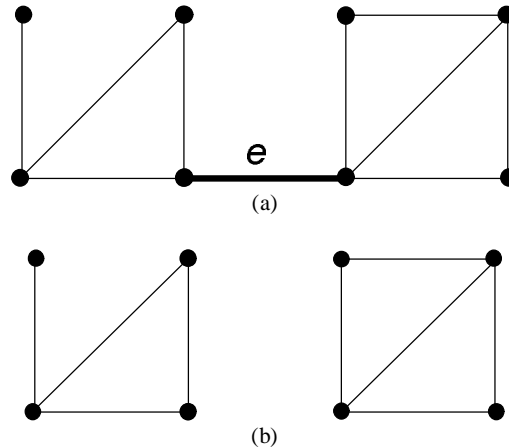
Hubungan yang menarik antara graf berarah dengan keterhubungan kuat dan lemah adalah setiap graf berarah dengan keterhubungan kuat pasti merupakan graf berarah yang memiliki keterhubungan lemah. Akan tetapi, graf berarah dengan keterhubungan lemah belum tentu merupakan graf berarah yang memiliki keterhubungan kuat.

6.2. Komponen Terhubung

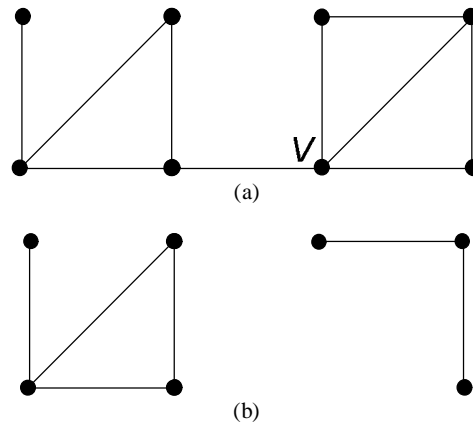
Sebuah komponen terhubung dari graf tak-berarah adalah sub-graf dimana semua simpul pada sub-graf tersebut terhubung. Sebagai contoh, graf G pada Gambar 6.1(a) memiliki jumlah komponen terhubung 1, sedangkan graf H pada Gambar 6.1(b) memiliki jumlah komponen terhubung 2.

6.3. Cut Edge (Bridge)

Sebuah busur e dihapus dari sebuah graph $G = (V, E)$ hasilnya adalah graph G' dengan komponen terhubung lebih banyak dari pada komponen terhubung graf G . Maka e disebut *cut edge*/ *bridge*. Pada Gambar 6.3, busur e merupakan *cut edge* / *bridge*.



Gambar 6.3. (a) Graf G dengan yang memiliki 1 komponen. (b) Graf G' yang merupakan graf G yang dihilangkan pada busur e sehingga memiliki 2 komponen.



Gambar 6.4 (a) Graf G dengan yang memiliki 1 komponen. (b) Graf G' yang merupakan graf G yang dihilangkan pada vertex V dan busur yang insiden kepadanya sehingga memiliki 2 komponen.

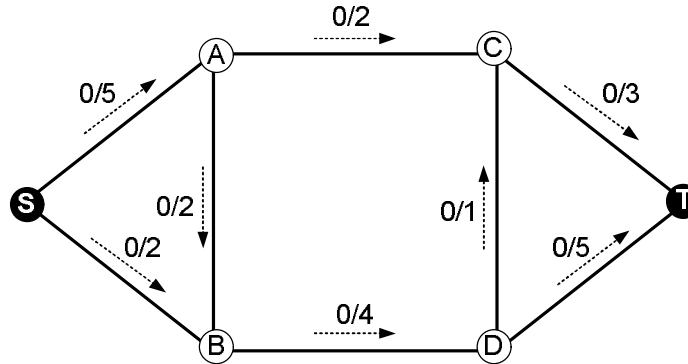
6.4. Cut Vertex

Jika sebuah simpul V dengan semua busur yang insiden pada V dihapus dari sebuah graf $G = (V, E)$ hasilnya adalah graph G' dengan komponen terhubung lebih banyak daripada komponen terhubung graf G , maka simpul V disebut *cut vertex*. Pada graf di Gambar 6.4, simpul V merupakan *cut vertex* karena jika V dihilangkan bersama dengan busur yang berinsiden dengannya maka komponen terhubung bertambah dari 1 menjadi 2.

6.5. Aliran pada Jaringan (*Flow in Network*)

Sebuah graf berarah G yang terhubung dapat disebut sebagai jaringan (*network*) jika :

1. Memiliki sebuah sumber (*source*), yaitu simpul yang hanya memiliki egde yang berarah keluar.
2. Memiliki sebuah muara (*sink*), yaitu simpul yang hanya memiliki egde yang berarah masuk.
3. Graf berarah G memiliki bobot yang disebut kapasitas (*capacity*) di setiap busur e yang menghubungkan simpul u dan v yang dinotasikan kedalam fungsi $c(e) = c(u, v)$.

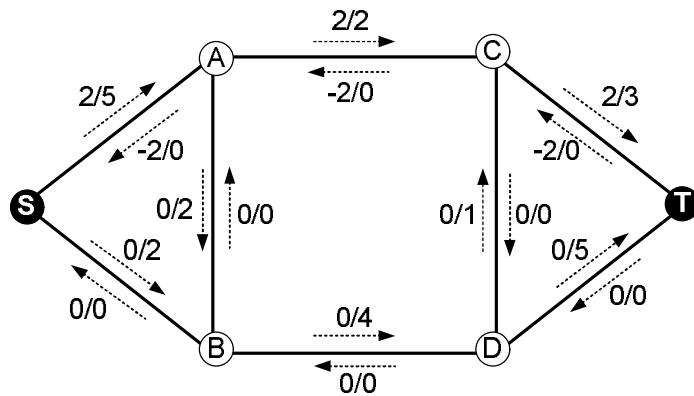


Gambar 6.5 Graf jaringan G yang memiliki kapasitas pada setiap busur

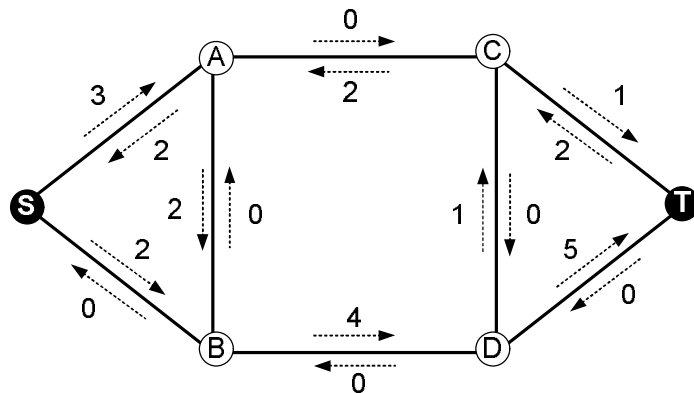
Graf G yang terdapat pada Gambar 6.5 memiliki simpul sumber di S dan muara di T . Salah satu contoh kapasitas adalah nilai 0/5 pada busur (S,A) . Nilai 0 berarti belum ada aliran pada jaringan dan nilai 5 adalah nilai kapasitas busur (S,A) , atau $c(S,A) = 5$. Pada kasus optimasi, nantinya akan dicari nilai aliran maksimum yang dapat dialirkan pada jaringan. Graf G yang terdapat pada Gambar 6.5 adalah graf baru yang belum dialiri sama sekali.

Prinsip-prinsip cara membuat aliran pada jaringan adalah sebagai berikut:

1. Setiap busur dapat mengalirkan/ menerima aliran (*flow*). Fungsi aliran pada sebuah busur (u,v) adalah $f(u,v)$ dimana $f(u,v) \leq c(u,v)$. Misalnya dialirkan aliran sebesar 2 pada lintasan $S-A-C-T$ pada graf G di Gambar 6.5, maka bentuk jaringan saat ini seperti ditunjukkan pada Gambar 6.6. Bobot busur (S,A) menjadi 2/5, artinya dari kapasitas 5 telah terisi aliran sebesar 2.
2. Berlaku aturan *skew symmetry*, yaitu $f(u,v) = -f(v,u)$. Artinya jika dialirkan aliran sebesar n pada busur (u,v) , maka harus dibuat aliran sebesar $-n$ dari busur (v,u) . Misalnya dibuat aliran sebesar 2 pada busur (S,A) , maka pada busur (A,S) terdapat aliran sebesar -2.
3. Terdapat kapasitas residu (*residual capacity*) $c_r(u,v)$ pada setiap busur (u,v) yang dirumuskan sebagai $c_r(u,v) = c(u,v) - f(u,v)$. Jaringan yang pada busurnya hanya dituliskan bobot kapasitas residunya disebut sebagai jaringan residu (*residual network*). Misalnya dialirkan aliran sebesar 2 pada lintasan $S-A-C-T$, jaringan residu dapat dilihat pada Gambar 6.7.
4. Terdapat istilah *augmented path*, yaitu jalur yang menghubungkan sumber S ke muara T pada jaringan residu. Jaringan dikatakan memiliki aliran maksimum apabila tidak terdapat lagi *augmented path* pada jaringan residu. Contoh salah satu *augmented path* pada jaringan residu di Gambar 6.7 adalah jalur $S-A-B-D-T$.



Gambar 6.6 Graf jaringan G pada Gambar 6.5 yang diberi aliran sebesar 2 pada jalur $S-A-C-T$



Gambar 6.7 Jaringan residu dari graf G di Gambar 6.6

6.7. Aliran Maksimum (*Maximum Flow*) pada Jaringan

Optimasi pada jaringan dilakukan dengan mencari nilai aliran maksimum yang dapat dialirkan pada jaringan. Algoritma *greedy* yang dilakukan untuk menentukan aliran maksimum pada jaringan adalah sebagai berikut:

1. Pilih salah satu *augmented path* yang yang dapat dapat dialiri aliran terbesar. Cara menentukan aliran terbesar pada *augmented path* $\{(S,A), (A,...), \dots, (... ,T)\}$ adalah $f_{\max} = \min (c_r(S,A), c_r(A,...), \dots, c_r(... ,T))$.
2. Perbarui jaringan residu setelah diberikan aliran pada salah satu *augmented path* terpilih.
3. Ulangi langkah 1 dan 2 hingga tidak ada lagi *augmented path*.

Contoh:

Carilah aliran maksimum dari graf jaringan G pada Gambar 6.5 menggunakan algoritma *greedy*!

Jawab:

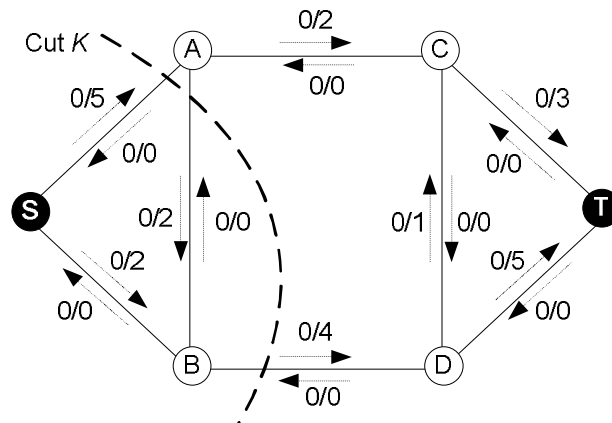
<i>i</i>	Jaringan	Jaringan Residu	Augmented path
1			S-A-C-T, $f_{\max} = 2$ (pilih) S-A-B-D-T, $f_{\max} = 2$ S-B-D-T, $f_{\max} = 2$ S-B-A-C-T, $f_{\max} = 2$ S-A-B-D-C-T, $f_{\max} = 1$ S-B-D-C-T, $f_{\max} = 1$
2			S-A-B-D-T, $f_{\max} = 2$ (pilih) S-B-D-T, $f_{\max} = 2$ S-A-B-D-C-T, $f_{\max} = 1$ S-B-D-C-T, $f_{\max} = 1$
3			S-B-D-T, $f_{\max} = 2$ (pilih) S-B-D-C-T, $f_{\max} = 1$
4			—

Aliran maksimum pada graf jaringan G adalah total aliran yang keluar dari simpul sumber S atau total aliran yang masuk ke muara T , yaitu 6.

6.8. Cut Set

Cut $K(P, P')$ pada graf G adalah garis yang memotong jaringan menjadi dua bagian. Dimana, P adalah himpunan simpul bagian pertama yang didalamnya terdapat simpul sumber S dan P' adalah komplemen P yaitu himpunan simpul bagian kedua yang didalamnya terdapat simpul muara T .

Cut $K(P, P')$ pada graf G di Gambar 6.8 memiliki busur $\{(S, A), (B, A), (B, D)\}$ dan membagi simpul menjadi $P = \{S, B\}$ dan $P' = \{A, C, D, T\}$. Nilai sebuah *cut* adalah nilai kapasitas pada busur yang dimiliki *cut* tersebut. Sehingga nilai $cut\ K(P, P') = c(S, A) + c(B, A), c(B, D) = 5 + (-2) + 4 = 7$. Sebagai catatan, $c(B, A)$ bernilai negatif karena arah aliran harus dari himpunan yang memuat simpul sumber S (yaitu P) menuju himpunan yang memuat simpul muara T ke (yaitu P'). Pada busur (B, A) arah aliran seharusnya dari simpul B ke A , namun diketahui aliran justru dari simpul A menuju B . Oleh karena itu, nilai kapasitas $c(B, A) = -c(A, B) = -2$.

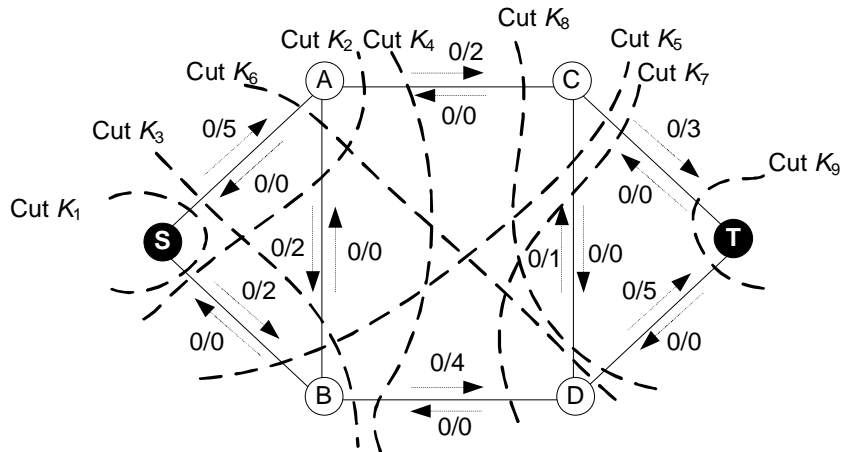


Gambar 6.8 Jaringan dengan cut K

6.9. Teorema Max-Flow Min-Cut

Teorema *max-flow min-cut* menyatakan bahwa pada sebarang jaringan nilai aliran maksimum selalu sama dengan nilai cut minimumnya.

Contoh penerapan teorema max-flow min-cut dapat diamati pada Gambar 6.9. Seluruh kemungkinan *cut* pada graf G di Gambar 6.5 ditampilkan pada Gambar 6.9, yaitu K_1, K_2, \dots, K_9 . Nilai masing-masing *cut* ditampilkan pada tabel di bawahnya. Dapat diamati bahwa nilai *cut* minimum adalah 6. Hal ini sama dengan nilai aliran maksimum pada graf yang sama yang ditemukan pada bagian 6.7, yaitu sebesar 6.



Gambar 6.9 Seluruh kemungkinan *cut* graf G pada Gambar 6.5.

<i>cut</i>	P	P'	nilai <i>cut</i>
K_1	$P = \{S\}$	$P' = \{A,B,C,D,T\}$	7
K_2	$P = \{S,A\}$	$P' = \{B,C,D,T\}$	6
K_3	$P = \{S,B\}$	$P' = \{A,C,D,T\}$	7
K_4	$P = \{S,A,B\}$	$P' = \{C,D,T\}$	6
K_5	$P = \{S,A,C\}$	$P' = \{B,D,T\}$	6
K_6	$P = \{S,B,D\}$	$P' = \{A,C,T\}$	8
K_7	$P = \{S,A,B,C\}$	$P' = \{D,T\}$	6
K_8	$P = \{S,A,B,D\}$	$P' = \{C,T\}$	8
K_9	$P = \{S,A,B,C,D\}$	$P' = \{T\}$	8

6.10. Problem

Maximum Bipartite Matching sebagai Permasalahan Aliran Maksimum

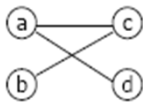
Salah satu implementasi dari aliran maksimum adalah untuk menentukan derajat kecocokan tertinggi, misalkan menentukan pasangan optimum dalam suatu jaringan. Aplikasi aliran maksimum dapat juga diterapkan dalam menentukan pasangan optimum pada graf *bipartite*. Permasalahan ini disebut dengan Maximum *Bipartite Matching* (MBM).

Langkah-langkah untuk menentukan MBM adalah sebagai berikut:

1. Tambahkan simpul sumber dan busur-busur berarah yang mengarah ke setiap simpul dalam V_1 dengan kapasitas 1.
2. Tambahkan simpul muara dan busur-busur berarah yang mengarah dari setiap simpul dalam V_2 dengan kapasitas 1
3. Setiap busur dalam G adalah busur dengan kapasitas 1 dan berarah dengan arah dari simpul yang berada di V_1 ke simpul yang berada di V_2

Contoh:

Selesaikan MBM dari graf di bawah ini!



Jawab:

Sebagai jaringan kapasitas, bobot tidak ditampilkan nilai adalah 1



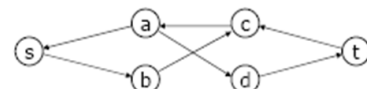
Kapasitas c dalam representasi matriks

c	s	a	b	c	d	t
s		1	1	0	0	0
a	0		0	1	1	0
b	0	0		1	0	0
c	0	0	0		0	1
d	0	0	0	0		1
t	0	0	0	0	0	

Jika jalur pertama yang dipilih adalah s-a-c-t, maka matriks flow f , kapasitas residu cr , dan bentuk jaringan adalah:

f	s	a	b	c	d	t
s		1	0	0	0	0
a	-1		0	1	0	0
b	0	0		0	0	0
c	0	-1	0		0	1
d	0	0	0	0		0
t	0	0	0	-1	0	

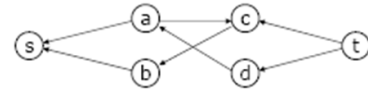
cr	s	a	b	c	d	t
s		0	1	0	0	0
a	1		0	0	1	0
b	0	0		1	0	0
c	0	1	0		0	0
d	0	0	0	0		1
t	0	0	0	1	0	



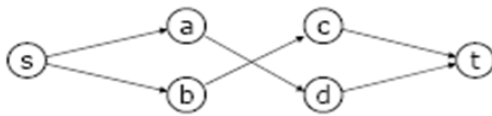
Jika jalur kedua yang dipilih adalah s-b-c-a-d-t, maka matriks flow f , kapasitas residu cr , dan bentuk jaringan adalah:

f	s	a	b	c	d	t
s		1	1	0	0	0
a	-1		0	0	1	0
b	-1	0		1	0	0
c	0	0	-1		0	1
d	0	-1	0	0		1
t	0	0	0	-1	-1	

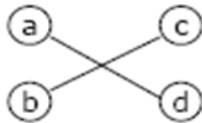
cr	s	a	b	c	d	t
s		0	0	0	0	0
a	1		0	1	0	0
b	1	0		0	0	0
c	0	0	1		0	0
d	0	1	0	0		0
t	0	0	0	1	1	



Tidak ada lagi aliran dari sumber s ke muara t (*augmented path*), sehingga graf dengan aliran maksimum berbentuk:

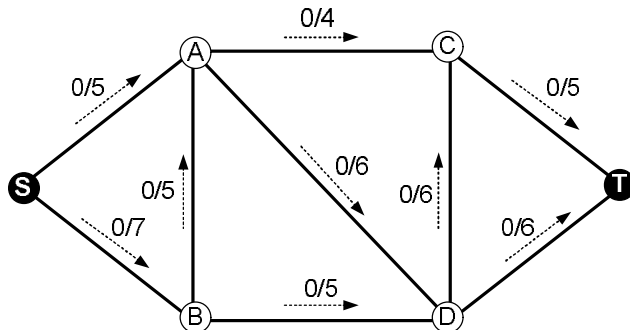


Jadi, pencocokan maksimum adalah a-d dan b-c

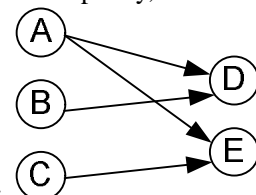


6.11. Latihan Soal

1. Carilah nilai aliran maksimum dari jaringan di bawah ini. Simpul S adalah sumber dan simpul T adalah muara.



2. Carilah nilai cut minimum dari jaringan di atas!
3. Tentukan solusi Maximum Bipartite Matching dengan menunjukkan matriks Capacity, Flow dan



Residual Capacity masing-masing proses pada graf bipartite di bawah ini:

4. Pelatih Kesebelasan "Red Allstar" memiliki masing-masing sebuah tempat kosong pada posisi striker dan pemain tengah. Kandidat pemain yang dapat mengisi posisi tersebut adalah Kaka, Lampard, dan Messi. Kaka dan Lampard mampu menjadi pemain tengah, sedangkan Messi mampu dipasang sebagai striker maupun pemain tengah.

- a. Buatlah graf bipartite dari pemain dan posisi yang digambarkan pada ilustrasi di atas.
- b. Tentukan solusi Maximum Bipartite Matching dengan menunjukkan matriks kapasitas, aliran dan kapasitas residu pada masing-masing proses.

BAB 7. TEORI BAHASA & OPERASI MATEMATIS PENDUKUNGNYA

7.1. Pentingnya Belajar Teori Komputasi

Komputer, disadari atau tidak, telah banyak melakukan revolusi pada kehidupan kita. Komputer telah banyak mengubah cara kita menjalani kehidupan sehari-hari. Cara kita mengeksplorasi dan memanfaatkan ilmu pengetahuan. Cara kita memanjakan diri. Dan cara kita menjalankan bisnis serta organisasi.

Teori yang akan kami bahas dalam bagian buku ini merupakan dasar dari semua aplikasi di atas. Karena seperti yang kita ketahui bersama bahwa jika kita menyebut kata komputer, maka selain perangkat keras, ada pula perangkat lunak penunjangnya. Dan semua bantuan perangkat lunak pendukung komputer dibangun dan dikembangkan dari bahasa pemrograman. Bahasa pemrograman sendiri telah mengalami perkembangan yang cukup signifikan dari waktu ke waktu.

```
ENTRY          SXA    4,RETURN
               LDQ    X
               FMP    A
               FAD    B
               XCA
               FMP    X
               FAD    C
               STO    RESULT
RETURN         TRA    0

A              BSS    1
B              BSS    1
C              BSS    1
X              BSS    1
TEMP           BSS    1
STORE          BSS    1
               END
```

Sebagai contoh, potongan program di atas ditulis untuk mesin IBM 7090. Rutin di atas adalah untuk mengkomputasi persamaan kuadrat sederhana $ax^2 + bx + c$. Tetapi meskipun sederhana, programmer masa kini tetap kesulitan membaca atau memahami program di atas. Oleh karenanya dalam perjalanan waktu, bahasa pemrograman selalu dikembangkan dan disempurnakan sedemikian hingga semakin mendekati bahasa manusia. Programmer masa lalu mungkin juga tidak pernah membayangkan bagaimana bentuk bahasa pemrograman masa kini (seperti halnya kita tidak mampu membayangkan bagaimana rupa bahasa pemrograman di masa yang akan datang). Dan beberapa hal yang dapat kita catat dari peran bahasa pemrograman terhadap kemajuan teori komputasi antara lain adalah:

- Bahasa (seperti yang akan banyak dibahas dalam bagian ini), telah memungkinkan komunikasi antara mesin-mesin dan manusia-mesin. Tanpa bahasa pemrograman, tidak satupun aplikasi dari komputasi akan dapat eksis seperti hari ini.
- Desain dan implementasi dari bahasa pemrograman modern memiliki ketergantungan yang sangat tinggi terhadap perkembangan teori bahasa bebas-konteks (context-free language). Dimana context-free grammar banyak digunakan dan mendasari syntax bahasa pemrograman dan grammar itulah yang menjadi basis dari teknik parsing yang digunakan oleh semua kompilator (compilers).
- Manusia menggunakan bahasa natural untuk berkomunikasi dengan sesama. Dan merupakan keinginan kita bersama agar bahasa natural tersebut juga dapat digunakan mengakses semua kebutuhan kita melalui komputer.
- Perilaku dari banyak sistem di dunia ini, seperti vending machine, communication protocols, perangkat keamanan gedung, dan lain sebagainya umumnya dapat digambarkan melalui finite automata, yang juga merupakan alat pendefinisi bahasa.
- Banyak video game yang beredar di masyarakat merupakan bentuk finite automata.

