MODUL KULIAH KE → 14

ALGORITMA dan STRUKTUR DATA (3 SKS)

MATERI KULIAH :

Algoritma Breath first search (BFS), algoritma Depth First Search (DFS), implementasi DFS pada topological Sort, Strongly Connected Component

POKOK BAHASAN :

Algoritma Graph Dasar

Oleh : Endang Sri Rahayu

Pada modul ini akan dibahas algoritma *searching* pada *graph*. Searching *graph* berarti secara sistematik melakukan penelusuran pada *edge* dari *graph* sehingga semua verteks terkunjungi. Algoritma *Graph-Searching* dapat meliputi struktur dari *graph*. Banyak algoritma memulai dengan pencarian masukan *graph* untuk menentukan informasi dari struktur ini. Algoritma *graph* yang lain dilakukan dengan cara memperluas algoritma graph *searching* dasar. Teknik-teknik *searching* pada *graph* adalah masalah utama pada algoritma *graph*.

1. Breadth-First Search

Bread-first Search adalah salah satu algoritma yang paling sederhana dalam pencarian pada *graph* dan merupakan pola dasar untuk banyak algoritma *graph* yang penting. Algoritma *shortest path* dan algoritma *spanning tree* menggunakan ide *breadth-first search*.

Diberikan sebuah *graph* G=(V,E) dan dan verteks s sebagai *source*. BFS secara sistematis menelusuri *edge* dari G untuk meliput setiap verteks yang dapat dapat dijangkau dari s. Dia menghitung jarak dari s ke semua verteks yang dapat dijangkau. Dia juga menghasilkan "*breath-first tree*" dengan *root* s yang berisi semua verteks yang dapat dijangkau. Untuk setiap verteks v yang terjangkau dari s, lintasan pada *breath first tree* dari s ke v berhubungan dengan jarak terpendek dari s ke v dalam G, sehingga, lintasan berisi panjang sisi terpendek. Algoritma bekerja pada *directed graph* maupun *undirected graph*.

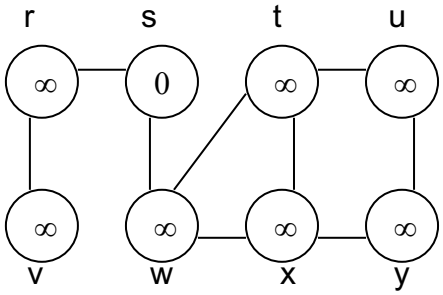
BFS(G,s)

1. **for** each vertex u $\in$ V[G] – {s}
2. **do** color[u[$\leftarrow$ white
3. d[u] $\leftarrow \infty$
4. π [u] $\leftarrow$ nil
5. color[s] $\leftarrow$ gray
6. d[s] $\leftarrow$ 0
7. π [s] $\leftarrow$ nil
8. Q $\leftarrow$ {s}
9. **while** Q $\neq$ 0
10. **do** u $\leftarrow$ head[Q]
11. **for** each v $\in$ Adj[u]

12. **do if** color[v] = white
13. **then** color[u] = gray
14. d[v] ← d[v] + 1
15. π [v] ← u
16. ENQUEUE(Q,v)
17. DEQUEUE(Q)
18. Color[u] ← green

Operasi BFS pada *graph* tak berarah akan ditunjukkan pada gambar berikut ini. 3 sisi dihasilkan dengan BFS ditunjukkan berbayang. Queue Q dipergunakan untuk menyimpan verteks-verteks yang akan dikunjungi dengan nilai prioritas seperti yang diberikan.