- DNA adalah bahasa kehidupan. Molekul-molekul DNA (sebagaimana halnya protein) merupakan rangkaian string yang tersusun dari simbol-simbol alphabet. Sehingga komputasi biologis juga memerlukan alat bantu komputasi (tools) yang sama dengan bidang lain. Dan oleh karenanya mereka juga akan sangat bergantung dengan teknik-teknik yang dikembangkan untuk finite automata dan context-free grammar.

Dalam bagian ini kita akan mempelajari mengenai bagaimana sebuah bahasa pemrograman dibuat. Bukan tentang bagaimana membuat program menggunakan bahasa pemrograman tertentu. Tetapi lebih pada memahami bentuk dan bagaimana grammar dilahirkan, serta bagaimana grammar tersebut dapat memeriksa kesesuaian antara rangkaian command yang ditulis oleh programmer dengan syntax bahasa pemrograman yang bersangkutan.

7.2. Terminologi Bahasa & Operasi Matematis yang Mendukungnya

Bahasa merupakan media komunikasi 2 pihak dengan menggunakan sekumpulan simbol dan dikombinasikan menurut aturan sintaksis tertentu (atau disebut grammar). Setiap bahasa apapun di dunia ini pasti memiliki himpunan sentence (kalimat) yang unik. Berbeda antara satu bahasa dengan bahasa yang lain. Kalimat sendiri tersusun dari rangkaian string (kata). Dimana setiap kata juga tersusun dari rangkaian karakter (bisa berupa alphabet/huruf, digit/angka, atau simbol). Jadi pada dasarnya sebuah bahasa tercipta dari manipulasi karakter. Dan oleh karenanya salah satu terminologi penting didalam memahami teori bahasa adalah pemahaman terhadap manipulasi string dan karakter.

Jika sintaks bahasa berfungsi memeriksa kebenaran rangkaian string pembentuk sebuah kalimat, maka semantik bahasa bertugas memastikan apakah kalimat yang tersusun secara benar (menurut sintaks) dapat memiliki arti/makna yang juga benar.

Jika kita ingin mendiskusikan bahasa, maka tidak akan terlepas dari cara menyatakan himpunan. Karena bahasa identik dengan himpunan-himpunan yang mendukung/membentuk bahasa tersebut. Dan seperti kita ketahui cara termudah untuk menyatakan himpunan adalah dengan cara menyebutkan semua anggota himpunan tersebut satu per satu.

Namun menyebutkan anggota himpunan satu per satu ternyata bukan merupakan keputusan yang tepat. Karena, walaupun jumlah huruf dan angka adalah berhingga (ada 26 huruf dan 10 angka), jumlah kalimat dan string yang merupakan hasil kombinasi dari huruf dan/atau angka dapat tak berhingga jumlahnya! Sehingga menyatakan sebuah bahasa akan lebih efisien jika tidak menyebut anggota bahasa tersebut (kalimat atau kata) secara listing (satu per satu), akan tetapi dapat lebih formal. Yaitu cukup dengan mengemukakan syarat-syarat atau ciri-ciri yang dimiliki bahasa/himpunan tersebut. Dan untuk selanjutnya pendekatan ini kita sebut dengan set theoretic notation.

Sebagai contoh misalkan terdapat sebuah himpunan alphabet $\Sigma = \{x\}$.

Jika dari himpunan Σ di atas kita akan mencoba mendefinisikan sebuah bahasa sederhana, L_1 . dimana L_1 adalah sebuah bahasa yang memiliki string hanya dari manipulasi karakter x saja. Maka L_1 dapat kita nyatakan sebagai:

$$L_1 = \{ x, xx, xxx, xxxx, \dots \}$$

Atau secara lebih formal L_1 dapat kita nyatakan sebagai:

$$L_1 = \{ x^n, \text{ untuk } n = 1, 2, 3, \dots \}$$

Atau kita mencoba mendefinisikan bahasa L_2 yang memiliki anggota berupa string dari hasil kombinasi ganjil anggota himpunan Σ . Dan L_2 dapat kita tulis:

$$L_2 = \{ x, xxx, xxxxx, \dots \}$$

secara formal L_2 dapat kita tulis dalam bentuk:

$$L_2 = \{ x^n, \text{ untuk } n = 1, 3, 5, \dots \}$$

Jika bahasa dapat dipandang sebagai manifestasi himpunan, maka tentunya ada operasi-operasi matematis yang berlaku pada himpunan juga dapat berlaku pada bahasa. Operasi-operasi tersebut antara lain adalah:

Union antara 2 himpunan L dan himpunan M , atau ditulis $L \cup M$. Dimana operasi ini akan menghasilkan himpunan baru yang memiliki anggota dari semua anggota himpunan L dan semua anggota himpunan M . Sebagai contoh, misalkan terdapat sebuah himpunan string $L = \{ a, aa, aaa \}$ dan $M = \{ bb, bbb \}$. Dan jika diberikan operasi union kepada kedua himpunan tersebut, maka $L \cup M = \{ a, aa, aaa, bb, bbb \}$.

Concatenation (penggabungan) antara 2 himpunan atau string L dan M . Ditulis LM . Hasil dari penggabungan ini akan membentuk string atau himpunan baru yang merupakan penggabungan 2 string asal atau penggabungan 2 string yang masing berasal dari himpunan-himpunan L dan M . Sebagai contoh jika kita menggunakan himpunan L dan M seperti terdefinisi di atas, maka hasil concatenation $LM = \{ abb, abbb, aabb, aabbb, aaabb, aaabbb \}$.

Closure dari sebuah string atau himpunan L . Atau ditulis L^* untuk Kleene closure dan L^+ untuk positive closure. Esensi dari operasi closure ini adalah pengulangan penulisan string dan menggabungkan (concatenate) dengan string yang tertulis lebih dahulu. Perbedaan antara Kleene closure dengan positive closure adalah jika pada Kleene closure pengulangan boleh tidak dilakukan atau bahkan penulisan/pemilihan sebuah string boleh tidak dilakukan sama sekali. Sedangkan pada positive closure, pengulangan boleh tidak dilakukan, namun penulisan/pemilihan sebuah string harus tetap dilakukan. Sehingga dalam operasi positive closure tidak akan dihasilkan empty/null string. Sebagai contoh, misalkan terdapat sebuah himpunan alphabet $\Sigma = \{ 0, 1 \}$. Yang mana pada himpunan tersebut akan diberlakukan operasi Kleene closure. Maka akan dihasilkan $\Sigma^* = \{ \lambda, 0, 1, 00, 01, 10, 11, 000, \dots \}$.

Sementara pada himpunan alphabet $\Sigma = \{ x \}$ akan diberikan operasi positive closure, yang akan menghasilkan himpunan $\Sigma^+ = \{ x, xx, xxx, \dots \}$.

Selain operasi mayor di atas, terdapat pula beberapa sifat/operasi minor yang berlaku pada bahasa, yaitu antara lain:

Reverse of String x , atau ditulis $\text{rev}(x)$. Operasi ini menghasilkan sebuah string baru yang merupakan kebalikan penulisan dari string x . Sebagai contoh terdapat sebuah string $aabbb$. Jika dilakukan pembalikan maka akan dihasilkan $\text{rev}(aabbb) = bbbaa$.

Length of string x , atau ditulis $\text{length}(x)$. operasi ini menghasilkan nilai integer positif yang merepresentasikan jumlah karakter yang membentuk string tersebut. Sebagai contoh string $aabbb$ di atas memiliki panjang atau $\text{length}(aabbb) = 5$.

Dan yang terakhir adalah **Palindrome**. Yaitu sebuah sifat yang dimiliki oleh string-string tertentu. Dimana string tersebut memiliki keistimewaan berupa susunan karakter yang sama jika dibaca dari depan maupun dari belakang. Atau $x = \text{rev}(x)$. Seperti misalnya string aaa , aba , $abba$, dan lain sebagainya.

Sejauh ini kita sudah mengenal 2 cara untuk menyatakan sebuah himpunan, yaitu dengan cara *me-listing* dan menggunakan *set theoretic notation*. Selain kedua cara di atas, masih terdapat teknik lain untuk mendeklarasi himpunan. Yaitu dengan menggunakan apa yang disebut sebagai **pendefinisian secara rekursif**. Secara definitif, definisi rekursif adalah upaya untuk menyatakan sebuah himpunan melalui penerapan sebuah/sekelompok aturan secara berulang-ulang terhadap sebuah obyek dasar.

Atau **definisi rekursif** dapat dilakukan melalui kedua langkah berikut:

1. Menentukan sebuah obyek dasar dari himpunan yang akan dibentuk;
2. Menetapkan aturan-aturan yang dapat digunakan untuk membantuk/membangkitkan obyek atau anggota lain dari himpunan tersebut.

Mungkin akan lebih mudah bagi kita untuk memahami konsep definisi rekursif tersebut jika melihat sebuah contoh berikut:

Misalkan kita akan membuat himpunan bilangan genap. Yang seperti kita ketahui bilangan genap antara lain adalah 2, 4, 6, 8, 10, dan seterusnya. Melalui definisi rekursif, maka kita tetapkan sebuah obyek awal/dasar dari himpunan bilangan genap tersebut, misalkan bilangan 2.

1. 2 adalah anggota himpunan bilangan genap.

Sekarang jika kita misalkan bilangan x adalah anggota bilangan genap, maka sebarang bilangan x jika ditambah 2 (yang pasti merupakan bilangan genap) tentu hasilnya adalah bilangan genap (yang otomatis akan menjadi anggota himpunan yang dimaksud). Sehingga

2. Jika x adalah anggota himpunan bilangan genap, maka $x+2$ adalah juga merupakan anggota himpunan tersebut.

Sekarang untuk membuktikan bahwa 14 adalah bilangan genap, maka kita memerlukan langkah-langkah seperti berikut:

- Melalui aturan 1: 2 adalah anggota pertama bilangan genap
- Melalui aturan 2: $2 + 2 = 4$ adalah bilangan genap
- Melalui aturan 2: $4 + 2 = 6$ adalah bilangan genap
- Melalui aturan 2: $6 + 2 = 8$ adalah bilangan genap
- Melalui aturan 2: $8 + 2 = 10$ adalah bilangan genap
- Melalui aturan 2: $10 + 2 = 12$ adalah bilangan genap
- Melalui aturan 2: $12 + 2 = 14$ adalah bilangan genap

Selain alternatif-1 di atas, pendefinisian himpunan bilangan genap dapat pula dinyatakan seperti alternatif-2 berikut:

- Aturan 1: 2 adalah bilangan genap;
- Aturan 2: jika x dan y adalah anggota bilangan genap, maka $x + y$ juga merupakan anggota bilangan genap.

Dengan definisi baru di atas, pembuktian bahwa 14 adalah bilangan genap ternyata memerlukan lebih sedikit langkah:

- Melalui aturan 1: 2 adalah anggota pertama bilangan genap
- Melalui aturan 2: $x=2$ dan $y=2 \rightarrow x+y=4$ adalah bilangan genap
- Melalui aturan 2: $x=2$ dan $y=4 \rightarrow x+y=6$ adalah bilangan genap
- Melalui aturan 2: $x=4$ dan $y=6 \rightarrow x+y=10$ adalah bilangan genap
- Melalui aturan 2: $x=4$ dan $y=10 \rightarrow x+y=14$ adalah bilangan genap

Namun pemahaman terhadap konsep definisi rekursif seringkali tidak semudah yang kita bayangkan di awal. Oleh karenanya akan lebih bagus jika kita mau memahami contoh-contoh lain seperti berikut:

Contoh :

Misalkan terdapat sebuah bahasa $L_1 = \{ \lambda, x, xx, xxx, \dots \}$. Melalui Definisi Rekursif, bahasa L_1 tersebut dapat pula kita nyatakan melalui cara berikut :

1. λ adalah anggota L_1
2. Jika Q adalah sebarang string di L_1 , maka xQ juga merupakan string dalam L_1

Contoh :

Misalkan terdapat sebuah bahasa $L_2 = \{ x, xxx, xxxxx, \dots \}$. Melalui Definisi Rekursif, bahasa L_2 tersebut dapat kita nyatakan melalui cara berikut :

1. x adalah anggota L_2
2. Jika Q adalah sebarang string di L_2 , maka xxQ juga merupakan string dalam L_2

Contoh:

Berikut adalah definisi rekursif untuk pendefinisian bilangan bulat/integer positif:

1. 1 adalah bilangan integer
2. Jika x adalah bilangan integer, maka demikian halnya dengan $x + 1$.

Contoh:

Berikut adalah definisi rekursif yang diupayakan untuk mendefinisikan bilangan riil positif:

1. x adalah bilangan positif
2. Jika x dan y adalah bilangan positif, maka demikian halnya dengan $x + y$ dan xy .

Tetapi definisi di atas ternyata masih menyimpan masalah mendasar. Yaitu tidak adanya bilangan riil positif terkecil yang dapat digunakan untuk membangun himpunan yang dimaksud.

Pun jika aturan 1 diubah menjadi:

1. Jika x adalah integer, “.” adalah penanda desimal, dan y adalah sebarang rangkaian digit berhingga, maka $x.y$ ada dalam himpunan bilangan riil positif.

Modifikasi aturan 1 di atas memunculkan 2 masalah baru. Yaitu, aturan tersebut tetap tidak dapat membangkitkan semua bilangan riil positif (seperti bilangan π yang memiliki panjang tak berhingga). Dan definisi tersebut juga tidak bersifat rekursif karena aturan kedua tidak menggunakan aturan pertama.

Demikian pula jika kita memodifikasi seluruh aturan menjadi:

1. 1 adalah bilangan riil positif
2. Jika x dan y adalah bilangan riil positif, maka demikian halnya dengan $x + y$, $x * y$, dan x/y .

Contoh:

Misalkan kita ingin mendefinisikan himpunan POLINOMIAL melalui definisi rekursif seperti berikut:

Aturan 1: sebarang bilangan ada dalam POLINOMIAL

Aturan 2: sebuah variabel x ada dalam POLINOMIAL

Aturan 3: jika p dan q ada dalam POLINOMIAL, maka demikian pula dengan $p + q$, $p - q$, dan pq

Simbol pq walaupun mirip dengan operasi concatenation, namun dalam POLINOMIAL diartikan sebagai multiplikasi/perkalian.

Untuk menguji apakah ekspresi $3x^2 + 7x - 9$ adalah anggota POLINOMIAL, maka diperlukan langkah-langkah pembuktian seperti berikut:

Melalui aturan 1: 3 adalah POLINOMIAL

Melalui aturan 2: x adalah POLINOMIAL

Melalui aturan 3: $(3)(x)$ adalah POLINOMIAL \rightarrow ditulis $3x$

Melalui aturan 3: $(3x)(x)$ adalah POLINOMIAL \rightarrow ditulis $3x^2$

Melalui aturan 1: 7 adalah POLINOMIAL

Melalui aturan 3: $(7)(x)$ adalah POLINOMIAL \rightarrow ditulis $7x$

Melalui aturan 3: $3x^2 + 7x$ adalah POLINOMIAL

Melalui aturan 1: -9 adalah POLINOMIAL

Melalui aturan 3: $3x^2 + 7x + (-9)$ adalah POLINOMIAL \rightarrow ditulis $3x^2 + 7x - 9$

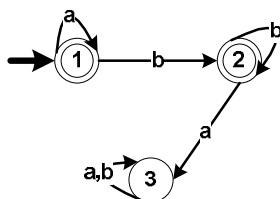
7.3. Mesin-mesin yang Berasosiasi dengan Klas Bahasa

Ke depan kita akan belajar mendefinisikan beberapa mesin yang umum digunakan untuk memodelkan proses komputasional. Model pertama akan terlihat sangat sederhana: umumnya program yang ditulis akan relatif mudah dipahami, dan program tersebut akan berjalan dalam koridor waktu yang linier. Sedangkan model kedua akan terasa lebih powerful, tetapi bagaimanapun tetap memiliki banyak keterbatasan. Sementara model ketiga diyakini lebih baik dari kedua model sebelumnya, karena mampu menjelaskan semua level pekerjaan yang dapat dikomputasi oleh sebarang komputer.

Semua model di atas memungkinkan kita untuk menulis program yang tugasnya adalah menerima sebuah bahasa L .

Bahasa Regular (Regular Languages)

Model pertama yang akan kita introduksi adalah finite automata (FA). Gambar 7.1 di bawah adalah contoh FA sederhana yang mampu menerima string (kata) yang terdiri dari alphabet a dan b, dimana semua a ada di depan b.



Gambar 7.1 sebuah FA sederhana

Sebagai input bagi mesin FA umumnya berupa string, yang kemudian akan diterima/diproses karakter per karakter, dari kiri ke kanan. Sebuah FA akan memiliki start state (yang ditunjukkan oleh lingkaran yang ditunjuk oleh panah tak berlabel), dan nol atau beberapa accepting state (state penerimaan) yang ditunjukkan oleh lingkaran dengan garis lingkaran ganda.

Setiap proses dalam FA selalu diawali dari start state. Dan untuk setiap karakter yang terbaca akan mengakibatkan pergerakan dalam FA (ada kesesuaian antara karakter yang dibaca dengan posisi panah berlabel pada mesin). Jika sebuah string s mengakibatkan terhentinya proses pergerakan pada mesin M setelah pembacaan karakter terakhir pada s , maka dikatakan M menerima s (atau disebut s adalah anggota dari bahasa yang didefinisikan oleh M).

Pada mesin di atas, tidak akan terjadi perpindahan state sepanjang pembacaan hanya melibatkan karakter input a. Baru jika terdapat input b maka akan terjadi pergerakan pada mesin dari state 1 menuju ke state 2. Dan akan terus berada di state 2 selama input karakter masih terus b. Dimana kedua state adalah accepting state. Mesin akan bergerak menuju dead-end state 3 jika input karakter berubah menjadi a lagi. Dan sekali proses berada di state 3 maka string apapun sudah pasti bukan merupakan anggota dari bahasa yang didefinisikan oleh mesin tersebut (karena kendali mesin tidak pernah lagi kembali ke state 1 atau 2 yang notabene merupakan accepting state). Sebagai contoh, mesin di atas dapat menerima string aab, aabbb, bb, dan lain-lain. Tetapi menolak string semacam ba, baab, dan lain sebagainya.

Atau mesin di atas disebut dapat menerima semua bentuk bahasa regular, seperti bilangan desimal yang syntactically well-formed, kombinasi rangkaian koin untuk membeli minuman pada mesin penjual, dan lain sebagainya.

Bahasa Bebas Konteks (Context-Free Language)

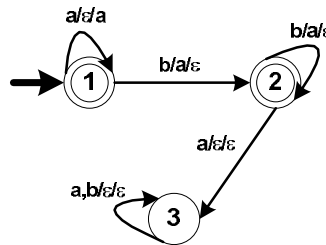
Tetapi ada pula bahasa yang tidak dapat disebut bahasa regular. Seperti bahasa Bal, yang menggunakan tanda kurung berimbang antara kurung buka dan kurung tutup. $()()$ atau $(())$. Atau sebarang string dalam bahasa $A^nB^n = \{a^nb^n; n \geq 0\}$.

Pada bahasa-bahasa di atas, mesin bertipe regular akan mengalami kesulitan terutama dalam memeriksa keseimbangan jumlah. Seperti bagaimana mesin regular dapat memeriksa bahwa jumlah kurung buka telah sama dengan jumlah kurung tutup? Atau bagaimana mesin regular dapat menghitung jumlah karakter a sedemikian hingga nantinya dapat dibandingkan dengan jumlah karakter b? Padahal seperti kita ketahui mesin FA tidak dilengkapi dengan memori. FA hanya dapat membaca dan tidak menyimpan informasi apapun tentang apa yang telah dibacanya. Pada bab selanjutnya akan kita buktikan bahwa memang mustahil membuat FA yang dapat menerima bahasa A^nB^n .

Tetapi bahasa semacam B^n dan A^nB^n ini sebenarnya merupakan bahasa yang sangat penting, khususnya bagi dunia pemrograman. Karena umumnya hampir semua bahasa pemrograman, bahasa markup maupun bahasa query mengijinkan penggunaan tanda kurung. Oleh karenanya harus ada bagian dari kompilator bahasa pemrograman tersebut yang bertugas memeriksa kesesuaian jumlah tanda kurung buka maupun tutup.

Jika yang kita butuhkan adalah sebuah mekanisme penyimpanan untuk menyimpan apa yang baru dibaca oleh mesin FA, maka dapat menambahkan sebuah single stack sederhana ke dalam FA tersebut. Modifikasi ini diperbolehkan, dan untuk selanjutnya mesin FA dengan tambahan stack ini kita sebut sebagai PDA (pushdown automata).

Dengan adanya mesin semacam PDA ini kita menjadi mudah untuk membangun mesin yang dapat menerima bahasa A^nB^n . Konsepnya sebenarnya sangat sederhana, yaitu setiap kali terbaca karakter a, maka PDA akan menyimpan karakter a tersebut ke dalam stack. Dan kemudian jika karakter b yang terbaca, maka mesin akan mem-pop (menghapus/mengeluarkan) karakter a dari dalam stack. Jika string yang dikenali memang berbentuk A^nB^n , maka seharusnya stack akan kembali dalam keadaan kosong pada saat seluruh karakter pada string tersebut selesai terbaca, dan penelusuran akan berhenti pada accepting state.



Gambar 7.2 sebuah PDA sederhana yang mendefinisikan bahasa A^nB^n

Label setiap panah pada PDA di atas berbentuk $x/y/z$ yang berarti “jika inputnya karakter x, maka yang ter-pop adalah karakter y, dan mem-push karakter z”. Dengan konsepsi yang sama, maka kita seharusnya dengan mudah dapat membangun mesin-mesin untuk bahasa yang sejenis dengan A^nB^n . Seperti bahasa palindrome misalnya. Dimana setiap string pada bahasa palindrome dapat dibaca dari kiri-ke-kanan atau dari kanan-ke-kiri dengan hasil pembacaan yang sama.

Decidable dan Semidecidable Language

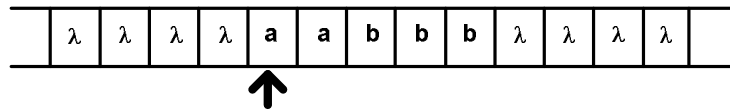
Walaupun PDA dapat dipandang cukup mumpuni, akan tetapi masih ada saja kekurangannya. Yaitu didalam menangani bahasa-bahasa yang bertipe lebih dari sekedar context-free. Semisal kalimat-kalimat dalam bahasa Indonesia atau Inggris dimana ada beberapa kata dalam sebuah kalimat yang muncul lebih dari satu kali. Atau dalam bentuk lebih sederhana, bahasa $A^nB^nC^n = \{ a^n b^n c^n; n \geq 0 \}$. Yaitu sebuah bahasa dengan string berbentuk gabungan dari karakter a, b dan c. Dimana semua karakter a ada di depan b, dan semua karakter b ada di depan c. Dan jumlah masing-masing karakter a, b, dan c adalah sama.

Jika kita mencoba membangun mesin PDA untuk bahasa $A^nB^nC^n$ ini, maka kita akan memanfaatkan stack untuk menampung a. Dan menghapusnya pada saat membaca b. Tetapi bagaimana dengan karakter c? Yang pasti, sesaat sebelum membaca c, kita telah kehilangan semua informasi mengenai a dan b. Karena jika keduanya memiliki jumlah yang sama, maka stack pasti akan kosong.

Padahal untuk menulis program yang dapat menerima bahasa $A^nB^nC^n$ bukanlah perkara sulit. Tetapi men-generate mesin yang dapat mengkomputasi semua jenis program yang dapat ditulis manusia, tentunya kita memerlukan model mesin dengan klas yang lebih tinggi dari FA maupun PDA. Dan untuk memenuhi kebutuhan ini, maka kita akan berkenalan dengan mesin jenis ketiga ini. Sebuah klas mesin yang tidak lagi menggunakan stack, tetapi memiliki tape. Sebuah tape yang dilengkapi dengan

head pembacaan atau penulisan (read/write head). Head ini berposisi di bawah tape dan dapat bergerak bebas ke kiri atau ke kanan. Dan mesin jenis ini kita sebut sebagai mesin Turing (Turing machine).

Dalam mesin Turing ini kita juga akan mengubah cara pemasukan input ke dalam mesin. Dari model streaming, satu per satu karakter seperti yang selama ini kita lakukan pada FA dan PDA. Menjadi lebih sederhana. Kita tuliskan input string ke dalam tape, dan head akan memulai membacanya dari karakter pertama di ujung kiri tape.



Gambar 7.3 struktur mesin Turing

Karena komponen head pada mesin Turing dapat bergerak bebas ke kiri maupun kanan, maka kita dapat dengan mudah merancang mesin yang dapat menerima bahasa $A^n B^n C^n$. Tinggal tuliskan saja input string ke dalam tape yang tersedia. Dan head akan menandai karakter a pertama di sisi paling kiri. Kemudian bergerak ke kanan mencari karakter b pertama. Jika ketemu, maka akan ditandai. Dan selanjutnya akan kembali bergerak ke kanan untuk mencari karakter c pertama. Setelah ketemu dan ditandai, maka head akan kembali ke kiri untuk mencari karakter a kedua, dan mengulangi lagi langkah-langkah di atas. Jika tidak ada lagi karakter a yang ditandai, maka pergerakan terakhir ke kanan akan mencari karakter b dan/atau c yang mungkin masih tersisa. Jika tidak ditemukan karakter tersisa maka input string yang tertulis dalam tape dapat dinyatakan accepted (diterima). Dan rejected untuk sebaliknya.

Di sisi lain, FA dan PDA keduanya adalah tipe mesin yang memiliki jaminan berhenti. Artinya, apapun input stringnya, keduanya akan berhenti begitu semua karakter dalam input string selesai dibaca. Dan hasilnya sudah jelas, keputusan accepted atau rejected. Sementara mesin Turing rupanya tidak memiliki garansi serupa. Karena input tertulis dalam tape, dan komponen head-lah yang harus bergerak. Ada kemungkinan head akan bergerak maju-mundur (kiri-kanan) terus. Atau bergerak ke satu arah tanpa berhenti.

Namun yang pasti mesin Turing ini memiliki kemampuan komputasi yang lebih hebat dibandingkan FA maupun PDA. Pada bab selanjutnya kita akan mengetahui bagaimana mesin Turing ini dapat menjelaskan proses komputasi yang ditulis dalam sebarang bahasa pemrograman atau yang berjalan pada mesin komputer modern. Tetapi tetap ada resiko dengan menggunakan mesin Turing ini, yaitu tidak adanya garansi berhenti. Dan sampai saat inipun belum ada algoritma yang dapat membantu memastikan apakah mesin Turing yang kita pakai akan dapat berhenti normal atau tidak. Inilah batasan komputasi yang disebut sebagai undecidability of the halting problem.

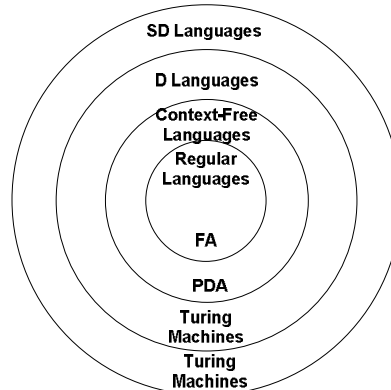
Selanjutnya kita akan menggunakan mesin Turing untuk mencoba mendefinisikan 2 klas bahasa baru, yaitu:

- Sebuah bahasa L disebut **decidable** jika dan hanya jika terdapat sebuah mesin Turing M yang dapat berhenti (dengan normal) untuk setiap input yang diberikan, menerima semua string yang ada dalam L, dan menolak semua string yang bukan anggota dari L. Atau dengan kata lain, M dapat diandalkan untuk mengeluarkan keputusan Ya atau Tidak, dengan cara yang benar.
- Sebuah bahasa L disebut **semidecidable** jika dan hanya jika terdapat sebuah mesin Turing M yang dapat menerima semua string dalam L, dan gagal menerima string yang bukan anggota dari L. Atau jika diberikan string yang bukan anggota L, M mungkin akan menolak atau justru akan terjadi loop terus menerus. Dengan kata lain, M dapat mengenali anggota L dengan mudah dan memutuskan Ya. Tetapi kadang tidak tahu kapan harus menghentikan penelusuran dan memutuskan Tidak untuk string yang bukan anggota L.

Sampai sejauh ini kita telah berhasil mendefinisikan 4 klas bahasa:

1. Regular languages, yang dimodelkan dengan finite automata;
2. Context-free languages, yang dapat dimodelkan dengan pushdown automata;
3. Decidable (D) languages, yang dapat dimodelkan dengan mesin Turing (dimana setiap penelusuran yang dilakukan akan selalu berakhir/berhenti dengan benar);
4. Semidecidable (SD) language, yang dapat dimodelkan (walaupun tidak sempurna) dengan mesin Turing (karena penelusuran akan berakhir/berhenti dengan benar hanya untuk string yang menjadi anggota bahasa yang didefinisikan oleh mesin tersebut).

Dan disebabkan oleh kemampuannya, setiap klas bahasa otomatis menjadi proper subset bagi klas bahasa selanjutnya (ditunjukkan oleh diagram pada gambar 7.4 berikut).



Gambar 7.4 Hirarki klas bahasa

Dari beberapa klas bahasa yang telah kita pelajari, kita mengetahui ada pilihan alat untuk mengekspresikan setiap bahasa dengan tingkat ekspresivitas yang berbeda-beda. Sebagai contoh, kita dapat memilih untuk mendefinisikan bahasa A^nB^n sebagai context-free language (bukan sebagai bagian dari bahasa regular). Atau memilih mendefinisikan bahasa $A^nB^nC^n$ sebagai bahasa decidable (bukan sebagai bahasa context-free ataupun bahasa regular). Itulah pilihan-pilihan yang tersedia untuk mendefinisikan atau memodelkan bahasa. Tetapi bagaimanapun pilihan alat untuk mengekspresikan bahasa ini “harga”-nya. Harga yang dimaksud adalah:

- **Computational efficiency:** semua mesin FA akan berjalan pada slot waktu linier sepanjang input string yang sedang dikenalnya. Sementara kebanyakan context-free parser yang menggunakan PDA sebagai basis modelnya akan membutuhkan waktu yang bertumbuh secara kubik selama mesin tersebut mengenali input string yang diberikan. Sedangkan mesin Turing justru membutuhkan waktu yang berkembang secara eksponen di dalam mengenali sebuah input string.
- **Decidability:** akan terdapat prosedur-prosedur yang jelas di dalam menjawab banyak pertanyaan mengenai FA. Seperti misalnya, apakah FA menerima string-string tertentu? Bilamanakah sebuah FA dapat disebut minimal? (atau apakah ini adalah mesin paling sederhana untuk mengerjakan sebuah pekerjaan?) bagaimana 2 FA dapat dikatakan identik?

Sebagian pertanyaan di atas masih dapat dijawab apabila kita menggunakan PDA. Namun tidak satupun pertanyaan di atas dapat dijawab untuk mesin Turing.

- **Clarity:** terdapat tools yang dapat kita gunakan untuk menganalisa FA. Karena setiap bahasa regular umumnya dapat dijelaskan dengan regular expression (yang akan kita kupas pada bab berikutnya). Demikian pula setiap bahasa context-free yang dapat dikenali melalui PDA, biasanya dapat dijelaskan/dimodelkan dengan context-free grammar. Namun hal ini tidak berlaku untuk mesin Turing. Sampai sejauh ini tidak ada tools yang dapat digunakan untuk menjelaskan bahasa decidable atau semidecidable.

7.4. Latihan Soal

- Misalkan $L_1 = \{a^n b^n; n > 0\}$ dan $L_2 = \{c^n; n > 0\}$. Untuk setiap string berikut, tentukan apakah string tersebut merupakan elemen dari $L_1 L_2$:
 - λ
 - aaabbccc
 - aaabbbccc
 - abc
- Misalkan $L_1 = \{\text{bakpia, roti}\}$ dan $L_2 = \{\text{coklat, keju, strawberry, nanas}\}$. Tuliskan semua elemen untuk himpunan baru $L_1 L_2$.
- Carilah alternatif lain untuk mendefinisikan bahasa $L_2 = \{x, xxx, xxxxx, \dots\}$ secara rekursif.
- Dengan menggunakan definisi rekursif alternatif-2, ada berapa banyak cara untuk membuktikan bahwa bilangan 14 adalah genap?
- Dengan menggunakan definisi rekursif alternatif-2, berapa jumlah langkah minimal yang dibutuhkan untuk membuktikan bahwa bilangan 100 adalah genap?
- Dengan menggunakan definisi rekursif untuk pendefinisian bilangan genap, tunjukkan bahwa digit terakhir dari setiap anggota himpunan bilangan genap adalah 0, 2, 4, 6, atau 8.
- Didefinisikan sebuah himpunan ekspresi aljabar yang disebut ALEX seperti berikut:

Aturan 1: semua polinomial adalah anggota ALEX

Aturan 2: jika $f(x)$ dan $g(x)$ ada dalam ALEX, maka demikian pula dengan:

 - $f(x)$
 - $-f(x)$
 - $f(x) + g(x)$
 - $f(x) - g(x)$
 - $f(x)g(x)$
 - $f(x)/g(x)$
 - $f(x)^{g(x)}$
 - $f(g(x))$
 - tunjukkan bahwa $(x + s)3x$ adalah anggota ALEX
 - apakah aturan 2(viii) sebenarnya memang diperlukan untuk mendefinisikan ALEX?
- Dengan memperhatikan fakta bahwa $3x^2 + 7x - 9 = (((((3)x) + 7)x) - 9)$, tunjukkan bagaimana kita dapat membentuk polinomial tersebut menggunakan POLINOMIAL tetapi hanya mengaplikasikan operator perkalian sebanyak 2 kali saja? Berapa minimal langkah yang diperlukan untuk membentuk $x^8 + x^4$? Berapa minimal langkah yang diperlukan untuk membentuk $7x^7 + 5x^5 + 3x^3 + x$?
- Buatlah 2 macam definisi rekursif untuk membangun himpunan Powers-of-Two:

Powers-of-Two = $\{1, 2, 4, 8, 16, \dots\}$

Gunakan salah satu dari definisi tersebut untuk membuktikan bahwa hasil perkalian dari 2 anggota Powers-of-Two akan menghasilkan bilangan yang juga Powers-of-Two.

BAB 8. REGULAR LANGUAGES

Pembahasan di dalam bab ini akan dimulai dari pengenalan notasi yang disebut sebagai regular expression (ekspresi regular). Dan untuk selanjutnya, sama seperti istilah-istilah lain dalam buku ini, kami akan konsisten menulis/menyebut istilah penting tetap dalam bahasa Inggris agar mempermudah pemahaman bagi pembaca. Regular expression merupakan notasi pendefinisian bahasa, yang telah disinggung sekilas pada bab sebelumnya. Setiap bahasa yang dapat didefinisikan atau dispesifikasikan dengan regular expression dapat disebut bahasa regular. Regular expression dapat memberikan pola atau template mengenai bagaimana seharusnya sebuah kata/string dan kalimat/sentence boleh ditulis pada suatu bahasa tertentu.

Selain dengan regular expression kita dapat pula memanfaatkan mesin finite automata untuk memvisualisasikan spesifikasi suatu bahasa. Melalui mesin visual ini kita dapat melihat lebih jelas tentang bagaimana sebuah string dikenali, dan untuk selanjutnya diputuskan apakah string tersebut merupakan anggota dari bahasa yang bersangkutan atau tidak.

8.1. Regular Expression

Walaupun bidang ilmu yang sedang kita pelajari sekarang bernama teori komputasi, namun di dalamnya berisi rangkuman penemuan berbagai topik komputasi dari banyak ilmuwan yang masing-masing bekerja secara individual. Hasil-hasil riset tersebut, walaupun memiliki latar belakang berbeda dan sepiintas nampak seperti blok-blok terpisah, namun ternyata dapat digabung/disusun menjadi sebuah bangunan utuh dimana blok-blok tersebut dapat saling menunjang dan memperkuat satu dengan lainnya.

Sebuah perjalanan sejarah yang panjang ini diawali pada awal abad ke-20 dimana 2 orang ilmuwan mathematical logic, Georg Cantor dan David Hilbert (pada kesempatan dan waktu yang terpisah) dihadapkan pada kenyataan munculnya banyak situasi paradoks, yang mengarah kepada ‘tuntutan’ akan adanya teori dan algoritma untuk membuktikan teori-teori matematika yang telah mapan.

Dan ternyata dibutuhkan banyak waktu untuk menjawab pertanyaan tentang keberadaan teori pembuktian (proof theorems). Dimulai oleh Kurt Godel yang menyatakan bahwa teori pembuktian mungkin dapat dibuat, tetapi tidak semua ‘true statements’ dalam dunia matematika dapat dibuktikan oleh teori pembuktian tersebut. Hingga disusul oleh ilmuwan-ilmuwan berikutnya, seperti Alonzo Church, Stephen Kleene, Emil Post, Andrei Andreevich Markov, John von Neumann, dan Alan Turing. Dimana masing-masing secara terpisah berusaha membangun teori, algoritma, dan model yang universal. Namun hasilnya adalah jawaban-jawaban atomik, dimana masing-masing algoritma atau model mesin tersebut tetap tidak mampu bekerja secara universal untuk semua kasus pembuktian.

Sebagai contoh pada tahun 1940, 2 orang *neuro-physiologist*, Warren McCulloch dan Walter Pitts mengembangkan sebuah model sistem syaraf untuk menggambarkan perilaku sensor penerima pada binatang, yang disebut Finite Automata.

Dan matematikawan Stephen Kleene selanjutnya memformalkan model sistem syaraf tiruan tersebut menjadi sebuah sistem aljabar yang berbentuk himpunan regular yang disebut sebagai Regular Expression (ekspresi regular). Hingga pada akhirnya kini kedua alat tersebut dimanfaatkan sebagai alat untuk mendefinisikan bahasa formal dan alat untuk menguji keanggotaan sebuah bahasa. Dan karena peruntukannya tersebut, *regular expression* ini sering disebut sebagai *language-defining expression* atau *language-defining tool*. Bahasa apapun yang dapat didefinisikan melalui alat ini disebut regular language.

Sebuah himpunan Regular Expression (RE) dapat didefinisikan melalui pendefinisian rekursif:

1. Setiap huruf dalam himpunan alphabet Σ dapat ditulis sebagai regular expression jika ditulis dalam huruf tebal. Dan λ (empty string) adalah regular expression;

2. Jika r_1 dan r_2 masing-masing adalah regular expression, maka demikian pula halnya dengan:
 - (i). (r_1) atau (r_2) ;
 - (ii). Produk concatenate : $r_1 r_2$
 - (iii). Produk union : $r_1 + r_2$
 - (iv). Produk closure : r_1^* , r_2^* , r_1^+ , atau r_2^+ .

Definisi di atas menggunakan simbolisasi semua huruf (dalam himpunan alphabet Σ), simbol null/empty string λ , tanda kurung, simbol operasi concatenate, union, dan closure (baik kleene * maupun positive $^+$).

Penggunaan tanda kurung dalam regular expression sepintas terkesan sangat sepele atau tidak penting. Namun jika kita cermati secara lebih mendalam maka dijamin kita akan mendapati fakta yang cukup mencengangkan. Sebagai contoh, misalkan ekspresi $r_1 = aa + b$. Jika kita berikan operasi kleene closure kepada ekspresi tersebut, r_1^* , maka akan menghasilkan:

$$r_1^* = aa + b^*$$

tentunya hasil di atas dapat membingungkan pembacanya. Padahal sebenarnya yang kita maksud adalah $(r_1)^*$. Artinya semua komponen string yang ada di dalam tanda kurung (dalam satu kesatuan) akan di-closure-kan bersama-sama. Sementara hasil di atas dapat ditafsirkan operator closure hanya berlaku bagi karakter b (dan bukan aa). Sehingga dengan adanya penggunaan tanda kurung diharapkan kerancuan seperti kasus di atas dapat dihindari. Dan jawaban yang benar adalah:

$$r_1^* = aa + b^* = (aa + b)^*$$

di awal bagian ini kita telah melihat bagaimana sebuah himpunan yang merepresentasikan sebuah bahasa didefinisikan. Mulai dari cara yang paling primitif (yaitu menggunakan cara listing atau menyebutkan semua anggota himpunan bahasa yang bersangkutan). Hingga memanfaatkan set theoretic notation. Maka melalui regular expression, cara peng-ekspresi-an bahasa dapat menjadi lebih variatif. Rangkaian contoh-contoh berikut akan memberikan gambaran betapa sederhananya pengungkapan bahasa dengan regular expression itu.

Contoh:

Misal terdapat sebuah bahasa $L_2 = \{\lambda, x, xx, xxx, xxxx, \dots\}$.

Bahasa tersebut dapat pula dituliskan berbasiskan simbol seperti berikut :

$S = \{x\}$, dimana $L_2 = S^*$

Atau, bahasa L_2 dpt pula dinyatakan dlm format *set theoretic notation*:

$$L_2 = \text{Language}(x^*)$$

Atau secara ringkas

$$L_2 = (x^*).$$

Contoh:

Sebuah bahasa yang memiliki string-string yang terdiri dari bentukan huruf a dan b dapat didefinisikan melalui regular expression seperti contoh berikut:

$$(a + b)^* a (a + b)^*$$

Regular expression di atas memiliki makna bahwa:

- a. semua string dalam bahasa tersebut akan berada dalam domain himpunan alphabet $\Sigma = \{a, b\}$;
- b. semua string dalam bahasa tersebut akan memiliki bentuk umum berupa prefix substring a dan/atau b, untuk kemudian diikuti oleh infix karakter a, dan diakhiri oleh postfix substring a dan/atau b.

sebagai contoh, string abbaab adalah anggota dari bahasa tersebut, karena kita dapat memperoleh string abbaab dengan cara:

$$(\lambda)a(bbaab) \quad \text{atau} \quad (abb)a(ab) \quad \text{atau} \quad (abba)a(b)$$

Dengan pendekatan yang sama, maka kita dapat dengan mudah berkreasi menciptakan bahasa-bahasa baru yang lebih variatif, seperti dapat dilihat pada contoh-contoh berikut:

Contoh:

Sebuah bahasa yang memiliki kata dengan setidaknya terdapat 2 karakter a di dalamnya dapat dituliskan dalam regular expression berikut

$$(a + b)^* a (a + b)^* a (a + b)^*$$

Contoh:

Sebuah bahasa yang memiliki kata dengan terdapat 2 karakter a di dalamnya dapat dituliskan dalam regular expression berikut

$$b^* a b^* a b^*$$

Contoh :

Misalkan terdapat himpunan alphabet $\Sigma = \{a, b\}$

Dan melalui himpunan tersebut didefinisikan sebuah bahasa $L_1 = \{a, ab, abb, abbb, abbbb, \dots\}$

Maka melalui regular expression, L_1 dapat dinyatakan sebagai $L_1 = (ab^*)$.

Contoh :

Misalkan terdapat himpunan alphabet $\Sigma = \{x\}$

Dan melalui himpunan tersebut didefinisikan sebuah bahasa $L_2 = \{x, xx, xxx, \dots\}$

Maka melalui regular expression, L_2 dapat dinyatakan sebagai $L_2 = (x^+)$.

Contoh-contoh di atas diharapkan mampu memperdalam pengertian kita mengenai pemanfaatan regular expression di dalam mendefinisikan bahasa-bahasa formal. Namun tetap saja ada hal yang perlu kita ingat, khususnya di dalam memainkan operator closure. Bahwa

$$xx^* = x^+ = xx^*x^* = x^*xx^* = x^+x^* = x^*x^+ = \dots$$

mungkin beberapa contoh lagi akan semakin menambah pemahaman kita akan karakteristik operator himpunan.

Contoh :

Misalkan terdapat himpunan alphabet $\Sigma = \{a, b\}$

Dan melalui regular expression tersebut akan didefinisikan sebuah bahasa $L_3 = \{\lambda, a, b, aa, bb, ab, ba, aaa, aab, aba, baa, \dots\}$

Maka melalui regular expression, L_3 dapat dinyatakan sebagai $L_3 = (a^*b^*)$.

Contoh di atas mengingatkan kita akan pentingnya peranan tanda kurung, karena $a^*b^* \neq (ab)^*$. Dan untuk selanjutnya kita harus lebih berhati-hati lagi dalam menyatakan sebuah ekspresi.

Contoh :

Jika terdapat 2 himpunan bahasa $P = \{a, bb, bab\}$ dan $Q = \{\lambda, bbb\}$

Maka concatenation kedua himpunan tersebut akan menghasilkan $PQ = \{a, bb, bab, abbb, bbbbb, babbbb\}$

Dan regular expression dari bahasa tersebut adalah $(a + bb + bab)(\lambda + bbb)$.

Contoh :

Jika terdapat 2 himpunan bahasa $M = \{\lambda, x, xx\}$ dan $Q = \{\lambda, y, yy, yyy, \dots\}$

Maka concatenation kedua himpunan tersebut akan menghasilkan $PQ = \{\lambda, y, yy, yyy, \dots, x, xy, xyy, xyyy, \dots, xx, xxy, xxyy, xxyyy, \dots\}$

Dan regular expression dari bahasa tersebut adalah $(\lambda + x + xx)(y^*)$ atau $y^* + xy^* + xxy^*$.

8.2. Finite Automata

Bahasa formal (yang sejauh telah kita kenal dan dapat didefinisikan melalui ekspresi regular) dapat dipandang sebagai himpunan entitas abstrak, atau berupa sekumpulan string yang berisi simbol-simbol alphabet. Dimana dalam bentuk entitas abstrak, bahasa-bahasa tersebut dapat dikenali atau di-generate oleh mesin komputasi.

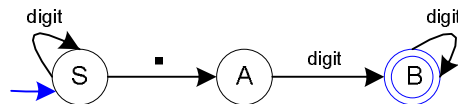
Dalam subbab ini kita akan mengenal sebuah klas mesin komputasi yang sesuai untuk kelas bahasa di atas, yaitu Finite (State) Automata (FA). FA pada dasarnya adalah sebuah model matematika yang bentuknya sepintas menyerupai diagram alir dengan input dan output diskrit.

Di awal perjalanan FA, diciptakan sub-subklas mesin yang disebut Finite State Machine (FSM) atau State Transition Diagram (STD).

Pada dasarnya FSM dan STD adalah model mesin yang dapat digunakan untuk memodelkan semua fenomena yang ada di dunia ini. Dimana sebuah fenomena/masalah yang dapat dimodelkan melalui FA dapat diartikan bahwa masalah tersebut bersifat computational solved (dapat diselesaikan secara komputatif). Dan sebaliknya, masalah tersebut tidak dapat diselesaikan secara komputasional apabila kita tidak dapat menemukan model untuk permasalahan tersebut.

Namun dalam konteks pembahasan kita kali ini adalah mempelajari tentang pengembangan bahasa. Jadi masalah kita adalah bagaimana mesin FA (dengan semua varian mesin yang dimilikinya) dapat membantu kita memodelkan rancangan bahasa pemrograman yang akan kita buat? Sehingga mulai saat ini hingga akhir pembahasan kita akan berkuat dengan pemodelan bahasa.

Untuk memudahkan pemahaman mengenai komponen dan karakteristik FA (khususnya FSM atau STD sebagai varian pertama yang kita kenal) maka contoh berikut akan menjelaskan sebuah desain model mesin FSM yang berfungsi memeriksa kebenaran penulisan bilangan riil:



Mesin di atas terlihat memiliki 3 buah state (yang diwakili oleh gambar lingkaran-lingkaran berlabel). Dan aliran dalam mesin tersebut ditunjukkan oleh garis-garis berarah (yang masing-masing juga mempunyai label) yang menghubungkan ketiga lingkaran yang ada. Awal dari aliran dalam mesin tersebut ditunjukkan oleh garis tidak berlabel yang menempel pada salah satu state (start state), dan akhir dari aliran ditunjukkan oleh state yang memiliki tanda lingkaran ganda (final state).

Karena mesin di atas sebenarnya adalah bagian dari sebuah bahasa pemrograman, maka mesin tersebut dapat digunakan untuk mengecek/memeriksa kebenaran penulisan bilangan riil yang ditulis oleh pengguna bahasa pemrograman (programmer) pada saat menyusun sebuah program.