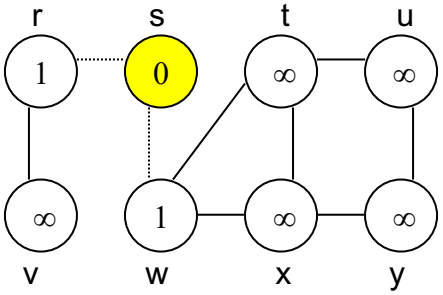
a)



Q | s |

0

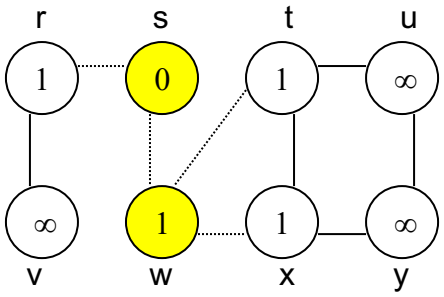
b)



Q | w | r |

1 1

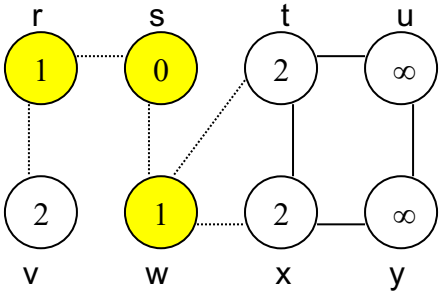
c)



Q | r | t | x |

1 2 2

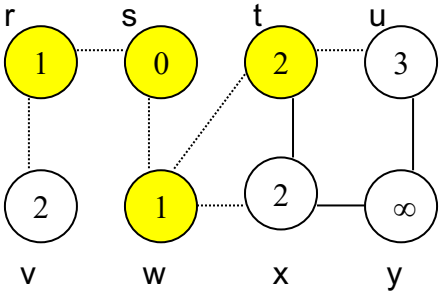
d)



Q | t | x | v |

2 2 2

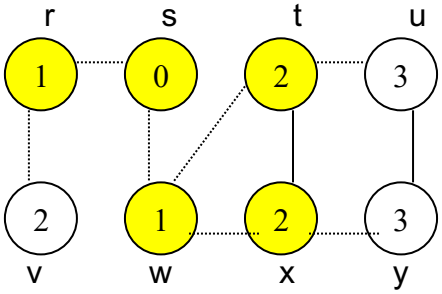
e)



Q | x | v | u |

2 2 3

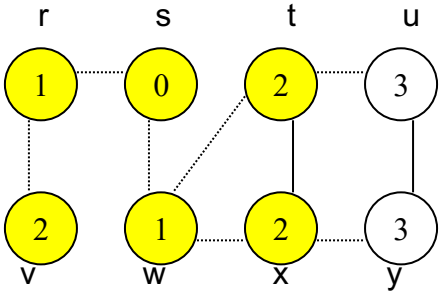
f)



Q | v | u | y |

2 3 3

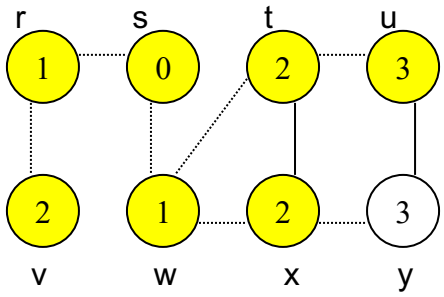
g)



Q | u | y |

3 3

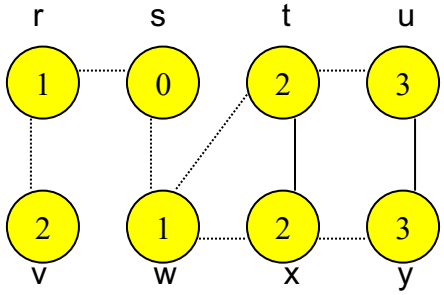
h)



Q v

3

i)



Q Ø

19. Depth-First Search

Strategi yang digunakan dalam DFS adalah mencari “kedalaman” graph semampu mungkin. Pada DFS, edge menelusuri semua verteks disekitar verteks yang baru saja ditemukan. Ketika verteks ditemukan, pencarian akan melakukan “backtracks” untuk menemukan edge dari verteks yang tertinggal dimulai dari verteks yang sudah ditemukan.

Pseudocode berikut ini adalah algoritma DFS dasar. Masukan pada *graph* G bisa berarah (*directed*) ataupun tidak berarah (*undirected*). *Variable time* adalah *global variable* yang digunakan untuk *timestamping*.