Untuk mengetahui cara kerja dari mesin di atas, maka paling mudah adalah kita coba dengan memberikan inputan berupa bilangan kepada mesin tersebut. Misalkan kita berikan/inputkan bilangan 9.8765, maka proses yang terjadi dalam mesin tersebut secara berurutan dapat digambarkan seperti berikut:

Status/Posisi Mesin	Action/Proses Mesin
(S, 9.8765)	dibaca 9 dan FSM tetap di state S
(S, .8765)	dibaca . dan FSM ada di state A
(A, 8765)	dibaca 8 dan FSM ada di state B
(B, 765)	dibaca 7 dan FSM ada di state B
(B, 65)	dibaca 6 dan FSM ada di state B
(B, 5)	dibaca 5 dan FSM tetap di state B
(B,)	

karena rangkaian input telah habis dan B merupakan final state, maka penulisan 9.8765 dinyatakan benar oleh bahasa/mesin di atas.

Setelah mengamati cara kerja di atas, maka diharapkan anda memahami alasan mengapa jika kita menuliskan “,” sebagai penanda desimal pada Microsoft Excel dianggap salah?! Itu karena desain mesin pengenalan bilangan riil pada Microsoft Excel di-set menggunakan simbol “.” dan bukan “,”.

Dan untuk lebih mengenal karakteristik FSM maka berikut kita coba menjalankan mesin contoh di atas dengan menggunakan inputan yang salah.

Input string a Tidak dikenali oleh FSM. Karena tidak ada panah berarah yang keluar dari start state yang memiliki label “a”. Sehingga kita tidak mencapai final state karena penelusuran tidak dapat berjalan sama sekali.

Input string 9, Penelusuran berhenti di S (bukan final state), sehingga input “9,” dinyatakan salah oleh mesin tersebut.

Input string 9. Penelusuran berhenti di A (bukan final state), sehingga input “9.” juga dinyatakan salah oleh mesin.

Input string 98765 Setelah beberapa kali looping di start state S, maka penelusuran harus berhenti di S (bukan final state), sehingga input “98765” juga dinyatakan salah oleh mesin.

Deterministic Finite Automata (DFA)

Setelah mengenal FSM yang merupakan varian pertama dari FA, maka berikutnya kita akan mengenal sebuah varian lain dari FA yang sangat populer di kalangan ahli simulasi dan pemodelan, yakni Deterministic Finite Automata (DFA).

Memang awalnya Finite Automata (yang diilhami oleh cara kerja jaringan syaraf neuron pada otak manusia) diciptakan untuk memvisualisasikan model bahasa formal yang dinyatakan melalui ekspresi regular. Dan dalam perkembangannya bidang ilmu pemodelan dengan FA ini terasa sangat lambat perkembangannya. Hingga terkesan bahwa yang dapat dilakukan menggunakan FA hanyalah memodelkan bahasa (pemrograman) saja. Baru setelah puluhan tahun muncullah varian-varian lain dari FA yang dapat diaplikasikan ke banyak aspek kehidupan lain (selain desain bahasa pemrograman). Sebut saja Cellular Automata, Hybrid Automata, Timed Automata, Stochastic Automata, dan lain sebagainya.

Nah, kembali kepada DFA. Jika diterjemahkan secara bebas deterministic finite automata dapat diartikan sebagai finite automata yang memiliki ketentuan/batasan yang sangat kuat atau mutlak. Ada satu jenis permainan anak-anak yang mungkin anda masih mengenal/mengingatnya. Yaitu Permainan ular dan tangga. Jika anda masih ingat, permainan ini memiliki sejumlah kotak yang tersusun sedemikian rupa, dimana kotak-kotak tersebut terurut berdasarkan nomor. Kemudian ada beberapa gambar tangga (dan ular), dimana kedua ujung dari masing-masing tangga dan ular tersebut berasosiasi dengan sebuah kotak secara unik. Dan dalam permainan ular dan tangga, urutan nomor kotak beserta tangga dan ular akan menjadi pedoman (guidance) di dalam menjalankan permainan ini. Arah pergerakan setiap pemain harus sesuai dengan pedoman tersebut. Konsep inilah yang disebut dengan deterministic.

Sementara jika anda cermati dengan teliti, maka anda akan menemukan bahwa semua komponen permainan ini bersifat terbatas (finite). Mulai dari jumlah kotak yang ada dalam permainan ini ($n \times n$), jumlah inputan yang mungkin didapat oleh setiap pemain (1 – 6 jika menggunakan 1 dadu, atau 1 – 12 jika menggunakan 2 dadu), sampai dengan jumlah pemain yang terlibat (1 – 4 orang, sesuai dengan ketersediaan jumlah penanda berupa kerucut merah, hijau, biru, dan kuning). Barangkali ini akan menjelaskan pengertian finite dari mesin yang sedang kita bahas.

Walaupun permainan ular dan tangga ini dijalankan oleh manusia, namun perlu diketahui bahwa manusia sebenarnya tidak menentukan apa-apa. Karena berapa langkah dia harus bergerak atau ke mana penanda harus digeser telah ditentukan oleh hal lain. Jumlah pergeseran ditentukan oleh hasil lemparan dadu. Sementara arah pergerakan telah ditentukan susunan kotak, tangga dan juga ular. Konsep inilah yang disebut automaton atau automata.

Secara definitif, sebenarnya DFA memiliki komponen-komponen yang sama dengan FSM yaitu:

1. S sebagai himpunan berhingga state untuk media perpindahan kendali mesin;
2. Σ sebagai himpunan berhingga alphabet untuk input karakter;
3. s_0 adalah salah satu state dari himpunan S yang diperlakukan sebagai start state;
4. s_n adalah salah satu state dari himpunan S yang diperlakukan sebagai final state;
(DFA dapat memiliki lebih dari satu final state)
5. δ sebagai himpunan berhingga fungsi transisi untuk memindahkan kendali mesin.

Untuk memudahkan pemahaman mengenai komponen DFA di atas, maka kita lihat contoh berikut:
Misalkan kita akan mendefinisikan sebuah mesin DFA yang memiliki

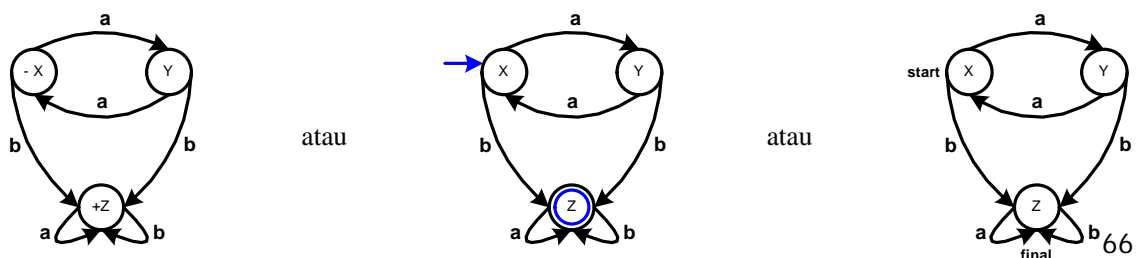
- Himpunan State $S = \{X, Y, Z\}$
- Himpunan alphabet $\Sigma = \{a, b\}$
- $X \in S$ sebagai start state
- $Z \in S$ sebagai final state
- Himpunan fungsi transisi δ didefinisikan sebagai :
 1. Dari X diberi input a ke Y
 2. Dari X diberi input b ke Z
 3. Dari Y diberi input a ke X
 4. Dari Y diberi input b ke Z
 5. Dari Z diberi input a atau b ke Z

Pernyataan di atas sebenarnya sudah dapat dianggap sebagai bentuk pendefinisian mesin. Namun bagaimanapun bentuk pernyataan di atas masih relatif sulit untuk dipahami. Oleh karena itu untuk tujuan memudahkan pemahaman dan penggunaan, maka perlu dicari alternatif pernyataan lain yang lebih mudah dan menarik. Salah satunya adalah menyatakan mesin DFA dalam format tabel yang disebut sebagai **Tabel Transisi**. Contoh tabel transisi untuk mesin pada contoh di atas adalah seperti berikut:

	a	b
(start) X	Y	Z
Y	X	Z
(final) Z	Z	Z

Tabel di atas memiliki makna yang sama dengan pernyataan mengenai mesin tersebut sebelumnya. Yaitu bahwa mesin ini memiliki 3 buah state, masing-masing state X , state Y , dan state Z . Dimana state X diperlakukan sebagai start state, dan state Z sebagai final statenya. Mesin tersebut juga memiliki 2 jenis karakter input, yaitu a dan b . Fungsi transisi pada tabel tersebut di atas juga menghasilkan pembacaan yang sama seperti pendefinisian fungsi transisi pada pernyataan sebelumnya.

Bentuk pernyataan lain yang lazim digunakan pada banyak kasus adalah bentuk **representasi piktorial** menggunakan labelled directed graph (seperti yang telah kita kenal pada pendefinisian FA di awal subbab ini). Dimana untuk mesin DFA yang sedang kita bahas dalam contoh ini dapat digambarkan seperti berikut:



Jika anda perhatikan, ketiga state pada masing-masing gambar di atas menyatakan state yang sama dengan pernyataan mengenai mesin tersebut sebelumnya (termasuk start dan final state-nya). Demikian pula dengan origin dan terminus dari setiap panah berarah (arc) yang ada. Oleh karena itu ketiga cara pernyataan di atas sebenarnya menyatakan mesin yang identik.

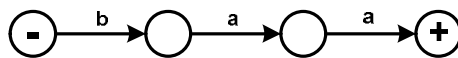
Dalam perkembangannya, muncul modifikasi pada DFA dengan mengubah beberapa karakteristik dasar DFA. Tujuan dari modifikasi ini adalah agar kita dapat memiliki mesin yang lebih 'powerful'. Karena untuk beberapa kasus pada subbab lain buku ini kita dituntut memiliki mesin yang lebih fleksibel namun masih dalam domain deterministik. Oleh karenanya diciptakan sebuah subklas mesin yang disebut Transition Graph (TG). TG ini dihasilkan dari beberapa modifikasi karakteristik pada DFA, diantaranya adalah:

- Label panah berarah (arc) dapat berupa string (pada DFA label arc harus berupa 1 karakter tunggal);
- Diperbolehkannya memiliki lebih dari 1 start state;
- Outgoing arc pada sebuah state dapat lebih dari 1, walaupun arc tersebut memiliki label yang sama. Dan modifikasi ketiga inilah yang sampai saat ini masih menjadi perdebatan karena jika keinginan awalnya adalah menjaga agar kita masih bermain dalam domain deterministik, maka munculnya aturan ketiga ini justru membuat TG berada garis batas. Oleh karenanya ada sebagian pihak yang mengatakan bahwa TG sebenarnya adalah bentuk primitif dari mesin non deterministik.

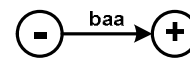
Tetapi bagaimanapun kehadiran TG ini dapat mengubah cara pandang kita terhadap mesin finite automata karena meskipun menyatakan bahasa yang sama, bentuk kedua mesin (DFA dan TG) akan terlihat cukup ekstrem perbedaannya.

Contoh-contoh berikut akan menjelaskan maksud kalimat di atas.

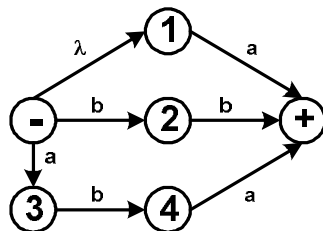
DFA-1:



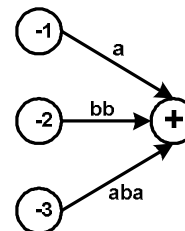
TG-1:



DFA-2:



TG-2:



8.3. Regular Language & Nonregular Language

Dengan munculnya cara mengekspresikan bahasa menggunakan ekspresi reguler (regular expression) dengan segala (keter)batasannya, maka muncul pula dikotomi bahasa reguler dan bahasa non reguler. Dikatakan bahasa reguler (regular language) jika bahasa tersebut dapat dinyatakan dengan regular expression. Sementara bahasa yang tidak dapat dinyatakan dengan regular expression disebut bahasa non reguler.

Regular Language

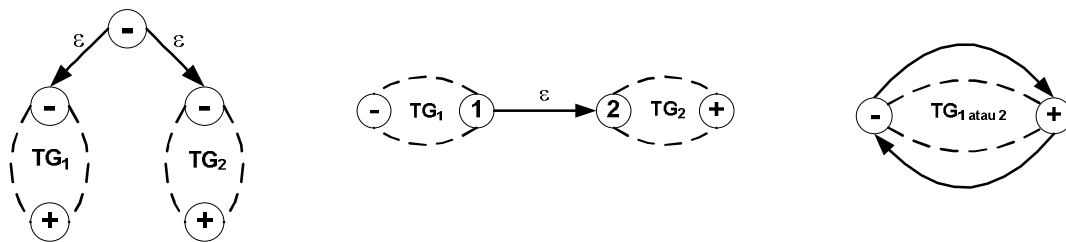
Muncul konsekuensi lain pada saat kita memperoleh kepastian bahwa sebuah bahasa dapat dinyatakan dengan regular expression (RE). Ini karena pada kalimat-kalimat RE yang kita bangun, kita dapat

menyimpulkan operator aljabar yang juga berlaku untuk himpunan, seperti union, concatenation, ataupun closure. Sehingga muncul 'tuntutan' bahwa operator-operator aljabar tersebut juga harus dapat diaplikasikan pada finite automata (khususnya yang bertipe deterministik). Jadi berikut ini kita akan mencoba mengaplikasikan operasi-operasi tersebut pada finite automata, plus beberapa operasi lain seperti komplemen dan interseksi.

Teorema 13.1. Jika L_1 dan L_2 masing² adalah bahasa regular, maka bahasa² sbg hasil operasi L_1+L_2 , $L_1 L_2$ dan L_1^* serta L_2^* adalah bahasa regular.

Bukti:

- Representasi bahasa regular L adalah melalui RE. Dan RE dapat menerima operasi himpunan. Sehingga hasilnya pun adalah bahasa regular.
- Selain RE, bahasa regular L dapat pula direpresentasikan dengan Transition Graph (TG). Dan TG pun harus dapat menerima operasi himpunan. Sehingga hasil operasi tsb juga berupa bahasa regular.



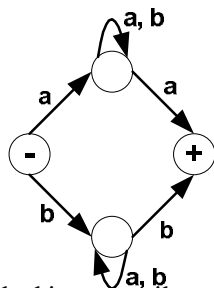
Contoh :

Misalkan terdapat 2 buah bahasa yang masing-masing dinyatakan melalui RE berikut:

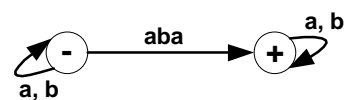
$$R_1 = a(a+b)^*a + b(a+b)^*b \quad R_2 = (a+b)^*aba(a+b)^*$$

Dimana jika dinyatakan dalam bentuk TG, kedua bahasa tersebut akan tampak seperti berikut:

TG-1:



TG-2:

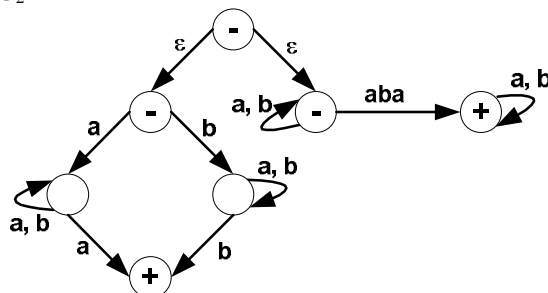


Sekarang jika kita operasikan union terhadap RE, maka diperoleh

$$R_1 + R_2 = [a(a+b)^*a + b(a+b)^*b + (a+b)^*aba(a+b)^*]$$

Dan operasi yang sama pada TG akan menghasilkan:

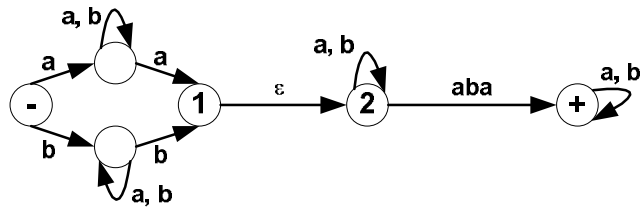
$TG_1 + TG_2 =$



Sementara jika operasi concatenate yang kita lakukan pada kedua RE, maka akan menghasilkan

$$R_1 R_2 = [a (a + b)^* a + b (a + b)^* b] [(a + b)^* aba (a + b)^*]$$

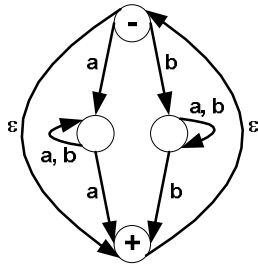
Dan operasi yang sama pada TG akan menghasilkan:

$$TG_1 + TG_2 =$$


Sedangkan jika operasi closure yang kita lakukan pada salah satu RE (misalnya R_1), maka akan menghasilkan

$$R_1^* = [a (a + b)^* a + b (a + b)^* b]^*$$

Dan operasi yang sama pada TG akan menghasilkan:

$$TG_1^* =$$


Teorema 13.2 Jika L adalah bahasa regular, maka komplemen dari L (L^1) juga bahasa regular.

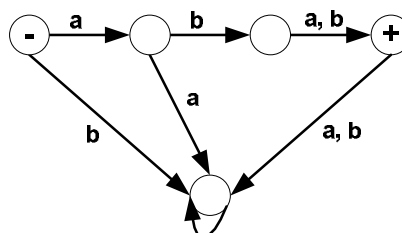
Bukti:

Komplemen dari bahasa L , adalah bahasa yang menerima semua string, SELAIN string yang diterima oleh bahasa L .

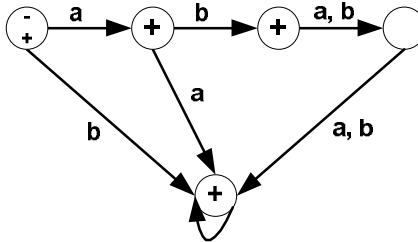
Implementasi operasi komplemen pada DFA adalah dengan menjadikan semua final state menjadi NON final state. *and vice versa*.

Contoh:

Misalkan terdapat sebuah DFA yang mendefinisikan bahasa $L = \{ aba, abb \}$ digambarkan seperti berikut:



Maka komplemen untuk bahasa L dapat diartikan sebagai $L^1 = \{ \text{semua string kecuali aba dan abb} \}$. Maka DFA untuk L^1 adalah seperti berikut :

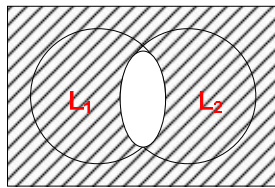


Teorema 13.3. (interseksi). Jika L_1 dan L_2 masing-masing adalah bahasa regular, maka $L_1 \cap L_2$ adalah juga bahasa regular.

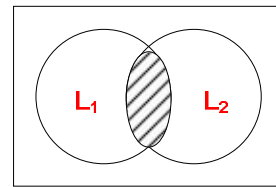
Bukti:

Pada sebarang himpunan berlaku hukum De Morgan : $L_1 \cap L_2 = (L_1^1 + L_2^1)^1$

Atau melalui diagram Venn keterhubungan di atas dapat ditunjukkan oleh ilustrasi berikut:



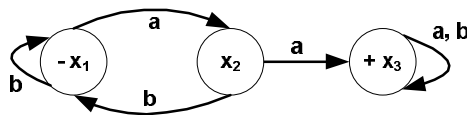
$$(L_1^1 + L_2^1)$$



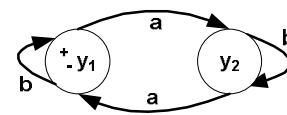
$$(L_1^1 + L_2^1)^1 = L_1 \cap L_2$$

Sebagai contoh, berikut terdapat 2 buah bahasa yang masing-masing dinyatakan oleh DFA-1 dan DFA-2:

DFA-1:

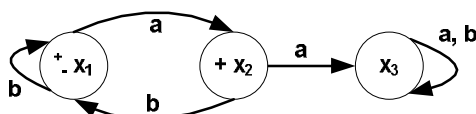


DFA-2:

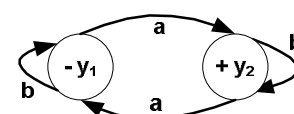


Untuk menginterseksikan kedua mesin di atas, kita akan memanfaatkan hukum de Morgan. Oleh karenanya kita perlu memiliki bentuk komplemen dari masing-masing DFA di atas:

DFA-1¹:

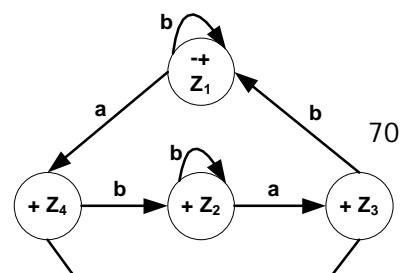


DFA-2¹:



$$(DFA-1^1 + DFA-2^1) =$$

	a	b	State Asal
--	---	---	------------



$\pm Z_1$	Z_4	Z_1	$X_1 \mid Y_1$
$+ Z_2$	Z_3	Z_2	$X_1 \mid Y_2$
$+ Z_3$	Z_6	Z_1	$X_2 \mid Y_1$
$+ Z_4$	Z_5	Z_2	$X_2 \mid Y_2$
Z_5	Z_6	Z_5	$X_3 \mid Y_1$
$+ Z_6$	Z_5	Z_6	$X_3 \mid Y_2$

Nonregular Language

Seperti telah disinggung sebelumnya bahwa sebuah bahasa dikatakan bertipe non reguler apabila bahasa tersebut tidak dapat dinyatakan dengan regular expression (RE). beberapa contoh berikut akan membantu menjelaskan tentang fenomena tipe bahasa ini.

1. Sebuah bahasa $L = \{ \lambda, ab, aabb, aaabbb, \dots \}$.
Atau L dapat pula dinyatakan dengan set theoretic notation $L = \{ a^n b^n, \text{ dengan } n = 0, 1, 2, 3, \dots \}$.
Namun yang pasti L tidak dapat dinyatakan dengan RE.

2. Sebuah bahasa yang memiliki anggota berupa string-string dengan jumlah karakter bilangan prima. Atau Bahasa Prima = $\{ aa, aaa, aaaaa, \dots \}$.
Dalam format set theoretic notation, bahasa di atas dapat ditulis sebagai Bahasa Prima = $\{ a^n \text{ dengan } n \text{ adalah bilangan prima} \}$

Identifikasi keberadaan sifat non regular pada sebuah bahasa dapat menggunakan Pumping Lemma. Lemma ini akan mem-'pompa'-kan substring-substring tertentu ke dalam string tertentu untuk menghasilkan string-string baru. Dan apabila string-string baru tersebut merupakan anggota dari bahasa yang didefinisikan, maka bahasa tersebut adalah bahasa reguler.

Tetapi sebaliknya jika pumping lemma tidak dapat diaplikasikan pada sebuah bahasa, maka bahasa tersebut adalah bahasa non reguler.

Lemma 13.1. (Pumping Lemma). Misal L adalah sebarang bahasa reguler yang memiliki himpunan string tak berhingga. String-string pada bahasa reguler tersebut dapat dikelompokkan menjadi 3 substring X , Y , dan Z (dimana Y bukan merupakan null string), sedemikian hingga ketiga substring di atas dapat dikembangkan menjadi string-string baru yang berbentuk :

$X Y^n Z$ untuk $n = 1, 2, 3, \dots$

dimana string di atas adalah anggota dari bahasa L .

Contoh :

Kita akan mencoba membuktikan apakah bahasa $L = \{ a^n b^n, \text{ untuk } n = 0, 1, 2, \dots \}$ adalah bahasa non reguler.

Kita asumsikan bahwa bahasa L adalah bahasa reguler, sehingga string-stringnya dapat dikelompokkan ke dalam subtring XYZ .

Misal kita ambil salah satu string: aabb.

Kemudian kita misalkan $X = a$, $Y = ab$, dan $Z = b$.

Multiplikasi pada substring Y akan menghasilkan string-string baru:

$XYZ = a ab b$, $XY^2Z = a abab b$, $XY^3Z = a ababab b, \dots$

Bisa dilihat bahwa string-string baru di atas bukan anggota dari bahasa L. Sehingga dapat kita simpulkan bahwa Pumping Lemma tidak dapat diterapkan pada bahasa L, sehingga bahasa L adalah bahasa non reguler.

8.4. Finite Automata with Output

Sampai sejauh ini automata yang kita pelajari hanya dapat memberikan hasil/jawaban berupa “ACCEPTED” atau “REJECTED” saja untuk menguji keanggotaan bahasa. Sehingga mesin kita ini masih berfungsi sebagai acceptor saja.