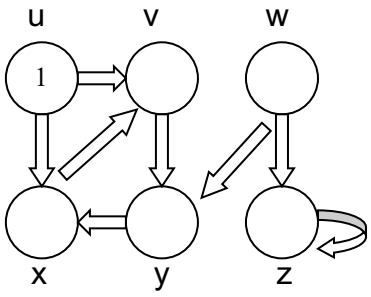
DFS(G)

1. **for** each verteks u ∈V(G)
2. **do** *color*[u] ← white
3. [u] ← NIL
4. time ← 0
5. **for** each vertex u ∈ V[G]
6. **do if** *color*[u] = white
7. **then** DFS-Visit(u)

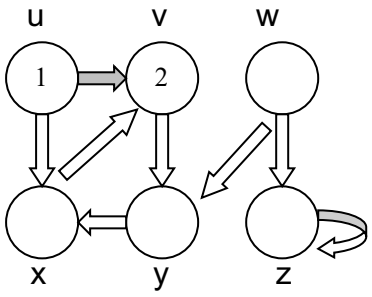
DFS-Visit(u)

1. *color*(u) ← Gray
2. d[u] ← time ← time + 1
3. **for** each v ∈ Adj[u]
4. **do if** color[u] = white
5. **then** π [u] ← u
6. DFS-Visit(u)
7. *color*[u] ← green
8. f[u] ← time ← time+1