Tetapi sebenarnya kita dapat pula membuat sebuah transducer. Yaitu sebuah mesin yang dapat menghasilkan berbagai macam bentuk output (*automata with output*). Ada 2 model *automata with output* yang tersedia, yaitu :

- Moore Machine
- Mealy Machine

Moore Machine

Secara definitif, Moore Machine memiliki komponen-komponen :

1. Q sebagai himpunan berhingga state untuk media perpindahan kendali mesin. Label pada setiap state ditulis q_i/o , dengan q_i adalah nama state dan o adalah output yang berkorespondensi dengan state tsb;
2. Σ sebagai himpunan berhingga alphabet untuk input karakter
3. Γ sebagai himpunan berhingga karakter untuk output karakter
4. q_0 adalah salah satu state dari himpunan S yang diperlakukan sebagai start state
5. δ sebagai himpunan berhingga fungsi transisi untuk memindahkan kendali mesin.

Contoh :

Sebuah Moore Machine didefinisikan sebagai berikut :

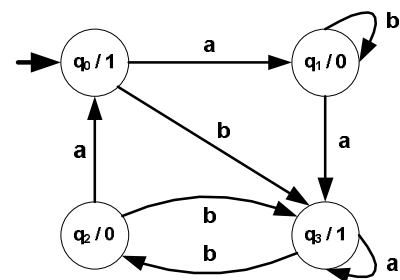
Himpunan state $Q = \{q_0, q_1, q_2, q_3\}$ dengan q_0 sebagai start state

Himpunan input karakter $\Sigma = \{a, b\}$

Himpunan output karakter $\Gamma = \{0, 1\}$

Sementara himpunan fungsi transisi didefinisikan pada tabel berikut :

State Awal	Input a	Input b	Output
- q_0	q_1	q_3	1
q_1	q_3	q_1	0
q_2	q_0	q_3	0
q_3	q_3	q_2	1



Pada contoh di atas terlihat bahwa mesin automata with output tidak memiliki final state. Hal ini dikarenakan fungsi mesinnya yang berbeda dengan finite automata sebelumnya yang berfungsi untuk melakukan checking terhadap input string. Pada automata with output, yang perlu dilakukan adalah menjalankan mesin sesuai dengan input string yang ada. Tidak ada persyaratan penelusuran harus berhenti dimana. Penelusuran dapat berhenti pada state mana saja.

Misalkan kita akan mencoba mesin di atas dengan sebuah input string: abab

Maka proses yang terjadi dapat digambarkan melalui tabel berikut:

Input		a	b	a	b
State Terkunjungi	q_0	q_1	q_1	q_3	q_2
Output	1	0	0	1	0

Mealy Machine

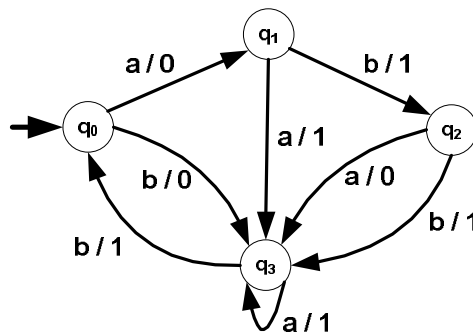
Secara definitif, Mealy Machine memiliki komponen-komponen :

1. Q sebagai himpunan berhingga state untuk media perpindahan kendali mesin.
2. Σ sebagai himpunan berhingga alphabet untuk input karakter
3. O sebagai himpunan berhingga karakter untuk output karakter
4. q_0 adalah salah satu state dari himpunan S yang diperlakukan sebagai start state
5. δ sebagai himpunan berhingga fungsi transisi untuk memindahkan kendali mesin.

Label pada setiap panah ditulis ϵ/o , dengan ϵ adalah input karakter dan o adalah output karakter yang berkorespondensi dengan kedua adjacent state tsb.

Dari uraian mengenai komponen mesin Mealy, jika dibandingkan dengan mesin Moore, maka akan nampak bahwa esensi perbedaan adalah pada penulisan karakter output. Jika pada mesin Moore karakter output dituliskan di dalam state (bersama dengan label state), maka pada mesin Mealy label karakter output terdapat pada arc (bersama dengan label arc).

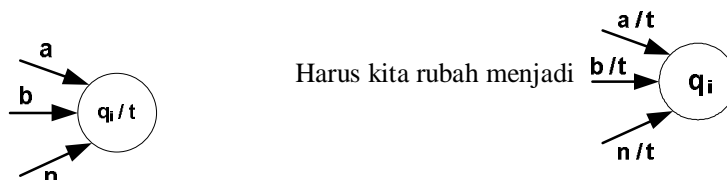
Sebagai contoh berikut terdapat Mealy Machine, kita akan mencoba melakukan penelusuran untuk input string: aaabb



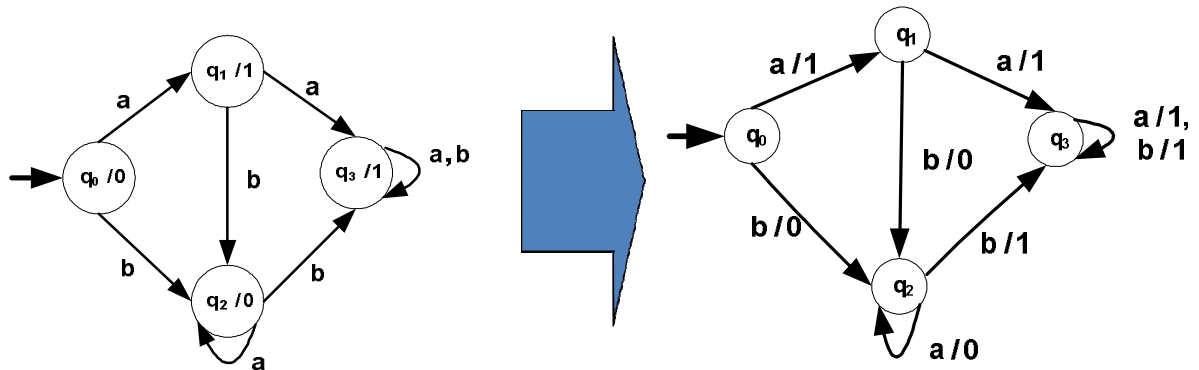
Hasil penelusuran untuk aaabb dapat kita lihat bersama pada tabel berikut:

Input		A	a	a	b	b
State Terkunjungi	q_0	q_1	q_3	q_3	q_0	q_3
Output		0	1	1	1	0

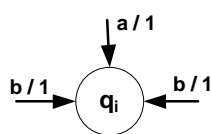
Konversi dapat dilakukan untuk mengubah Moore Machine menjadi Mealy Machine, dan sebaliknya. Untuk proses konversi dari Moore Machine ke bentuk Mealy Machine dapat dilakukan dengan sangat mudah. Kasus yang mungkin akan kita jumpai dalam proses konversi tersebut adalah:



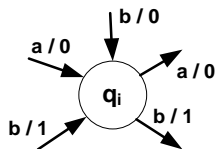
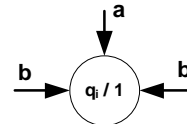
Sehingga jika memiliki mesin Moore seperti contoh pada gambar sebelah kiri, dapat dengan mudah kita rubah menjadi bentuk mesin Mealy seperti gambar sebelah kanan:



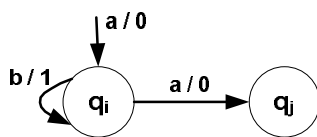
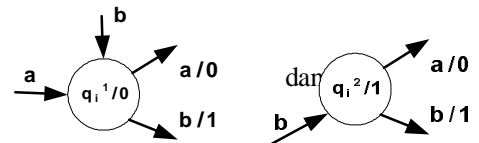
Tetapi sebaliknya mengkonversikan mesin Mealy menjadi bentuk mesin Moore tidaklah semudah proses konversi sebelumnya. Ada beberapa kasus yang mungkin kita jumpai dalam proses konversi yang kedua ini, yaitu antara lain:



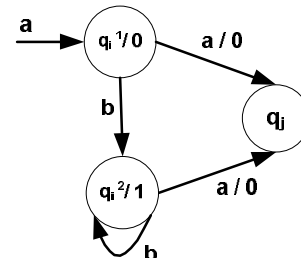
Harus kita rubah menjadi



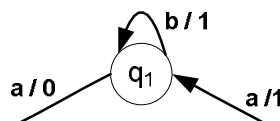
Harus kita rubah menjadi



Harus kita rubah menjadi

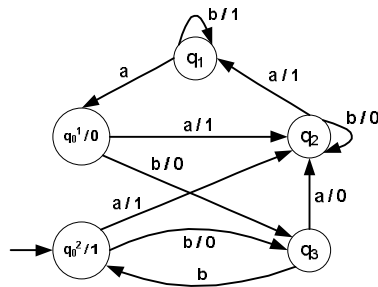


Sekarang waktunya kita mencoba mengaplikasikan petunjuk di atas pada contoh berikut: Misalkan kita memiliki sebuah mesin Mealy seperti gambar berikut, dan kita ingin merubahnya menjadi mesin Moore.

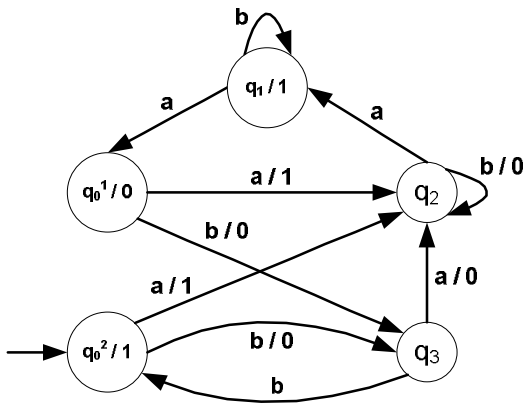


Mengerjakan proses konversi sekaligus untuk semua state rasanya bukanlah keputusan yang bijak. Oleh karena itu, kita akan lakukan proses konversi tersebut untuk masing-masing state secara bergantian. Dan untuk memudahkan, maka akan lebih baik jika anda gambarkan terlebih dahulu bagian-bagian mesin yang tidak sedang terlibat dalam proses konversi pada tahap itu. Setelah itu baru anda mengerjakan proses konversi untuk sebuah state.

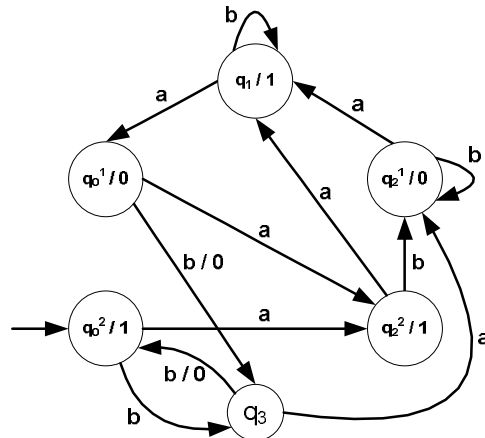
Konversi untuk state Q_0 :



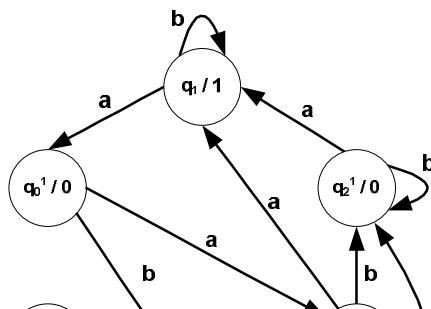
Konversi untuk state Q_1 :



Konversi untuk state Q_2 :



Konversi untuk state Q_3 :



8.5. Latihan Soal

1. Amatilah lingkungan hidup anda sehari-hari. Tentukan sebuah obyek (misalnya, sistem lift, sistem traffic light, sistem perpanjangan STKB, dll) yang anda dapat gambarkan/modelkan dengan Automata.

2. Lakukan penelusuran input-input string berikut pada FSD di atas :

- | | | |
|--------|-----------|---------|
| a. 10 | c. 110 | e. 0010 |
| b. 010 | e. 110010 | f. 001 |

Carilah RE dan DFA yang dibentuk oleh bahasa $L_1 \cap L_2$ berikut :

L_1	L_2

3. $(a\ b)^* a$	$b (a + b)^*$
4. $(a + b) b (a + b)^*$	$b (a + b)^*$
5. $(a + ab)^* (a + \lambda)$	$(a + ab)^* a$
6. $(ab^*)^*$	$(a + b)^* aa (a + b)^*$

Tentukan apakah kedua bahasa di bawah bersifat regular atau non regular :

7. Kuadrat Ganda = $\{ a^{n^2}b^n, \text{ dengan } n = 1, 2, 3, \dots \}$
8. $\{ a^n b^{n+1} \} = \{ abb, aabbb, aaabbbb, \dots \}$

BAB 9. NONDETERMINISM

Pada saat kita menyusun, meng-compile dan menjalankan sebuah program pada sebuah komputer, maka pikiran yang ada di benak kita adalah bahwa kita sedang memberikan kepercayaan kepada komputer. Kita berharap bahwa 'makhluk' komputer akan selalu mampu menyelesaikan persoalan-persoalan yang kita berikan. Karena komputer memang memiliki kemampuan menyelesaikan permasalahan-permasalahan komputasional berukuran besar dengan cepat. Tetapi yang seringkali kita lupa adalah bahwa bagaimanapun komputer adalah tetap benda mati. Komputer tidak memiliki intelegensia seperti manusia.

Oleh karenanya menjadi tugas kita untuk merancang dan memberikan 'kepandaian' kepada komputer, agar komputer dapat melakukan segala sesuatu seperti yang kita harapkan. Dan ini tentunya bukan tugas yang mudah bagi kita. Kita harus mampu merancang 'kepandaian' tersebut sedemikian hingga mampu dipahami dan diadaptasi oleh komputer.

Sebuah mesin/komputer hanya mengenal 'hitam' dan 'putih'. Komputer tidak mengenal 'abu-abu'. Komputer hanya bisa memilih sejauh pilihan-pilihan yang tersedia sangat jelas perbedaannya. Dia tidak memiliki indra perasa atau insting. Oleh karenanya komputer tidak dapat/boleh berada dalam situasi 'harus mempertimbangkan'.

Sejauh ini kita sudah melihat contoh-contoh beragam persoalan yang dapat dimodelkan oleh mesin finite automata (FA). Dimana FA yang kita gunakan masih bersifat deterministik (DFA). Namun dalam kenyataannya ada, bahkan banyak, permasalahan-permasalahan yang tidak dapat dimodelkan dengan menggunakan mesin dari kelas DFA. Oleh karena itu kita akan mempelajari kelas mesin di atas DFA. Sebuah kelas mesin yang mampu mengakomodasi alternatif pilihan yang tidak pasti.

9.1. Determinism vs Nondeterminism

Pada bagian sebelumnya telah disinggung bahwa terdapat sebuah varian/pengembangan dari DFA yang disebut transition graph (TG). Dimana terdapat beberapa bagian dari mesin tersebut yang menyebabkan munculnya perdebatan apakah mesin dari kelas TG masih termasuk jenis mesin yang bersifat deterministik?

Jika pengertian deterministik mengacu kepada adanya kepastian mengenai arah/pola pergerakan kendali pada mesin, maka konsepsi dasar dari istilah non deterministik secara mudah dapat diartikan sebagai terdapat situasi-situasi dimana arah pergerakan kendali mesin menjadi tidak pasti. Dan pada sebuah mesin finite automata, arah pergerakan ditentukan oleh garis berarah (arc) dari satu state ke state yang lain. Segala sesuatunya akan sangat pasti jika hanya terdapat satu pilihan jalur keluar dari sebuah state untuk sebuah input karakter tertentu. Tetapi sebaliknya, akan dapat membingungkan jika pilihan tersebut menjadi lebih dari satu. Dan inilah yang diperbolehkan dan mungkin terjadi pada mesin TG. Dimana diperkenankan adanya lebih dari satu outgoing arc yang keluar dari sebuah state dan memiliki label yang sama.

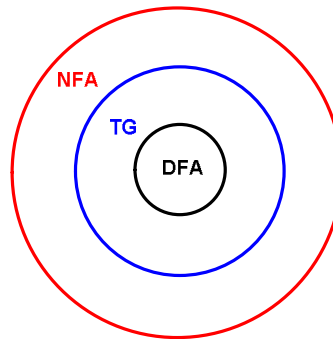
9.2. Nondeterministic Finite Automata (NFA)

secara definitif, NFA tidak berbeda jauh jika dibandingkan DFA maupun TG. Adapun komponen-komponen NFA secara lengkap (sekaligus menunjukkan perbedaannya dengan kedua mesin sebelumnya) adalah:

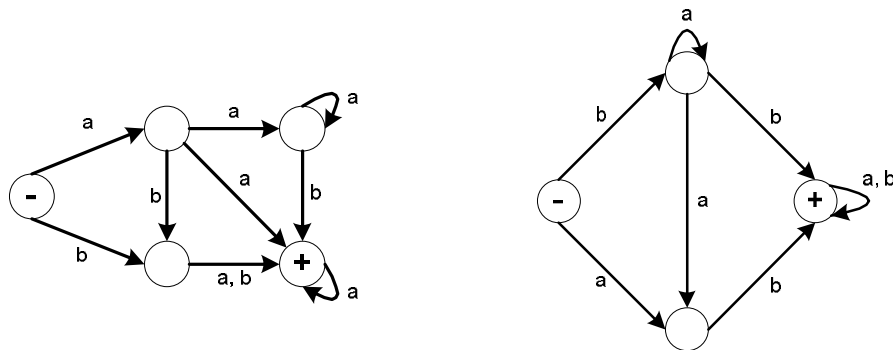
1. S sebagai himpunan berhingga state untuk media perpindahan kendali mesin;
2. Σ sebagai himpunan berhingga alphabet untuk input karakter;
3. s_0 adalah salah satu state dari himpunan S yang diperlakukan sebagai start state;
4. s_n adalah salah satu state dari himpunan S yang diperlakukan sebagai final state (DFA dapat memiliki lebih dari satu final state);
5. δ sebagai himpunan berhingga fungsi transisi untuk memindahkan kendali mesin. Dimungkinkan adanya lebih dari satu outgoing edge dengan label sama yang keluar dari sebuah state.

Dari uraian di atas terlihat bahwa perbedaan NFA dengan TG adalah pada NFA dimungkinkan memiliki lebih dari satu final state. Sementara TG hanya boleh memiliki lebih dari satu start state. Dan

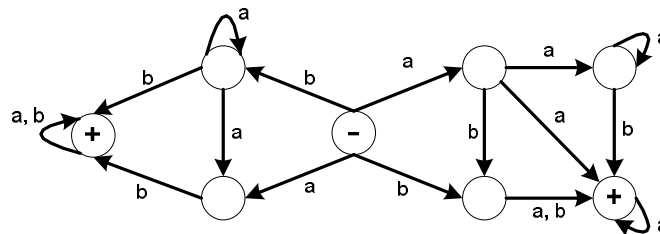
DFA hanya boleh memiliki masing-masing satu start dan final state saja. Sehingga dapat kita ilustrasikan posisi dari masing-masing mesin yang telah kita kenal mungkin seperti berikut:



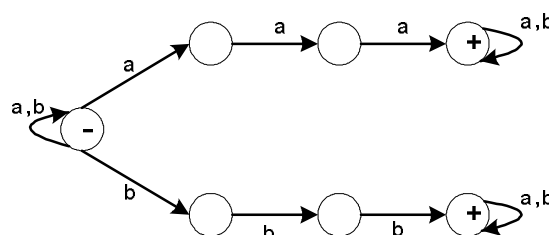
Dengan segala keleluasaan yang dimilikinya mestinya akan lebih mudah mendefinisikan bahasa dengan menggunakan NFA (daripada DFA dan TG). Sebagai contoh misalkan terdapat 2 bahasa r_1 dan r_2 yang masing-masing berkorespondensi dengan DFA_1 dan DFA_2 :



Untuk melakukan operasi union kepada kedua DFA di atas, mungkin bukan hal mudah. Setidaknya memerlukan waktu. Namun jika hasil operasi union tidak diharuskan dalam bentuk DFA, maka kita dapat mengerjakannya dengan sangat mudah, dan hasilnya akan berbentuk NFA:

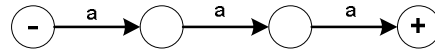


Contoh lain, misal kita akan mendefinisikan sebuah bahasa yang dapat menerima semua string yang mengandung substring aaa atau bbb. Maka melalui NFA, bahasa tersebut dapat didefinisikan sebagai berikut :

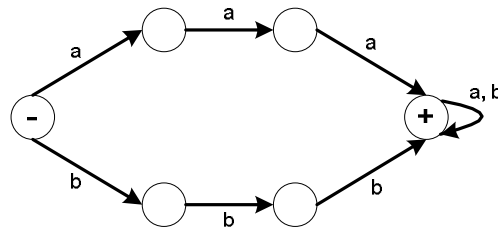


Sementara jika menggunakan DFA, maka hasilnya pun akan tampak lebih kompleks. Dengan tahapan pengerjaan seperti berikut:

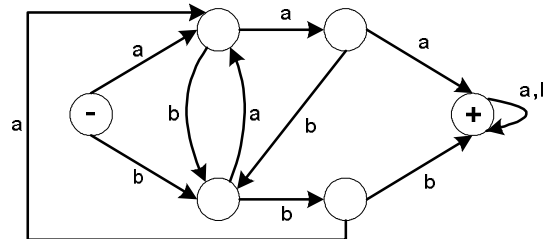
Untuk substring : aaa



Untuk substring : $aaa(a + b)^*$ atau $bbb(a + b)^*$



Untuk substring : $[(a + b)^* aaa (a + b)^* + (a + b)^* bbb (a + b)^*]$



9.3. Konversi NFA menjadi DFA

Jika sebuah permasalahan pada akhirnya hanya dapat dimodelkan dengan menggunakan NFA, maka itu belum dapat dikatakan solusi final bagi kita. Karena bagaimanapun tujuan kita adalah dapat membawa proses penyelesaian masalah ke dalam 'jalur' komputasi. Dimana untuk selanjutnya komputer yang akan memproses solusinya bagi kita. Artinya adalah kita tetap dituntut mampu merepresentasikan model yang kita buat ke dalam bentuk yang dapat diterima oleh komputer. Oleh karena itu kita memerlukan panduan agar model non deterministik yang kita punya dapat ditransformasikan ke dalam bentuk yang deterministik.

Secara umum proses konversi model dalam bentuk NFA menjadi model dalam bentuk DFA tidaklah sulit. Beberapa operasi yang digunakan dalam proses konversi antara lain :

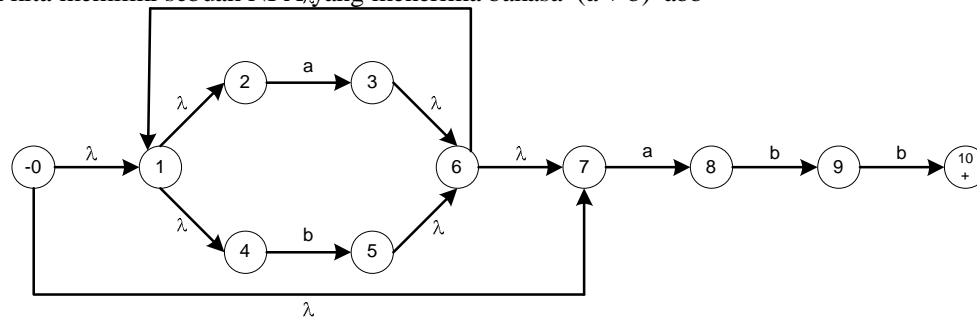
Jenis Operasi	Keterangan
λ -closure(s)	Himpunan state pada NFA yang dapat dikunjungi dari state S melalui input karakter λ
λ -closure(T)	Himpunan state pada NFA yang dapat dikunjungi dari satu/lebih state pada sub-

	himpunan T melalui karakter λ
Move(T, a)	Himpunan state pada NFA yang dapat dikunjungi melalui input karakter a dari satu/lebih state pada sub-himpunan T

Aplikasi dari ketiga operasi di atas akan lebih mudah kita pahami melalui contoh soal berikut.

Contoh :

Misalkan kita memiliki sebuah NFA yang menerima bahasa $(a + b)^*abb$



Proses penyelesaian masalah di atas akan kita lakukan secara bertahap dan diawali dari start state. Dimana tujuan kita adalah mencari kelompok-kelompok state yang nantinya masing-masing kelompok akan diwakili oleh sebuah state pada DFA yang akan kita bangun.

$$\lambda\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A$$

Dengan operasi di atas akan dilakukan penelusuran dari start state ke state-state lain yang dapat dikunjungi melalui arc berlabel λ . Dan kelompok state yang dapat kita kunjungi akan diberi nama kelompok A.

Dan jika A diberi input a, maka akan menghasilkan

$$\text{Move}(A, a) = \{3, 8\}$$

Dan dari sub-himpunan $\{3, 8\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$$\lambda\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B$$

Sementara jika A diberi input b, maka akan menghasilkan

$$\text{Move}(A, b) = \{5\}$$

Dan dari sub-himpunan $\{5\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$$\lambda\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C$$

Sejauh ini A, B, dan C bukan merupakan himpunan yang identik. Dan untuk seterusnya, kita akan memberi nama himpunan yang baru terbentuk, selama himpunan baru tersebut berbeda dengan himpunan-himpunan yang telah ada.

Sekarang jika B diberi input a, maka akan menghasilkan

$$\text{Move}(B, a) = \{3, 8\}$$

Dan dari sub-himpunan $\{3, 8\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$$\lambda\text{-closure}(\{3, 8\}) = B$$

Ternyata himpunan yang dihasilkan oleh operasi $\lambda\text{-closure}(\{3, 8\})$ sama dengan himpunan yang telah ada sebelumnya, yaitu B.

Dan jika B diberi input b, maka akan menghasilkan

$\text{Move}(B, b) = \{5, 9\}$

Dan dari sub-himpunan $\{5, 9\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{5, 9\}) = \{1, 2, 4, 5, 6, 7, 9\} = D$

Jika C diberi input a, maka akan menghasilkan

$\text{Move}(C, a) = \{3, 8\}$

Dan dari sub-himpunan $\{3, 8\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{3, 8\}) = B$

Jika C diberi input b, maka akan menghasilkan

$\text{Move}(B, b) = \{5\}$

Dan dari sub-himpunan $\{5\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{5\}) = C$

Jika D diberi input a, maka akan menghasilkan

$\text{Move}(D, a) = \{3, 8\}$

Dan dari sub-himpunan $\{3, 8\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{3, 8\}) = B$

Jika D diberi input b, maka akan menghasilkan

$\text{Move}(D, b) = \{5, 10\}$

Dan dari sub-himpunan $\{5, 10\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\} = E$

Jika E diberi input a, maka akan menghasilkan

$\text{Move}(E, a) = \{3, 8\}$

Dan dari sub-himpunan $\{3, 8\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{3, 8\}) = B$

Jika E diberi input b, maka akan menghasilkan

$\text{Move}(E, b) = \{5\}$

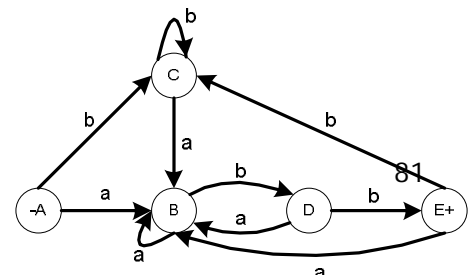
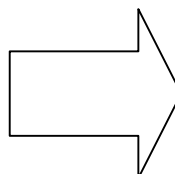
Dan dari sub-himpunan $\{5\}$ ini kita akan menelusuri state-state lain dengan melalui arc berlabel λ , dan dihasilkan:

$\lambda\text{-closure}(\{5\}) = C$

Kita akan mengerjakan proses di atas sampai tidak ditemukan himpunan state baru. Dan pada contoh di atas ternyata himpunan E adalah kelompok state terakhir yang kita temukan. Dan DFA kita akan memiliki 5 state, yaitu $\{A, B, C, D, E\}$.

Jika kita gambarkan kelima state tersebut (beserta proses pemunculannya berdasarkan input karakter a atau b), maka akan terbentuk DFA seperti berikut:

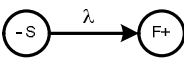
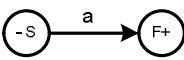
	a	b
- A	B	C



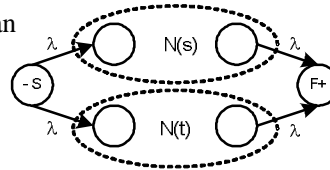
B	B	D
C	B	C
D	B	E
+ E	B	C

9.4. Konversi RE menjadi NFA

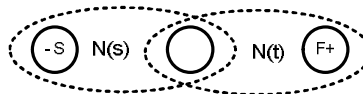
Dalam bagian ini pula disertakan panduan untuk melakukan konversi sebuah bahasa yang dinyatakan dengan menggunakan regular expression (RE) menjadi berbentuk NFA. Beberapa situasi yang mungkin akan kita jumpai selama proses konversi tersebut dan penanganannya antara lain seperti berikut:

1. Untuk ekspresi λ : 
2. Untuk karakter **a** : 
3. Misal N(s) dan N(t) yang masing-masing adalah NFA untuk RE s dan t:

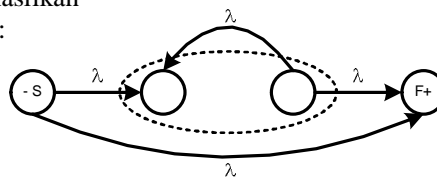
- a. Operasi **UNION** $s|t$ akan menghasilkan NFA yang bersesuaian $N(s|t)$:



- b. Operasi **CONCATENATE** st menghasilkan NFA yang bersesuaian $N(st)$:



- c. Operasi **CLOSURE** s^* menghasilkan NFA yang bersesuaian $N(s^*)$:



- d. Untuk ekspresi **(s)** tidak menghasilkan perubahan apapun pada NFA asal.

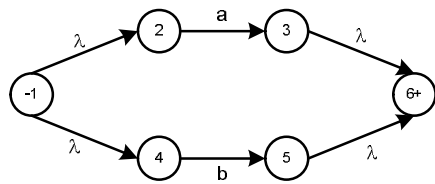
Sebagai contoh mari kita lihat bagaimana proses konversi yang terjadi, seandainya kita memiliki sebuah bahasa yang dinyatakan dalam bentuk RE seperti berikut: $r = (a + b)^* abb$

Untuk melakukan konversi, maka kita akan lakukan bertahap dengan jalan mengkonversi RE suku per suku.

Untuk $r_1 = a$ dan $r_2 = b$

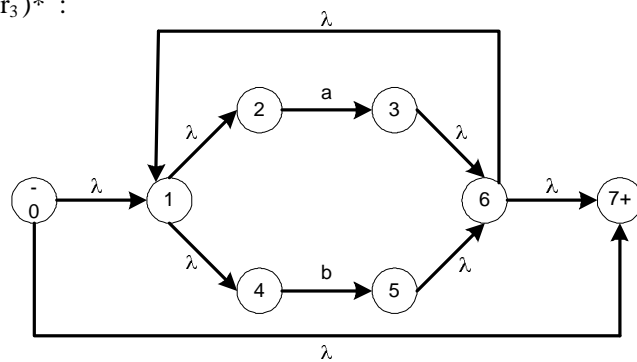


Untuk $r_3 = r_1 + r_2$



Untuk (r_3) , maka NFA tidak berubah, sama dengan yang di atas.

Untuk $(r_3)^*$:

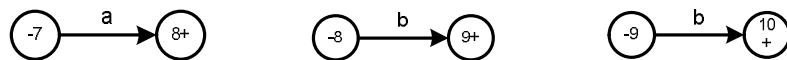


Untuk

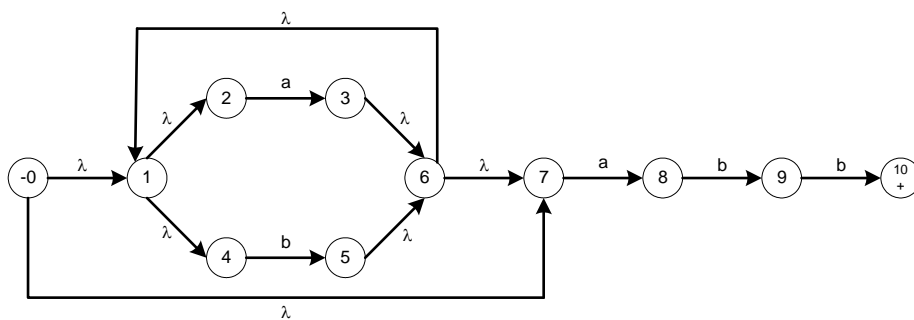
$r_4 = a$

$r_5 = b$

$r_6 = b$

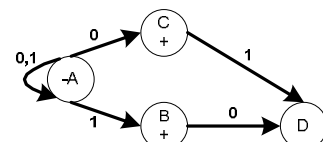
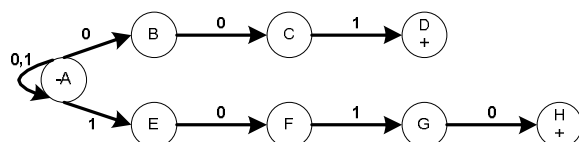


Sehingga jika $(r_3)^* r_4 r_5 r_6$ akan menghasilkan NFA seperti berikut :

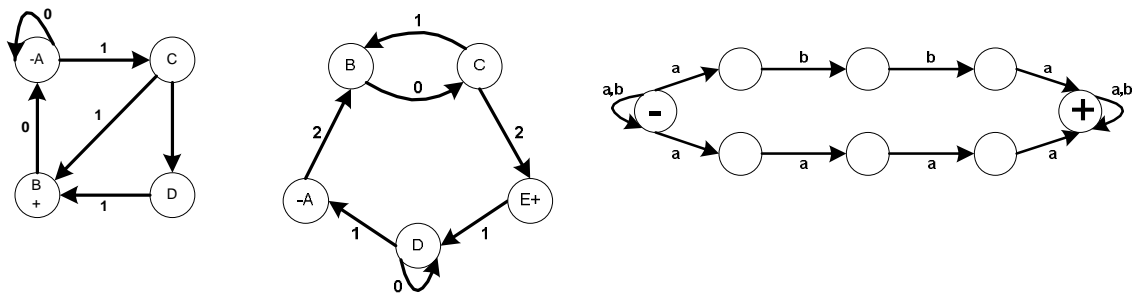


9.5. Latihan Soal

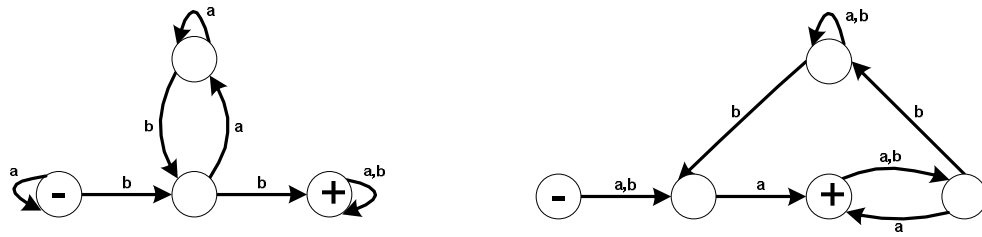
1. Konversikan NFA berikut menjadi DFA :



2. Deskripsikan bahasa yang diterima oleh NFA berikut :



3. Carilah NFA yang ekivalen (dengan state lebih sedikit) dengan DFA berikut :



BAB 10. CONTEXT-FREE LANGUAGE

Sebenarnya selain finite automata, terdapat pula alat pemodelan lain yang dapat digunakan untuk merancang sebuah bahasa (pemrograman). Jika finite automata lebih cenderung merepresentasikan model komputasi dari bahasa yang kita bangun, maka alat pemodelan yang akan kita pelajari berikut justru merupakan alat pemodelan yang lazim dan banyak digunakan untuk mengembangkan bahasa-bahasa pemrograman yang ada di dunia ini. Alat pemodelan ini dikenal dengan sebutan grammar.

Grammar telah dikenal memiliki kemampuan memodelkan berbagai macam karakteristik bahasa pemrograman. Dan melalui grammar kita dapat mengenalkan atau memberikan panduan kepada pengguna mengenai tata bahasa (khususnya yang berkaitan dengan sintaksis) dari bahasa pemrograman kita. Dan lebih jauh pada kompilator (compiler), grammar memainkan peran yang sangat vital, yaitu berfungsi sebagai pedoman atau tolok ukur untuk memeriksa kebenaran penulisan program yang ditulis oleh programmer.

Secara konseptual grammar didefinisikan sebagai sebuah sistem matematis yang dapat digunakan untuk mendefinisikan bahasa. Sementara secara formal sebuah grammar G didefinisikan memiliki 4 tuple (V_N, V_T, S, θ) dimana V_N adalah himpunan berhingga non-terminal, V_T adalah himpunan berhingga terminal, S adalah salah satu anggota V_N yang dijadikan start symbol, dan θ adalah himpunan berhingga production yang berbentuk $\alpha \rightarrow \beta$

(dimana α adalah salah satu simbol dari himpunan V_N dan β berbentuk rangkaian *terminal* dan/atau *non-terminal*).

Mungkin akan lebih mudah jika kita lihat contoh pendefinisian sebuah grammar sederhana seperti berikut:

Misal terdapat sebuah grammar $G = (V_N, V_T, S, \phi)$ yang berfungsi untuk mendefinisikan pembentukan identifier pada sebuah bahasa pemrograman:

Sub-himpunan nonterminal : $V_N = \{I, L, D\}$

Sub-himpunan terminal : $V_T = \{a, b, c, \dots, z, 0, 1, 2, \dots, 9\}$

Sub-himpunan start symbol : $S = I$

Sub-himpunan production :

$$\phi = \{ I \rightarrow L, I \rightarrow IL, I \rightarrow ID, L \rightarrow a, L \rightarrow b, \dots, L \rightarrow z, \\ D \rightarrow 0, D \rightarrow 1, D \rightarrow 2, \dots, D \rightarrow 9 \}$$

Grammar di atas jika dicermati terlihat memiliki 3 nonterminal, yaitu I , L , dan D . Nonterminal ini merupakan bentuk antara dimana dalam prosesnya nanti nilainya akan (harus) berubah menjadi rangkaian terminal. Untuk lebih mudahnya anda dapat menganggap nonterminal ini seperti layaknya sebuah variabel pada persamaan matematika. Sementara untuk terminal, grammar di atas menyediakan pilihan 26 huruf dan 10 angka yang dapat dipilih dan dikombinasikan satu dengan lainnya untuk menyatakan identifier.

Selain itu terdapat pula sebuah start symbol yang berfungsi sebagai penanda untuk mengawali proses pengubahan rangkaian nonterminal untuk dijadikan rangkaian terminal semua. Dan terakhir adalah uraian production yang berfungsi sebagai pemandu proses pengubahan tersebut. Bagaimana dan apa yang diperbolehkan serta yang tidak diperkenankan, semua diatur di dalam himpunan production.

Grammar untuk pembentukan identifier di atas dapat pula diekspresikan/dinyatakan dalam bentuk lain seperti berikut :

$$\begin{array}{lll} I \rightarrow L & L \rightarrow a & D \rightarrow 0 \\ I \rightarrow IL & L \rightarrow b & D \rightarrow 1 \\ I \rightarrow ID & \dots & \dots \\ & L \rightarrow z & D \rightarrow 9 \end{array}$$

Pernyataan grammar di atas terlihat lebih kompak dan lebih enak dilihat. Sepintas ekspresi grammar tersebut hanya menyatakan rangkaian production-nya saja. Namun itu tidak menjadi masalah. Karena melalui konsensus tertentu, kita dengan mudah dapat mengenali komponen-komponen sebuah

grammar yang ditulis dengan gaya seperti di atas. Biasanya non-terminal ditulis dengan huruf besar. Selain itu (angka, karakter, simbol, tanda baca dan huruf kecil) adalah terminal, dan ditulis dengan huruf kecil dan/atau tebal. Dan yang menjadi start symbol adalah nonterminal yang berada pada posisi kiri-atas (dalam hal ini adalah “I”).

Atau jika penulisan di atas masih dianggap masih terlalu panjang, kita masih diperbolehkan untuk menulis secara singkat:

$$\begin{aligned} I &\rightarrow L \mid IL \mid ID \\ L &\rightarrow a \mid b \mid \dots \mid z \\ D &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Dimana penulisan production-production alternatif yang berasal dari nonterminal yang sama dapat dituliskan secara berurutan dengan dipisahkan oleh vertical bar (“|”).

Contoh lain pendefinisikan grammar, misalkan terdapat sebuah grammar $G = (V_N, V_T, S, \phi)$ untuk pembentukan bilangan bulat positif, dimana:

$V_N = \{ \text{ANGKA}, \text{DIGIT_AWAL}, \text{DIGIT_LAIN} \}$

$V_T = \{ 0, 1, 2, \dots, 9 \}$

$S = \text{ANGKA}$

$\phi = \{ \text{ANGKA} \rightarrow \text{DIGIT_AWAL}, \text{DIGIT_AWAL} \rightarrow \text{DIGIT_AWAL DIGIT_LAIN}, \\ \text{DIGIT_AWAL} \rightarrow 1, \text{DIGIT_AWAL} \rightarrow 2, \dots, \text{DIGIT_AWAL} \rightarrow 9, \\ \text{DIGIT_LAIN} \rightarrow 0, \text{DIGIT_LAIN} \rightarrow 1, \dots, \text{DIGIT_LAIN} \rightarrow 9 \}$

Grammar di atas dapat ditulis lebih ringkas dengan hanya menyatakan himpunan production seperti berikut:

$\text{ANGKA} \rightarrow \text{DIGIT_AWAL}$

$\text{DIGIT_AWAL} \rightarrow \text{DIGIT_AWAL DIGIT_LAIN} \mid 1 \mid 2 \mid \dots \mid 9$

$\text{DIGIT_LAIN} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

10.1. Derivasi dan Parsing

Derivasi atau parsing adalah sebuah metode/cara untuk mengidentifikasi keanggotaan sebuah bahasa. Melalui derivasi atau parsing dapat diuji keanggotaan sebuah kata/string atau sentence/kalimat terhadap sebuah bahasa pemrograman tertentu.

Disebut derivasi atau parsing karena prosesnya adalah men-derive (menurunkan) sebuah string/kalimat yang di-input-kan dari start symbol menuju ke rangkaian terminal yang sama persis dengan string/kalimat input.

Dalam aplikasinya, derivasi menggunakan pendekatan substitusi linier, yakni dengan mengganti setiap nonterminal dengan terminal yang bersesuaian. Sedangkan parsing menggunakan ‘pohon parsing’ (parse tree) untuk membantu proses substitusi tersebut.

Sebuah program akan dinyatakan benar secara sintaktik adalah apabila dari start symbol sebuah grammar bahasa pemrograman dapat dibentuk sebuah parse tree dimana leaf-nya adalah semua string & karakter dari program tersebut.

Untuk memperjelas maksud uraian di atas, maka berikut akan kita aplikasi derivasi pada sebuah bahasa yang didefinisikan melalui grammar berikut :

$$\begin{aligned} I &\rightarrow L \mid IL \mid ID \\ L &\rightarrow a \mid b \mid \dots \mid z \\ D &\rightarrow 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Proses penurunan/derivasi dapat dilakukan pada string-string inputan untuk menentukan keanggotaan string-string tersebut terhadap bahasa di atas. Misalkan sebagai input diberikan string : a15

Terdapat 2 pendekatan untuk melakukan derivasi, yaitu dengan leftmost atau rightmost derivation. Leftmost derivation akan selalu melakukan proses penurunan (substitusi nonterminal) berdasarkan posisi nonterminal paling kiri dari setiap rangkaian yang akan diderivasi.

Leftmost Derivation

$I \Rightarrow \underline{I}D \Rightarrow \underline{I}DD \Rightarrow \underline{L}DD \Rightarrow a\underline{D}D \Rightarrow a1\underline{D} \Rightarrow a15$

Sementara rightmost derivation akan selalu melakukan proses penurunan (substitusi nonterminal) berdasarkan posisi nonterminal paling kanan dari setiap rangkaian yang akan diderivasi.

Rightmost Derivation

$I \Rightarrow \underline{I}D \Rightarrow \underline{I}5 \Rightarrow \underline{ID}5 \Rightarrow \underline{I}15 \Rightarrow \underline{L}15 \Rightarrow a15$

Dari kedua proses derivasi (leftmost dan rightmost) dapat dilihat bahwa keduanya dapat menghasilkan string "a15" (sama dengan string yang menjadi inputan). Sehingga dapat disimpulkan bahwa string "a15" adalah penulisan variabel/identifier yang diperbolehkan/benar.

Sebagai pedoman awal, proses derivasi selalu diawali dari start symbol. Tetapi jika start symbol-nya tidak diketahui secara pasti, maka non-terminal yang berada pada ujung kiri atas dari grammar yang dijadikan acuan dapat dianggap sebagai start symbol.

Sementara untuk melihat proses parsing, kita akan gunakan sebuah grammar yang mengekspresikan persamaan matematis sederhana seperti berikut:

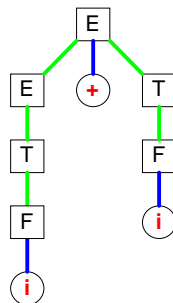
$$E \rightarrow T \mid E + T \mid E - T$$

$$T \rightarrow F \mid T * F \mid T / F$$

$$F \rightarrow i \mid (E)$$

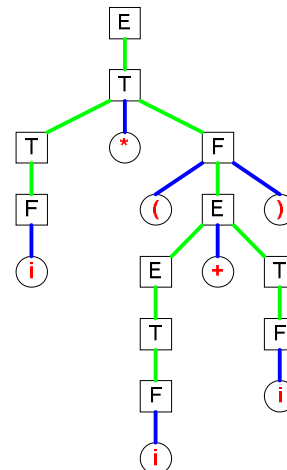
Jika diberikan inputan ekspresi : $i + i$

Maka pohon parsing yang dihasilkan adalah:



Jika diberikan inputan ekspresi : $i * (i + i)$

maka pohon parsing yang dihasilkan adalah:



Kedua pohon parsing di atas memiliki karakteristik yang sama, yaitu sama-sama memiliki root (dimana masing-masing root adalah start symbol). Keduanya juga memiliki intermediate node (berupa nonterminal), dan leaf (ditunjukkan dengan bagan lingkaran) yang berisi karakter terminal yang kebetulan sama dengan urutan karakter yang membentuk ekspresi input.

Namun adakalanya leaf yang dihasilkan oleh pohon parsing ini memang tidak bisa sama persis dengan ekspresi inputnya. Dan dalam kasus seperti ini, bukan pohon parsingnya yang salah. Melainkan penulisan ekspresi inputnya yang salah. Karena proses parsing tidak dapat menghasilkan rangkaian leaf yang tidak sama dengan inputan.

10.2. Klasifikasi Grammar

Sintaks bahasa pemrograman umumnya dinyatakan melalui grammar, yg secara garis besar dibagi menjadi 2 klas utama, yaitu :

1. Backus-Naur Form (BNF)

Sebuah meta-language yang dikembangkan oleh Johan Backus & Peter Naur.

Didalam perkembangannya, cakupan BNF ini diperluas & cara mengekspresikannya pun dirampingkan menjadi EBNF (*Extended Backus-Naur Form*).

Dalam BNF, aturan penulisan production-nya antara lain adalah:

Non-terminal	ditulis	<non-terminal>
Terminal	ditulis	terminal
Simbol “ \rightarrow ”	ditulis	::=

Contoh:

Grammar bahasa Pascal yang kali pertama ditulis oleh Niclaus Wirth menggunakan format BNF.

```
<identifier> ::= <letter> | <identifier> <letter> |  
               <identifier> <digit>  
<letter> ::= a | b | c | ... | z  
<digit> ::= 0 | 1 | 2 | ... | 9  
dst...
```

2. Chomsky Normal Form (CNF)

Terbagi ke dalam 2 sub-klas :

- **Unrestricted Grammar (grammar kelas 0)**

Aturan-aturan sintaktik (production) yang digunakan untuk membentuk kalimat tidak mempunyai batasan yang jelas.