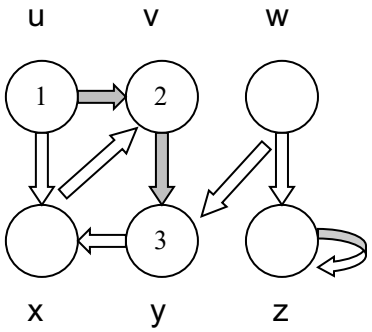
Ilustrasi



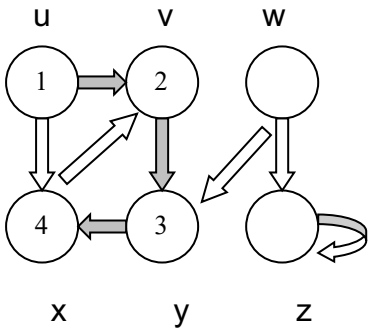
a



b

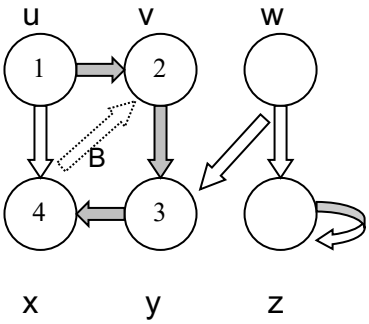


c

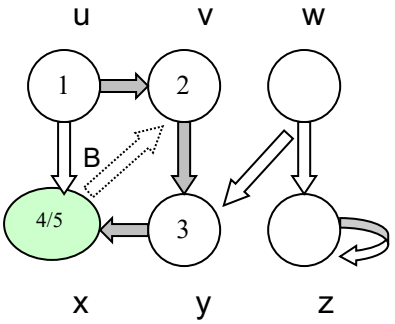


d

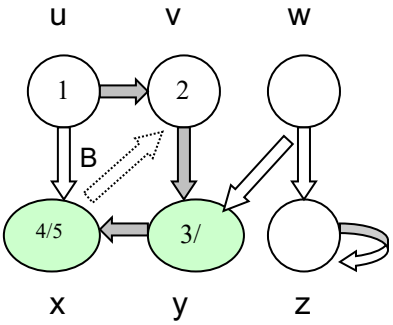
backward



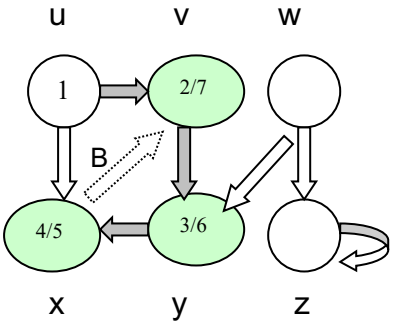
e



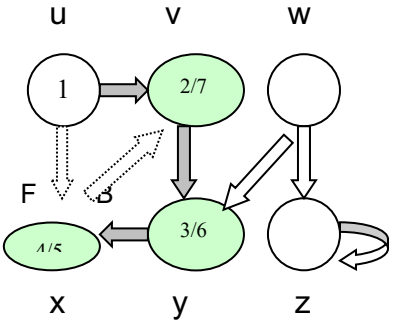
f



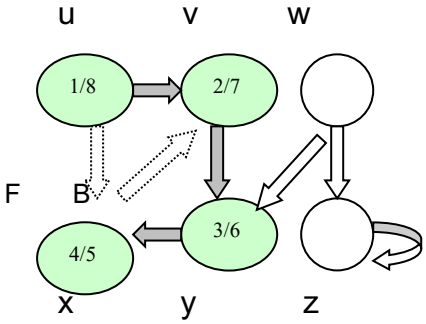
g



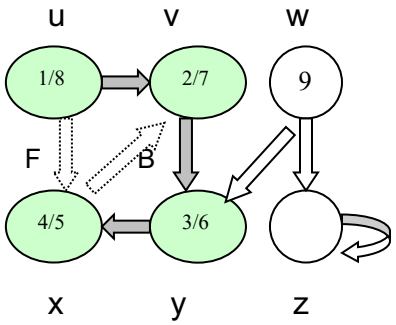
h



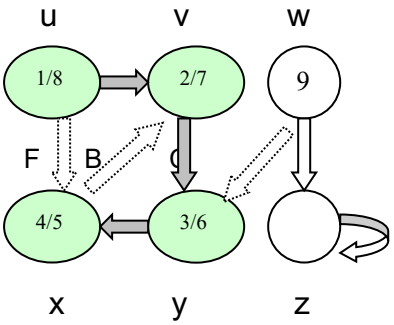
i



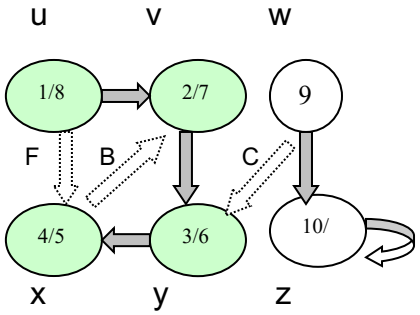
j



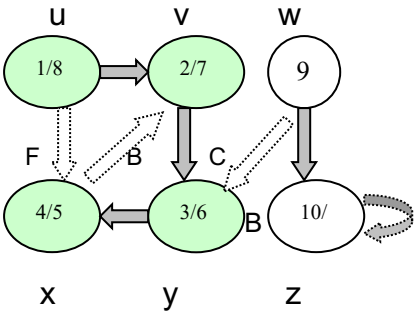
k



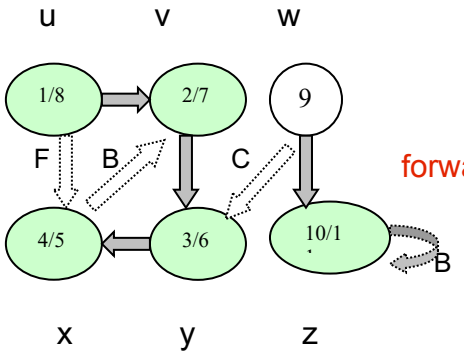
l



m

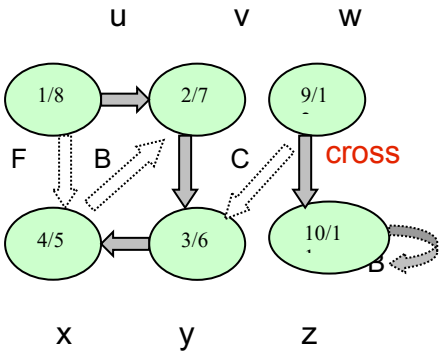


n



forward

o



cross

p

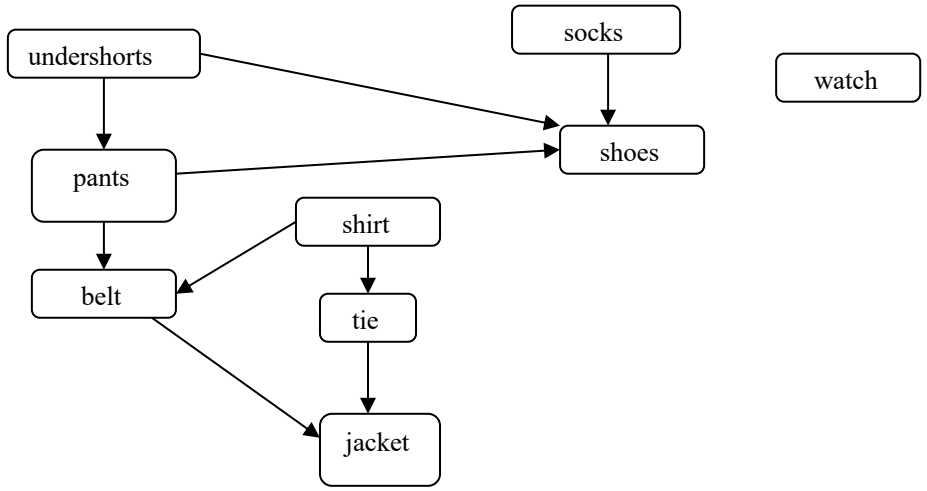
8. *Topological Sort*

Bagian ini akan menunjukkan bagaimana *depth-first search* dapat digunakan untuk membentuk *topological sort* dari *acyclic graph* berarah atau biasa disebut "DAG" (*Directed Acyclic Graph*)

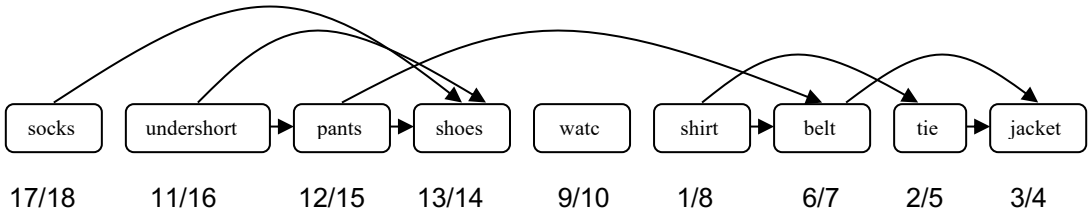