Contoh :

$G = (\{S, A, B, C, D\}, \{a, b\}, S, \theta)$, dengan θ adalah :

$S \rightarrow CD$	$Aa \rightarrow aA$	$C \rightarrow e$
$C \rightarrow aCA \mid bCB$	$Ab \rightarrow bA$	$D \rightarrow e$
$AD \rightarrow aD$	$Ba \rightarrow aB$	
$BD \rightarrow bD$	$Bb \rightarrow bB$	

Bahasa yang didefinisikan oleh grammar di atas adalah :

$$L(G) = \{ ww \mid w \in \{a, b\}^* \}$$

- **Restricted Grammar**, yang terdiri dari 3 sub sub-klas :

- **Context-Sensitive Grammar (grammar kelas 1)**

Grammar dengan production berbentuk $\alpha \rightarrow \beta$, dimana $|\alpha| \leq |\beta|$

Contoh :

$G = (\{S, A, B, C, D\}, \{a, b\}, S, \theta)$, dengan θ adalah :

$S \rightarrow aSBC \mid aBC$	$bB \rightarrow bb$	$bC \rightarrow bc$	$CB \rightarrow BC$	$cC \rightarrow cc$
-------------------------------	---------------------	---------------------	---------------------	---------------------

Misal diberi input string $a^2b^2c^2$, maka proses derivasi akan tampak seperti berikut :

$S \Rightarrow aSBC \Rightarrow aabCBC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$

- **Context-Free Grammar (grammar kelas 2)**

Grammar dengan production yang berbentuk $\alpha \rightarrow \beta$, dimana $|\alpha| \leq |\beta|$ dengan $\alpha \in V_n$ dan $|\alpha| = 1$. Dengan demikian, production-production pada klas grammar ini hanya memiliki satu non-terminal di sisi kiri setiap production-nya.

Bahasa yang didefinisikan oleh CFG ini disebut sebagai Context-Free Language (CFL).

CFG merupakan satu-satunya klas grammar yang telah memiliki algoritma parsing yang optimal. Sehingga hampir semua bahasa pemrograman menggunakan CFG untuk mendefinifikan aturan-aturan sintaktik bahasanya.

Contoh :

Bahasa = $\{ a^n b a^n \mid n \geq 1 \}$ didefinisikan melalui grammar berikut :

$$\begin{aligned} S &\rightarrow aCa \\ C &\rightarrow aCa \mid b \end{aligned}$$

Derivasi untuk input string $a^3 b a^3$ adalah sebagai berikut :

$$S \Rightarrow aCa \Rightarrow aaCaa \Rightarrow aaaCaaa \Rightarrow aaabaaa$$

o Regular Grammar (grammar kelas 3)

Grammar dengan production yang berbentuk $\alpha \rightarrow \beta$, dimana $|\alpha| \leq |\beta|$ dengan $\alpha \in V_n$ dan $|\alpha| = 1$. Sedangkan β mempunyai bentuk aB atau a ($a \in V_T$ dan $B \in V_N$).

Bahasa yang didefinisikan oleh Regular Grammar ini disebut Regular Language.

Bahasa pemrograman yang menggunakan aturan sintaktik bahasa regular ini antara lain adalah javascript, perl, dan lain-lain.

Contoh :

Bahasa = $\{ a^n b a^m \mid n \geq 1 \}$ didefinisikan melalui grammar berikut :

$$\begin{aligned} S &\rightarrow aS \mid aB \\ C &\rightarrow aC \mid a \\ B &\rightarrow bC \end{aligned}$$

Derivasi untuk input string $a^3 b a^2$ adalah sebagai berikut :

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaB \Rightarrow aaabC \Rightarrow aaabaC \Rightarrow aaabaa$$

15.3. Ambiguitas

Sebuah grammar dikatakan memiliki sifat ambiguous (atau singkatnya dapat ditulis ambigu) apabila grammar tersebut dapat menghasilkan lebih dari satu parse tree (derivasi) utk kalimat yg sama.

Sebagai contoh perhatikan grammar berikut:

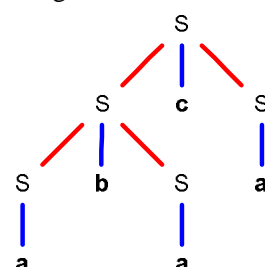
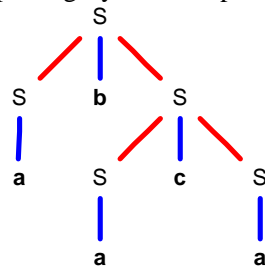
$$S \rightarrow SbS \mid ScS \mid a$$

Dimana untuk kalimat “abaca” dapat dibentuk 2 derivasi yang berbeda, yaitu :

$$S \Rightarrow SbS \Rightarrow SbScS \Rightarrow SbSca \Rightarrow Sbaca \Rightarrow abaca$$

$$S \Rightarrow ScS \Rightarrow SbScS \Rightarrow abScS \Rightarrow abacS \Rightarrow abaca$$

Pun dengan parsing-nya. Kita dapat membentuk 2 pohon parsing berbeda untuk kalimat “abaca”:



Menciptakan grammar yang ambiguity-free menjadi isu penting dalam perancangan bahasa pemrograman atau compiler. Selama proses parsing/derivasi, compiler akan berusaha semaksimal mungkin mendapatkan rangkaian terminal yang sama persis dengan susunan source program. Maka untuk menghemat waktu kompilasi dan menghindarkan komputer dari kesalahan, maka harus berupaya untuk membatasi jumlah alternatif production yang muncul pada setiap langkah proses derivasi atau parsing. Utamanya mencegah agar tidak sampai muncul pilihan bentuk derivasi/parsing untuk sebuah kalimat/string inputan.

Namun sayangnya sampai saat ini belum ada metode yang dapat digunakan untuk menghilangkan sifat ambiguitas pada grammar. Yang ada hanyalah metode untuk meminimalkan kemungkinan munculnya sifat ambiguitas pada grammar yang kita rancang. Metode-metode yang dimaksud antara lain adalah :

1. Rekursif Kiri

Sebuah grammar dikatakan memiliki sifat rekursif kiri apabila terdapat derivasi $\alpha \Rightarrow \alpha S_i$ (untuk satu atau lebih S_i).

Perubahan yang harus dilakukan adalah sebagai berikut :

Untuk production yang berbentuk :

$$\alpha \rightarrow \alpha S_1 \mid \alpha S_2 \mid \dots \mid \alpha S_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

(dan tidak β_i yang tidak diawali α)

Harus diubah menjadi bentuk-bentuk production :

$$\begin{aligned}\alpha &\rightarrow \beta_1 \alpha^1 \mid \beta_2 \alpha^1 \mid \dots \mid \beta_n \alpha^1 \\ \alpha^1 &\rightarrow S_1 \alpha^1 \mid S_2 \alpha^1 \mid \dots \mid S_m \alpha^1 \mid \varepsilon\end{aligned}$$

Misal terdapat sebuah grammar seperti berikut:

$$\begin{aligned}E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id\end{aligned}$$

Dapat dilihat bahwa terdapat bentuk rekursif kiri pada grammar tersebut, yaitu pada production berikut :

$$\begin{aligned}E &\rightarrow E + T \\ T &\rightarrow T * F\end{aligned}$$

Maka dengan menggunakan aturan perubahan rekursif kiri, grammar di atas akan dibentuk menjadi :

$$\begin{aligned}E &\rightarrow T E^1 & T &\rightarrow F T^1 & F &\rightarrow (E) \mid id \\ E^1 &\rightarrow + T E^1 \mid \varepsilon & T^1 &\rightarrow * F T^1 \mid \varepsilon\end{aligned}$$

2. Faktorisasi Kiri

Metode ini digunakan untuk meminimalkan munculnya jumlah alternatif production yang harus dipilih selama proses parsing sedang berlangsung.

Perubahan yang harus dilakukan adalah sebagai berikut :

Untuk production yang berbentuk :

$$\alpha_1 \rightarrow \alpha_2 \beta_1 \mid \alpha_2 \beta_2 \mid \dots \mid \alpha_2 \beta_m \mid \gamma$$

(γ adalah alternatif yang tidak diawali α_2)

Harus diubah menjadi bentuk-bentuk production :

$$\begin{aligned}\alpha_1 &\rightarrow \alpha_2 \alpha_1^1 \mid \gamma \\ \alpha_1^1 &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \mid \varepsilon\end{aligned}$$

contoh : terdapat sebuah grammar seperti berikut

$$S \rightarrow i E + S \mid i E + S e S \mid a$$

$$E \rightarrow b$$

Dapat dilihat bahwa terdapat 2 alternatif yang sama-sama diawali dengan $i E + S$, sehingga perubahan pada grammar dapat dilakukan seperti berikut :

$$\begin{aligned} S &\rightarrow i E + S S^1 \mid a \\ S^1 &\rightarrow e S \mid \varepsilon \\ E &\rightarrow b \end{aligned}$$

10.3. Penyederhanaan Context-Free Language

Penyederhanaan pada sistem bahasa bebas-konteks bertujuan untuk mengurangi kerumitan pada sistem tata bahasa itu sendiri. Misalnya menghilangkan kemungkinan munculnya ‘dead-end parsing’ atau proses parsing yang terlalu panjang.

Grammar-grammar berikut dapat memberikan ilustrasi yang dimaksud :

$$\begin{array}{ll} S \rightarrow A B \mid a & S \rightarrow A \\ A \rightarrow a & A \rightarrow B \\ & B \rightarrow C \\ & C \rightarrow D \\ & D \rightarrow a \mid A \end{array}$$

‘dead-end’ parsing akan terjadi pada grammar di sebelah kiri atas, dimana tidak terdapat alternatif penggantian untuk nonterminal “B”. Sedangkan pada grammar di sebelah kanan atas terlihat sangat tidak efisien karena penurunan yang dapat dilakukan sebenarnya hanya akan menghasilkan karakter “a” saja.

Metode-metode penyederhanaan yang dapat dilakukan antara lain adalah:

1. Useless Production

Yang dimaksud dengan useless production adalah:

- Production dimana terdapat non-terminal yang tidak dapat dirubah menjadi rangkaian terminal;
- Production yang tidak akan pernah tercapai melalui penurunan apapun.

Grammar berikut dapat memberikan contoh keberadaan useless production tersebut:

$$\begin{aligned} S &\rightarrow a S a \mid A b d \mid B d e \\ A &\rightarrow A d a \\ B &\rightarrow B B B \mid a \\ C &\rightarrow c \end{aligned}$$

Beberapa analisis yang dapat kita peroleh dari grammar di atas antara lain:

- Non-terminal A tidak dapat dirubah menjadi bentuk terminal seluruhnya, sehingga production $A \rightarrow A d a$ dan $S \rightarrow A b d$ dapat di-eliminasi;
- Production $C \rightarrow c$ tdk akan pernah terpakai oleh proses derivasi apapun & kapanpun, sehingga dapat di-eliminasi;

Maka grammar di atas seharusnya dapat ditulis menjadi lebih sederhana seperti berikut:

$$\begin{aligned} S &\rightarrow a S a \mid B d e \\ B &\rightarrow B B B \mid a \end{aligned}$$

2. Unit Production

Unit Production adalah production berbentuk $\alpha_1 \rightarrow \alpha_2$ yg akan menyebabkan munculnya bentuk derivasi $\alpha_1 \Rightarrow \dots \Rightarrow \alpha_2 \Rightarrow \dots$

Maka modifikasi yang dapat dilakukan adalah :

Jika terdapat $\alpha_2 \rightarrow S_1 \mid S_2 \mid \dots$

Maka ditambahkan $\alpha_1 \rightarrow S_1 \mid S_2 \mid \dots$

contoh : $S \rightarrow A \mid b b$
 $A \rightarrow B \mid b$
 $B \rightarrow S \mid a$

Unit Production yang dapat kita identifikasi adalah:

$S \rightarrow A$
 $A \rightarrow B$
 $B \rightarrow S$

Dan decent Folks yang terdapat dalam grammar di atas adalah:

$S \rightarrow b b$
 $A \rightarrow b$
 $B \rightarrow a$

Sehingga perubahan yang terjadi antara lain adalah:

$S \rightarrow A$ membentuk $S \rightarrow b$
 $S \rightarrow A \rightarrow B$ membentuk $S \rightarrow a$
 $A \rightarrow B$ membentuk $A \rightarrow a$
 $A \rightarrow B \rightarrow S$ membentuk $A \rightarrow b b$
 $B \rightarrow S$ membentuk $B \rightarrow b b$
 $B \rightarrow S \rightarrow A$ membentuk $B \rightarrow b$

Dan CFG yang baru adalah :

$S \rightarrow b b \mid b \mid a$
 $A \rightarrow b \mid a \mid b b$
 $B \rightarrow a \mid b b \mid b$

3. ϵ -Production

Sebuah non-terminal disebut nullable jika terdapat production berbentuk $\alpha \rightarrow \epsilon$ yang akan memunculkan proses derivasi $\alpha \Rightarrow \dots \Rightarrow \epsilon$.

Maka modifikasi yg dpt dilakukan adalah :

1. Menghapus semua ϵ -production;
2. Menambahkan production-production baru yg berbentuk $\alpha \rightarrow \beta$ dimana α adalah non-terminal *nullable* dan β adalah string bentukan dr sisi kanan ϵ -production setelah dikurangi non-terminal *nullable*.

contoh : $S \rightarrow a S a \mid b S b \mid \epsilon$

dimana $S \rightarrow \epsilon$ adalah ϵ -production dan S adalah *nullable*.

Maka grammar hasil modifikasi adalah :

$S \rightarrow a S a$
 $S \rightarrow b S b$
 $S \rightarrow a a$
 $S \rightarrow b b$

10.4. Chomsky Normal Form dan Greibach Normal Form

Bentuk Normal Chomsky merupakan salah satu bentuk grammar yg sangat sederhana. Bentuk ini sangat berguna untuk menyusun grammar bahasa-bahasa yang bersifat context-free (bebas konteks). Secara definitif, sebuah grammar yang dapat dikategorikan dalam bentuk Normal Chomsky adalah: Grammar dengan production yang berbentuk $\alpha \rightarrow \beta$, dimana $|\alpha| \leq |\beta|$ dengan $\alpha \in V_n$ dan $|\alpha| = 1$. Sedangkan β mempunyai bentuk a atau BB ($a \in V_T$ dan $B \in V_N$).

Selain itu Normal Chomsky juga mensyaratkan bahwa grammar di atas juga bebas dari:

- Useless Production
- Unit Production
- ϵ - Production

Contoh berikut akan memberikan ilustrasi pengaplikasian bentuk Normal Chomsky pada sebuah grammar yang telah memenuhi syarat di atas:

$S \rightarrow aB \mid CA$

$A \rightarrow a \mid bc$

$B \rightarrow BC \mid Ab$

$C \rightarrow aB \mid b$

Production-production di atas yang telah memenuhi bentuk Normal Chomsky adalah :

$S \rightarrow CA$

$A \rightarrow a$

$B \rightarrow BC$

$C \rightarrow b$

Maka untuk production-production yang belum memenuhi bentuk Normal Chomsky harus kita lakukan modifikasi dengan menambahkan nonterminal-nonterminal dummy baru agar production tersebut memenuhi bentuk Normal Chomsky, seperti berikut:

$S \rightarrow aB$ menjadi $S \rightarrow P_1 B$

$A \rightarrow bc$ menjadi $A \rightarrow P_2 P_3$

$B \rightarrow Ab$ menjadi $B \rightarrow A P_2$

$C \rightarrow aB$ menjadi $C \rightarrow P_1 B$

Modifikasi di atas memunculkan konsekuensi berupa lahirnya production-production baru, yaitu:

$P_1 \rightarrow a$

$P_2 \rightarrow b$

$P_3 \rightarrow c$

Sehingga bentuk keseluruhan grammar yang telah memenuhi bentuk Normal Chomsky adalah :

$S \rightarrow CA$ $S \rightarrow P_1 B$ $P_1 \rightarrow a$

$A \rightarrow a$ $A \rightarrow P_2 P_3$ $P_2 \rightarrow b$

$B \rightarrow BC$ $B \rightarrow A P_2$ $P_3 \rightarrow c$

Sementara **bentuk Normal Greibach** merupakan penyederhanaan lebih lanjut dari Normal Chomsky. Tujuannya masih sama, yaitu memudahkan dlm penyusunan grammar yg bersifat context-free (bebas konteks).

Secara definitif, grammar dalam bentuk Normal Greibach adalah sebagai berikut:

Grammar dg production yg berbentuk $\alpha \rightarrow a\beta$, dimana $|\alpha| \leq |\beta|$ dg $\alpha \in V_n$ & $|\alpha| = 1$. $a \in V_T$.

Sedangkan $\beta \in V_N$ & $|\beta| \geq 0$.

Grammar yang akan dibentuk menjadi Normal Greibach harus memenuhi syarat :

- Sudah dalam bentuk Normal Chomsky
- Tidak memiliki sifat Rekursif Kiri

Terdapat 2 metode untuk membentuk grammar ke dalam format Normal Greibach:

▪ Substitusi

Langkah-langkah dalam metode substitusi antara lain adalah:

1. Melakukan pengurutan terhadap semua non-terminal yang ada (mulai start symbol hingga non-terminal yang terakhir kali terjumpai);
2. Untuk setiap production dengan simbol pertama di sisi kanan berupa non-terminal, lakukan pengecekan urutannya dengan non-terminal di sisi kiri production tersebut;
3. Untuk setiap production yang tidak memenuhi urutan, lakukan modifikasi terhadap production tersebut dengan men-substitusi maju lewat production lain;
4. Melakukan substitusi mundur untuk setiap production yang belum memenuhi bentuk Normal Greibach (dimulai dari production dengan non-terminal sisi kiri yang urutannya paling belakang);

Sebagai contoh berikut kita akan mentransformasi sebgau grammar yang telah memnuhi bentuk Normal Chomsky ke dalam bentuk Normal Greibach:

$$\begin{array}{lll} S \rightarrow C A & C \rightarrow D D & B \rightarrow b \\ A \rightarrow a \mid d & D \rightarrow A B & \end{array}$$

Langkah 1 : menentukan urutan non-terminal

misalkan kita tentukan urutan : $S \rightarrow C \rightarrow A \rightarrow B \rightarrow D$

Langkah 2 : memeriksa urutan non-terminal

production yang harus diperiksa adalah :

$$\begin{array}{ll} S \rightarrow C A & \text{(telah memenuhi urutan)} \\ C \rightarrow D D & \text{(telah memenuhi urutan)} \\ D \rightarrow A B & \text{(\underline{tidak} memenuhi urutan)} \end{array}$$

Langkah 3 : melakukan substitusi maju

$$D \rightarrow A B \text{ menjadi } D \rightarrow a B \mid d B$$

Langkah 4 : melakukan substitusi mundur

$$\begin{array}{ll} C \rightarrow D D & \text{menjadi } C \rightarrow a B D \mid d B D \\ S \rightarrow C A & \text{menjadi } S \rightarrow a B D A \mid d B D A \end{array}$$

Sehingga akhirnya diperoleh hasil sebuah grammar yang memenuhi format Normal Greibach :

$$\begin{array}{ll} S \rightarrow a B D A \mid d B D A \\ A \rightarrow a \mid d \\ B \rightarrow b \\ C \rightarrow a B D \mid d B D \\ D \rightarrow a B \mid d B \end{array}$$

▪ Perkalian Matriks

Sementara pada metode perkalian matriks, langkah-langkahnya mirip dengan jika kita mencari nilai variabel pada sebuah sistem persamaan linier dengan memanfaatkan matriks. Adapun langkah-langkah dalam metode Perkalian Matriks tersebut adalah:

1. Menyatakan production-production sebagai sistem persamaan linier;
2. Mengubah sistem persamaan linier tersebut menjadi persamaan matriks $V = VR + S$
dimana : V adalah vektor baris $1 \times n$ yang berisi non-terminal sisi-kiri;
 R adalah matriks $n \times n$ yang berisi non-terminal di sisi-kanan;
 S adalah vektor baris $1 \times n$ yang berisi terminal sisi-kanan;

3. Membuat persamaan matriks $V = SQ + S$, dengan Q adalah matriks $n \times n$ yang berisi non-terminal baru. Dapatkan hasil perkalian matriks tersebut dan nyatakan dalam bentuk sistem persamaan linier;
4. membuat persamaan matriks $Q = RQ + R$. Dapatkan hasil perkalian matriks tersebut dan nyatakan dalam bentuk sistem persamaan linier;
5. Menggabungkan dan menyederhanakan kedua sistem persamaan linier yang diperoleh;
6. Menyatakan sistem persamaan linier tersebut menjadi production-production;

Aplikasi metode perkalian matriks di atas dapat kita lihat pada contoh berikut:

Misalkan terdapat sebuah grammar sederhana seperti berikut :

$$A \rightarrow BB \quad B \rightarrow a$$

Langkah 1 : membentuk sistem persamaan linier awal

$$\begin{aligned} A &= BB \\ B &= a \end{aligned}$$

Langkah 2 : membentuk persamaan matriks $V = VR + S$

$$\begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} 0 & 0 \\ B & 0 \end{bmatrix} + \begin{bmatrix} 0 & a \end{bmatrix}$$

Langkah 3 : membentuk persamaan matriks $V = SQ + S$

$$\begin{matrix} E & F \end{matrix} \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} 0 & a \end{bmatrix} \begin{bmatrix} C & D \\ E & F \end{bmatrix} + \begin{bmatrix} 0 & a \end{bmatrix}$$

Dengan hasil perkalian matriks $V = SQ + S$ adalah :

$$\begin{aligned} A &= a E \\ B &= a F + a \end{aligned}$$

Langkah 4 : membentuk persamaan matriks $Q = RQ + R$

$$\begin{bmatrix} C & D \\ E & F \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ B & 0 \end{bmatrix} \begin{bmatrix} C & D \\ E & F \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ B & 0 \end{bmatrix}$$

Dengan hasil perkalian matriks $Q = RQ + R$ adalah :

$$\begin{aligned} C &= 0 \\ D &= 0 \\ E &= B C + B \\ F &= B D \end{aligned}$$

Langkah 5 : hasil penggabungan dan penyederhaan sistem persamaan linier adalah

$$\begin{aligned} A &= a E & A &= a B \\ B &= a & \text{menjadi } B &= a \\ E &= B \end{aligned}$$

Langkah 6 : mendapatkan grammar Normal Greibach

$$\begin{aligned} A &\rightarrow a B \\ B &\rightarrow a \end{aligned}$$

10.5. Latihan Soal

1. Bahasa apakah yg didefinisikan oleh CFG berikut :

$$S \rightarrow XbaaX \mid aX$$

$$X \rightarrow Xa \mid Xb \mid \varepsilon$$

carilah sebuah contoh string yg dapat dikenali oleh CFG di atas melalui 2 derivasi berbeda.

2. Melalui grammar di bawah :
- $$\begin{aligned} S &\rightarrow E \\ E &\rightarrow T + E \mid T \\ T &\rightarrow F * T \mid F \\ F &\rightarrow (E) \mid i \end{aligned}$$

buatlah left-most derivation untuk input string :

- $((i) * (i + i)) + i$
- $(i) + ((i))$

3. Jan Lukasiewicz, seorang logician Polandia memperkenalkan notasi parenthesis-free (yg disebut Polish Notation) utk membantu proses derivasi ekspresi matematis yg umumnya menggunakan notasi infix (operator diposisikan ditengah² 2 operand), seperti contoh berikut:
 $((1 + 2) * (3 + 4) + 5) * 6$.

Jelaskan bagaimana Polish atau Lukasiewicz notation bekerja & jelaskan mengapa notasi tsb lebih mudah utk dikomputasi?

4. Tunjukkan bahwa grammar berikut memiliki sifat ambigu :

$$S \rightarrow aB \mid bA$$

$$A \rightarrow a \mid aS \mid bAA$$

$$B \rightarrow b \mid bS \mid aBB$$

5. Hilangkan useless production pada grammar berikut :

$$S \rightarrow aB \quad C \rightarrow bCb \mid a d F \mid a b$$

$$A \rightarrow b c D \mid d a C \quad F \rightarrow c F B$$

$$B \rightarrow e \mid A b$$

6. Hilangkan unit production pada grammar berikut :

$$S \rightarrow A \mid A a \quad C \rightarrow D \mid a b$$

$$A \rightarrow B \quad D \rightarrow b$$

$$B \rightarrow C \mid b$$

7. Hilangkan ε -production pada grammar berikut :

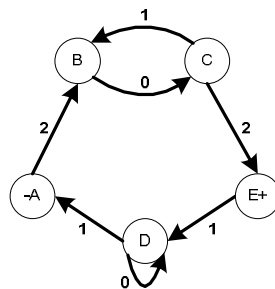
$$S \rightarrow AB$$

$$A \rightarrow a b B \mid a C a \mid \varepsilon$$

$$B \rightarrow b A \mid B B \mid \varepsilon$$

$$C \rightarrow b C b \mid a d F \mid \varepsilon$$

8. Carilah CFG yang berasosiasi dengan DFA di bawah :



9. Transformasikan grammar yang dihasilkan pada no. 8 menjadi bentuk Normal Chomsky.

10. Transformasikan grammar yang dihasilkan pada no. 9 menjadi bentuk Normal Greibach (menggunakan metode Perkalian Matriks & substitusi).