Topological sort dari *graph* dapat dipandang sebagai urutan *vertex* sepanjang garis horizontal sehingga semua sisi berarah bergerak dari kiri ke kanan. Jadi *topological sort* berbeda dengan *sorting* yang telah dipelajari pada bagian terdahulu.

Acyclic graph berarah digunakan pada banyak aplikasi untuk menunjukkan hubungan antara kejadian-kejadian. Berikut ini diberikan contoh yang muncul ketika Professor Bumstead mengenakan pakaian di pagi hari. Profesor harus mengenakan perlengkapan tertentu sebelum mengenakan yang lain (contoh, socks sebelum shoes). Beberapa item yang lain mungkin diletakkan pada arah berbeda (contoh*, socks* dan *pants*).

Directed Edge (u, v) dalam DAG pada gambar dibawah ini menunjukkan bahwa perlengkapan u harus dikenakan sebelum perlengkapan v.



Gambar berikut ini menunjukkan urutan secara *topological* dari DAG dengan urutan sepanjang garis horisontal sehingga semua edge bergerak dari kiri ke kanan.



Berikut ini diberikan algoritma sederhana dari pengurutan secara *topological* dari sebuah DAG :

Topological-Sort(G)

1. Call DFS untuk menghitung *finishing time* f[v] dari setiap verteks
2. Setiap verteks berakhir, sisipkan pada linked list
3. Kembali ke linked list dari verteks

4. Strongly Connected Components

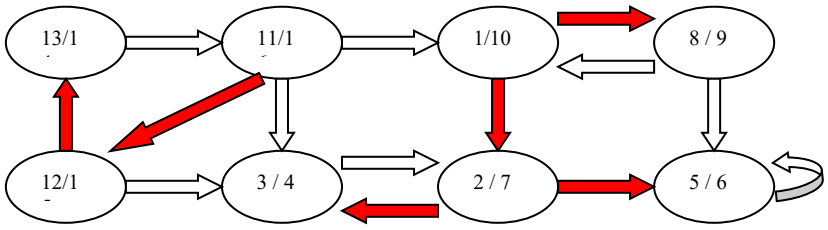
Berikut ini akan dibahas mengenai aplikasi dari *Depth-first Search* yaitu berupa dekomposisi *graph* berarah kedalam komponen-komponen yang saling terhubung (*strongly connected components*). Bagian ini akan menunjukkan bagaimana mengerjakan dekomposisi menggunakan 2 DFS. Banyak algoritma bekerja dengan *graph* berarah yang dimulai dengan

dekomposisi, pendekatan ini seringkali berdasarkan problem yang dibagi menjadi beberapa subproblem, setiap subproblem untuk setiap *strongly connected components*. Penggabungan solusi mengikuti struktur koneksi antara *strongly connected components*

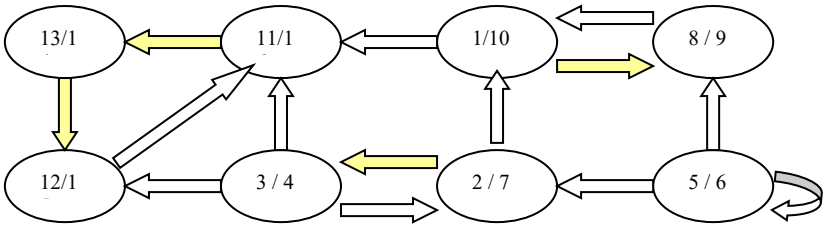
Algoritma *Strongly Connected Component*

1. Buat DFS(G) untuk menghitung *finishing time* f[u] untuk setiap *vertex* u
2. Buat G^T
3. Buat DFS(G^T), tetapi pada *loop* utama DFS, anggap *vertex* dalam arah *decreasing* f[u]
4. *Output vertex* pada setiap *tree* dalam *depth-first forest* pada langkah 3 adalah bagian *strongly connected component*

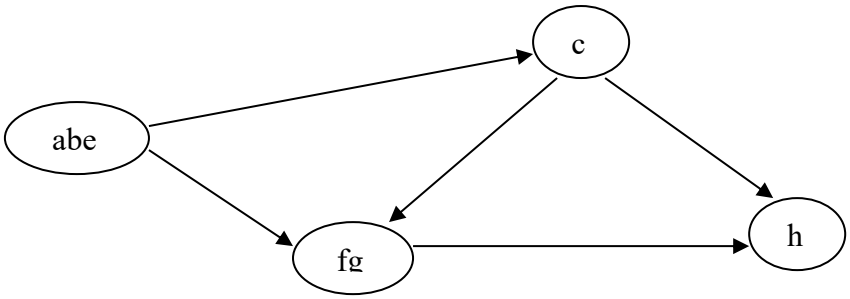
Gambar berikut ini menunjukkan *graph* G. *Strongly connected components* dari G ditunjukkan dengan garis berwarna merah. Setiap vertex diberi label dengan nilai *discovery* dan *finishing time*-nya.



Gambar berikut ini adalah *graph* G^T, yaitu *transpose* dari G. *Depth first Tree* yang dihitung pada langkah ke-3 dari algoritma *Strongly-Connected-components* ditunjukkan dengan sisi-sisi tree yang diberi warna kuning. Setiap *strongly connected components* berhubungan ke *depth first tree*. *Vertex* b, c, g dan h, dengan warna hijau adalah *forefather* dari setiap *vertex* pada *strongly connected components*-nya, *vertex-vertex* ini juga merupakan *root* dari *depth first tree* yang dihasilkan oleh *depth first search* dari G^T



Acyclic component graph ditentukan dengan meringkas setiap *strongly connected component* dari G menjadi *single vertex*.



REFERENSI

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, **Introduction to Algoritms**, Mc Graw–Hill, 1990
- Dr W. Nugroho, **Bahan Kuliah Desain Analisis Algoritma**, Fasilkom, Universitas Indonesia, 2001