BAB 11. PUSHDOWN AUTOMATA DAN MESIN TURING

Jika bahasa reguler (regular language) yang telah kita pelajari pada beberapa bab sebelumnya berasosiasi dengan mesin yang dapat memvisualisasikan bahasa yang didefinisikan oleh regular expression (RE). Dimana mesin finite automata tersebut dapat digunakan pula untuk memeriksa keanggotaan bahasa yang bersangkutan.

Maka pada bab ini kita akan berkenalan dengan sebuah kelas mesin yang berasosiasi penggunaannya dengan bahasa bebas-konteks (context-free languages), yaitu disebut dengan Pushdown Automata (PDA).

11.1. Pushdown Automata (PDA)

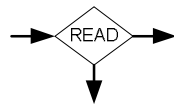
Secara definitif sebuah PDA akan memiliki komponen-komponen sebagai berikut:

1. Himpunan berhingga alphabet Σ
input string untuk PDA dibentuk dari himpunan ini
2. Sebuah state START



Yaitu state untuk memulai penelusuran

3. Satu atau lebih operator READ



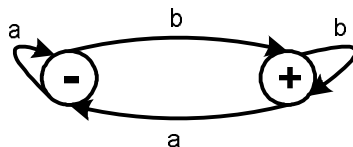
Yaitu state yang berfungsi untuk melakukan pembacaan karakter input string

4. Dua atau lebih halt state yang berbentuk state ACCEPTED dan REJECTED

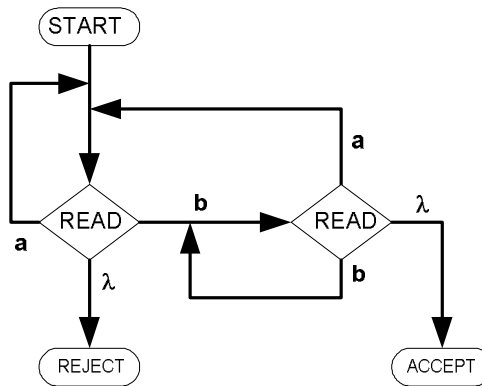


input string akan dinyatakan terkenali jika penelusuran berhenti pada state ACCEPTED. Dan sebaliknya jika penelusuran berhenti pada state REJECTED.

Untuk sementara pendefinisian komponen PDA kita batasi sampai 4 komponen dulu. Dan untuk membantu anda, sebuah contoh sederhana berikut akan mempermudah pemahaman anda terhadap karakteristik mesin PDA. Misalkan kita memiliki sebuah mesin DFA seperti berikut:



Maka bentuk lain dari mesin tersebut jika direpresentasikan dengan format PDA akan nampak seperti berikut:



Kedua mesin di atas memiliki tujuan dan karakteristik yang sama. Penelusuran keduanya sama-sama dimulai state inialisasi (state “-“ untuk DFA, dan state “START” pada PDA). Tahapan berikutnya dari kedua mesin adalah kemampuan menampung input karakter “a” dan/atau “b”. dimana untuk penerimaan/pembacaan input karakter “b” akan menyebabkan kendali mesin beralih ke state selanjutnya yang juga berupa state penerimaan/pembacaan input karakter “a” dan/atau “b”. dan sekaligus pada state ini pula terdapat alternatif pemnghentian penelusuran (state “+” pada DFA, dan state “ACCEPTED” untuk PDA).

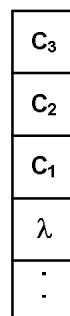
Namun bagaimanapun PDA diciptakan dengan gagasan agar dapat menangani pendefinisian bahasa-bahasa bebas-konteks (yang notabene merupakan subset dari bahas reguler). Oleh karenanya PDA juga ‘dibekali’ dengan kemampuan yang lebih tinggi jika dibandingkan dengan finite automata seperti yang telah kita kenal sebelumnya. Dan dari keempat komponen awal PDA, kita masih belum melihat perbedaan yang signifikan dibandingkan dengan DFA. Perbedaan tersebut akan terlihat pada keempat komponen berikut:

5. Sebuah INPUT TAPE yang berisi sel-sel

Sel i	Sel ii	Sel iii	Sel iv	
C ₁	C ₂	C ₃	λ	...

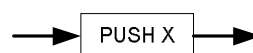
yang berfungsi untuk menampung karakter-karakter pada input string

6. Sebuah PUSHDOWN STACK



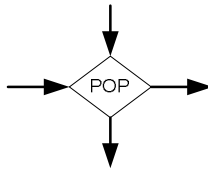
yang berguna untuk menampung karakter input yang telah terbaca. Dan seperti halnya stack pada umumnya, pushdown stack yang digunakan ini juga bersifat LIFO (last in first out). Artinya, karakter yang masuk paling belakang akan keluar paling dulu.

7. Satu atau lebih operator PUSH



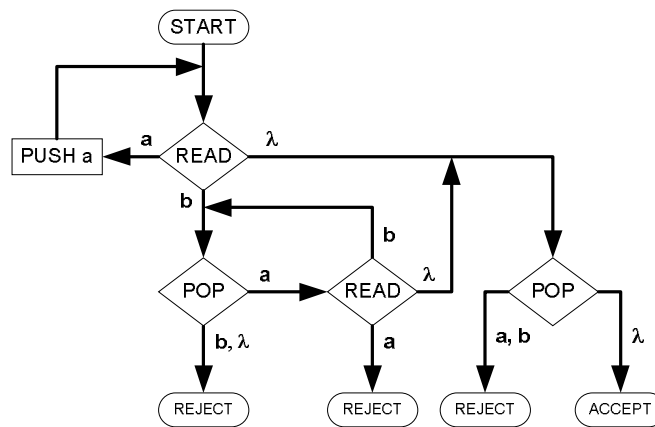
berfungsi untuk memasukkan karakter yang telah terbaca ke dalam stack. Dalam sebuah operator PUSH, akan terdapat outgoing edge = 1, tetapi incoming edge ≥ 1 .

8. Satu atau lebih operator POP

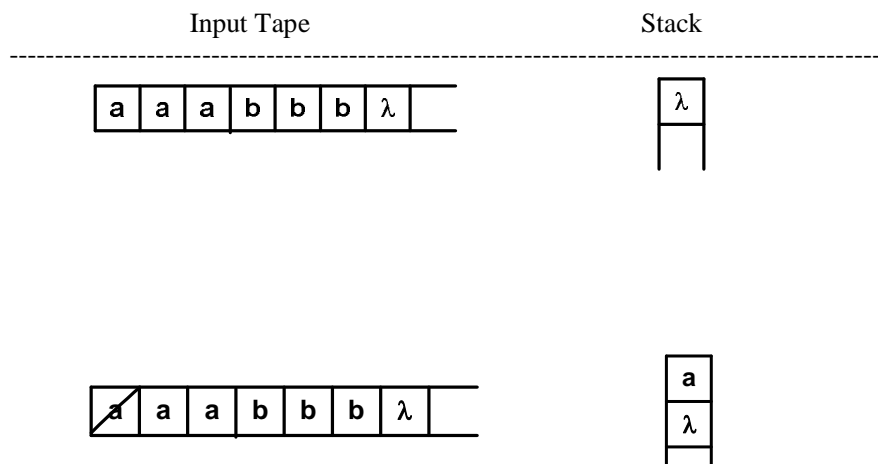


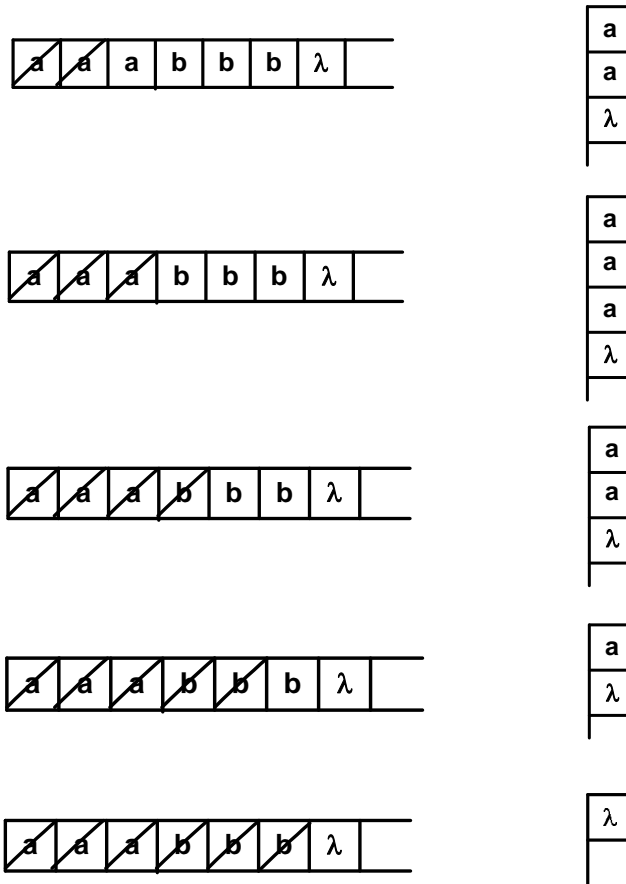
berfungsi untuk mengambil/menghapus karakter dari dalam stack. Dimana dalam sebuah operator POP, akan terdapat incoming edge = jumlah outgoing edge.

Jika demikian, maka bentuk mesin yang telah kita buat sebelumnya secara otomatis juga akan berubah menyesuaikan dengan bentuk mesin PDA yang sebenarnya:



Dapat kita lihat pada gambar di atas, bahwa perubahan yang harus dilakukan memang cukup drastis. Sehingga bentuk mesin terlihat menjadi lebih kompleks. Tetapi esensi mesin tersebut tetap sama. Mesin tersebut juga masih mendefinisikan bahasa yang sama. Dan jika kita coba melakukan penelusuran untuk input aaabbb, maka proses yang terjadi pada input tape dan stack dapat digambarkan seperti berikut:



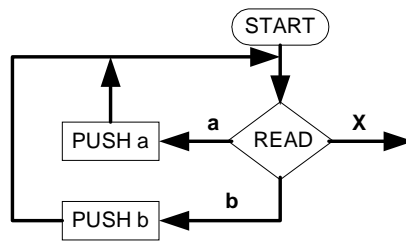


Proses penelusuran sebuah input string akan diinisialisasi oleh terisinya input tape dengan rangkaian karakter input string dan stack berada dalam posisi kosong. Dan di akhir penelusuran, seharusnya input tape dan stack berada dalam keadaan kosong. Selain itu, sebuah input string dapat dinyatakan diterima (accepted) yaitu apabila penelusuran berakhir pada accepted state.

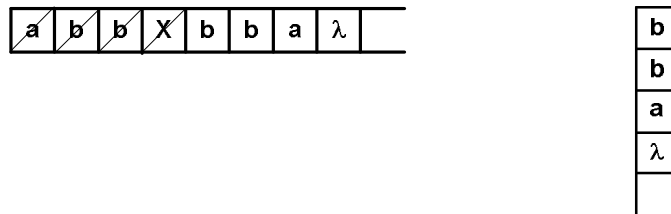
Contoh di atas hanya menggambarkan bentuk dan karakteristik mesin PDA secara umum. Namun belum menunjukkan kelebihan mesin PDA jika dibandingkan dengan mesin pada kelas finite automata. Dan berikut ini kita akan melihat sebuah contoh yang dapat menggambarkan kelebihan tersebut.

Misalkan dibuat PDA untuk bahasa palindrome yang berbentuk $s \text{ X reverse}(s)$ dimana s adalah substring dari $(a + b)^*$. Dan dengan segala keterbatasan yang dimiliki, yang pasti untuk tujuan tersebut, kita tidak dapat melakukannya dengan DFA maupun NFA.

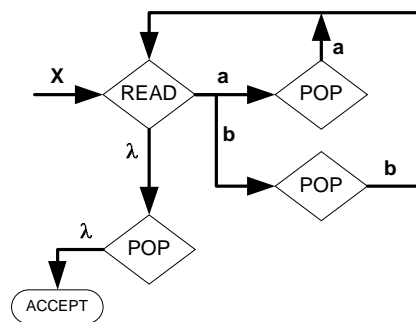
Bagian depan dari PDA akan mempunyai bentuk :



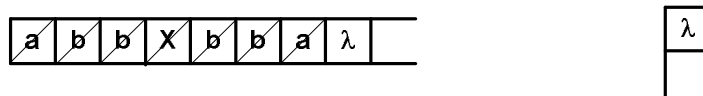
Misal jika diberi input string $abbXbba$, maka pemrosesan untuk substring abb adalah seperti berikut :



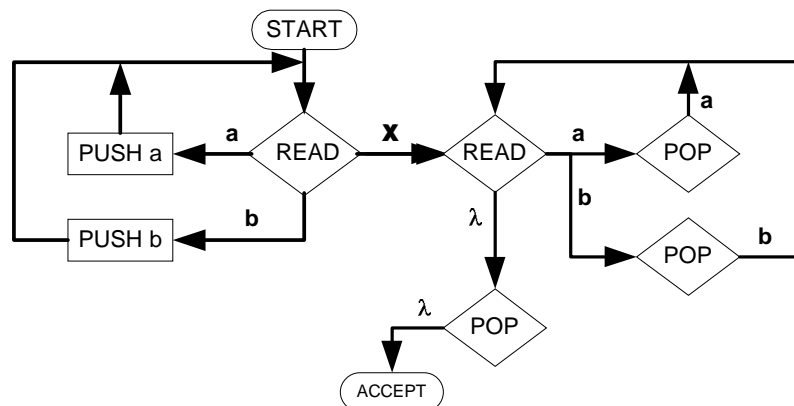
Dan bagian lain dibuat untuk mengakomodasi penelusuran substring $reverse(s)$:



Dan sisa substring bba akan diproses seperti berikut :



Sehingga bentuk keseluruhan PDA untuk palindrome $sXreverse(s)$ adalah seperti berikut :

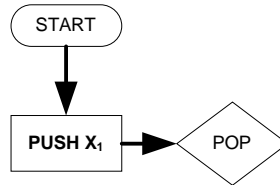


11.2. Membentuk PDA dari CFG

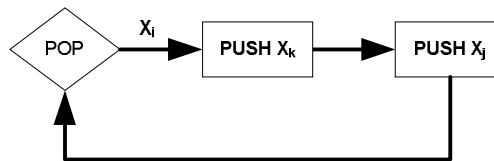
Karena PDA adalah sebuah mesin yang diciptakan dengan tujuan agar dapat mendefinisikan bahasa dari kelas bahasa bebas-konteks, sekaligus mampu mengenali keanggotaan bahasa, maka selayaknya apabila kedua alat pendefinisi bahasa dari kelas bebas-konteks (CFG dan PDA) ini dapat saling ditransformasikan.

Dan berikut ini adalah beberapa langkah mudah untuk mentransformasikan sebuah CFG menjadi bentuk PDA:

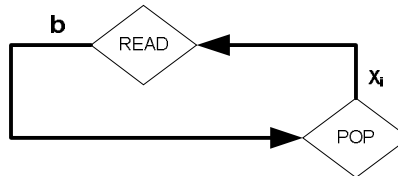
1. Sebuah non-terminal X_1 yang menjadi Start Symbol akan direpresentasikan menjadi :



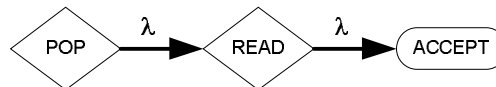
2. Sebuah production $X_i \rightarrow X_j X_k$, direpresentasikan menjadi :



3. Sebuah production $X_i \rightarrow b$, direpresentasikan menjadi :



4. Sebuah production $X_i \rightarrow \epsilon$, akan menjadi :

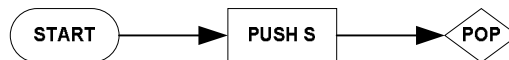


Sebagai contoh proses konversi dari CFG menjadi PDA dapat dilihat sebagai berikut:

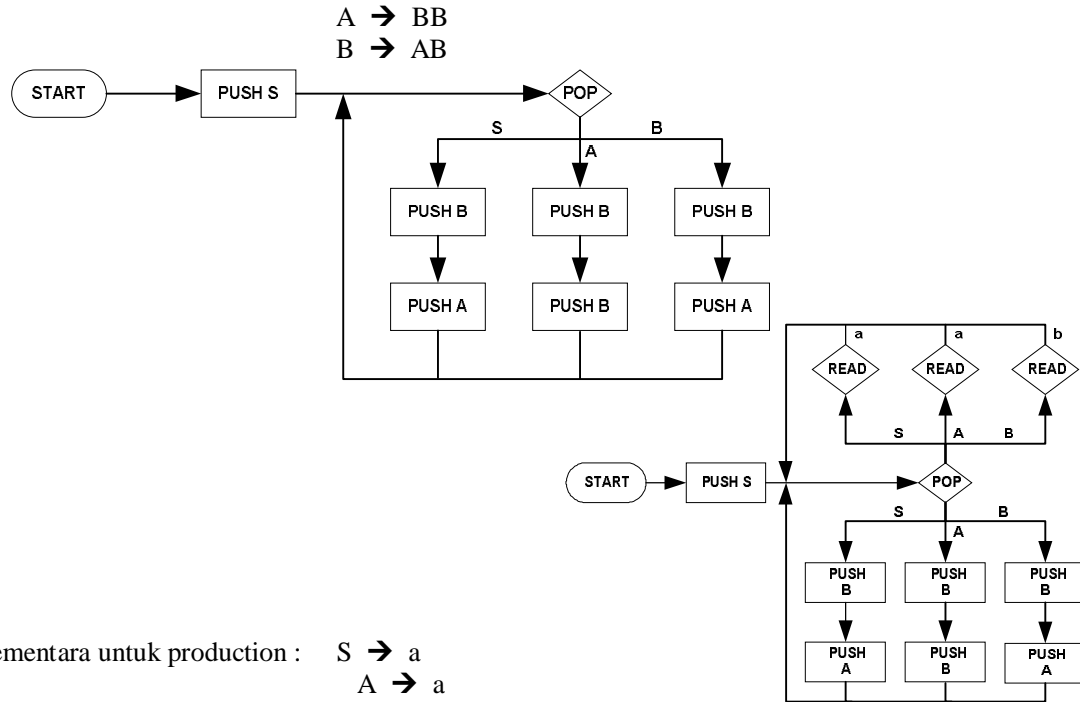
$S \rightarrow AB$	$A \rightarrow a$
$A \rightarrow BB$	$B \rightarrow a$
$B \rightarrow AB$	$B \rightarrow b$

Proses pembentukan PDA dari CFG di atas adalah seperti berikut :

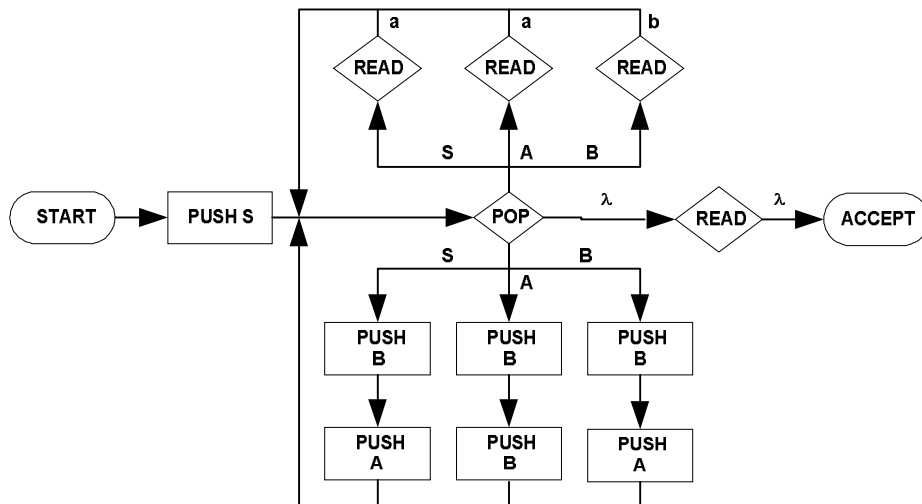
Untuk Start Symbol S :



Sedangkan untuk production : $S \rightarrow AB$



Dan bentuk keseluruhan PDA hasil konversi adalah sebagai berikut:



11.3. Mesin Turing

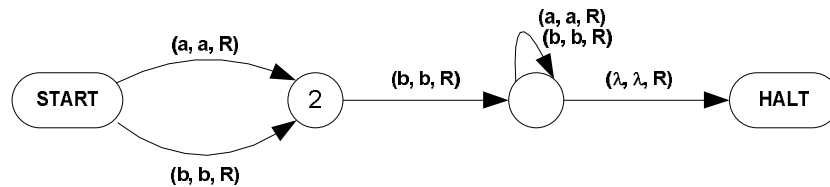
Dalam bab ini kita juga akan berkenalan dengan kelas mesin terakhir yang disebut sebagai mesin Turing. Mesin ini diyakini memiliki kemampuan lebih tinggi dibandingkan 2 kelas mesin sebelumnya (FA dan PDA). Karena selain mampu memodelkan bahasa reguler dan mampu menangani pemodelan bahasa yang bersifat bebas-konteks, mesin ini ternyata juga mampu memodelkan bahasa-bahasa dari kelas yang lain. Sehingga secara umum dikatakan bahwa mesin Turing mampu memodelkan solusi bagi permasalahan-permasalahan yang bersifat decidable (dapat diselesaikan) maupun yang bersifat undecidable (tidak dapat diselesaikan) dan yang bersifat intractable (sangat sulit diselesaikan). Dan karena kemampuannya itulah mesin Turing disebut sebagai model atau miniatur komputer modern yang sangat ideal.

Mesin Turing yang akan kita pelajari ini memiliki komponen-komponen sebagai berikut:

1. Himpunan berhingga alphabet Σ
Dimana input string untuk PDA dibentuk dari himpunan ini.
2. Sebuah INPUT TAPE
Yang berbentuk rangkaian sel yang masing-masing berisi satu karakter.
3. Sebuah TAPE HEAD
untuk membaca karakter input. Pembacaan dilakukan per karakter. Saat inialisasi, tape head berada pada posisi pertama.
4. Himpunan berhingga alphabet Γ
dimana output string yang dihasilkan adalah anggota himpunan Γ^* .
5. Himpunan berhingga STATE
dengan salah satu state diperlakukan sebagai START STATE dan satu lagi sebagai HALT STATE.
6. Himpunan berhingga ARC
Yang berfungsi sebagai penghubung antar state. Label ARC berbentuk (input, output, arah).

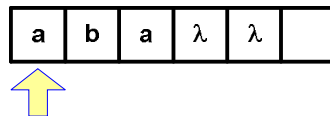
Contoh:

Sebuah mesin turing yang mendefinisikan bahasa $(a+b) b (a+b)^*$ digambarkan sebagai berikut :



Misalkan kita akan mengenali sebuah input string : aba

Maka pada saat inialisasi, posisi input tape dan tape head akan terlihat seperti berikut :

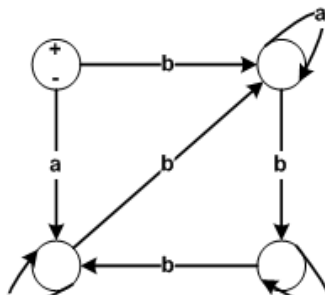


Proses pengenalan input string akan kami sajikan dalam bentuk tabular agar mudah dipahami:

STATE	POSISI TAPE HEAD	OUTPUT KARAKTER
start	<u>a</u> b a λ	
2	a <u>b</u> a λ	a
3	a b <u>a</u> λ	a b
3	a b a <u>λ</u>	a b a
halt	a b a λ	a b a

11.4. Latihan Soal

1. Konversikan FA di bawah menjadi PDA yang ekivalen :



2. $S \rightarrow XaaX$
 $X \rightarrow aX \mid bX \mid \lambda$
3. $S \rightarrow XY$
 $X \rightarrow aX \mid bX \mid a$
 $Y \rightarrow Ya \mid Yb \mid a$
4. $S \rightarrow Xa \mid Yb$
 $X \rightarrow Sb \mid b$
 $Y \rightarrow Sa \mid a$
5. $S \rightarrow XaX \mid YbY$
 $X \rightarrow YY \mid aY \mid b$
 $Y \rightarrow b \mid bb$

6. abb
7. abab
8. aabb
9. aabbbb
10. Tentukan bentuk CFG yg dpt menerima bahasa yg dinyatakan oleh PDA tsb.

