Perkenalan Graf

Tim Olimpiade Komputer Indonesia

Pendahuluan

Melalui dokumen ini, kalian akan:

- Mengenal konsep dan terminologi graf.
- Mengetahui jenis-jenis graf.
- Mengenal representasi graf pada pemrograman.
- Mengenal metode-metode yang digunakan dalam graf.

Motivasi

- Diberikan sebuah struktur kota dan jalan.
- Terdapat V kota, dan E ruas jalan.
- Setiap ruas jalan menghubungkan dua kota.
- Diberikan kota awal, tentukan berapa banyak ruas jalan paling sedikit yang perlu dilalui untuk mencapai suatu kota tujuan!

Pertanyaan

Bagaimana cara merepresentasikan struktur perkotaan dan jalan pada pemrograman?

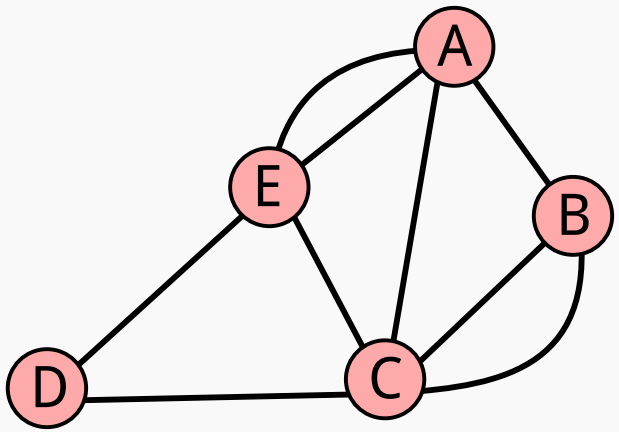
Bagian 1

Perkenalan Graf

Mengenal Graf

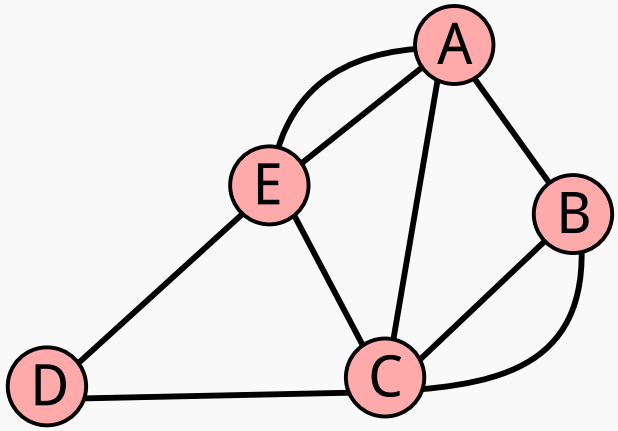
Graf adalah struktur yang terdiri dari **node/vertex** dan **edge.**

Node direpresentasikan dengan bentuk lingkaran dan *edge* direpresentasikan dengan bentuk garis pada ilustrasi berikut:



Mengenal Graf (lanj.)

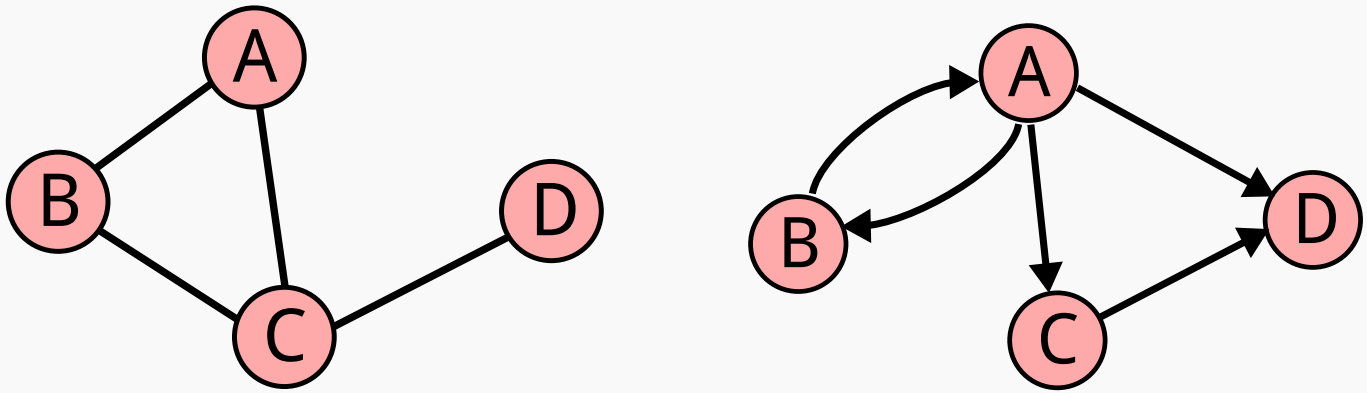
- *Edge* merupakan penghubung antar *node*.
- *Degree* suatu *node* merupakan jumlah *edge* yang terhubung pada *node* tersebut
- Pada contoh ilustrasi berikut, *degree node* A $= 4$, *degree node* B $= 3$, dan *degree node* C $= 5$.



Jenis Graf

Berdasarkan hubungan antar *node*:

- **Graf tak berarah**: *edge* dari A ke B dapat ditelusuri dari A ke B dan B ke A.
- **Graf berarah**: *edge* dari A ke B hanya dapat ditelusuri dari dari A ke B.

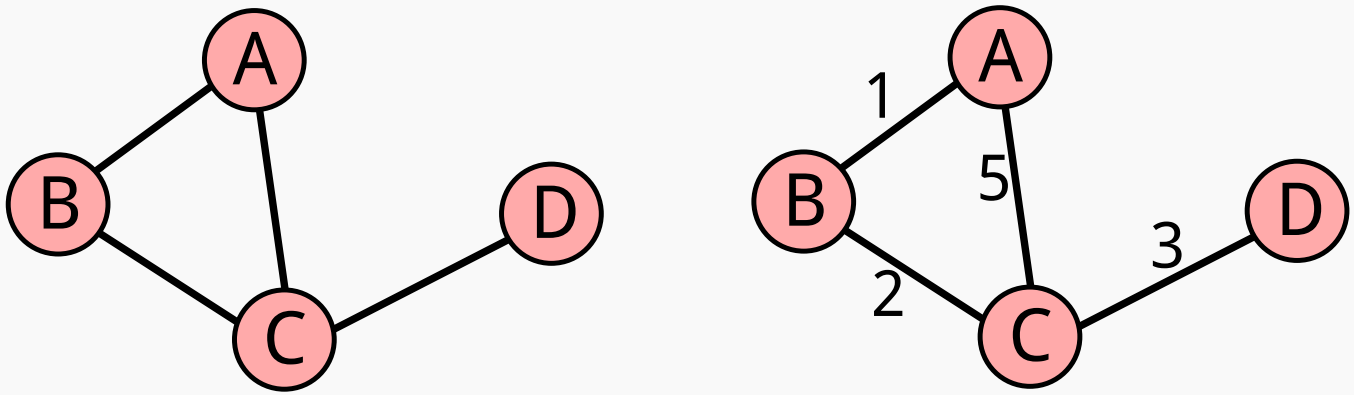


Graf tak berarah dan graf berarah

Jenis Graf (lanj.)

Berdasarkan bobot dari *edge*:

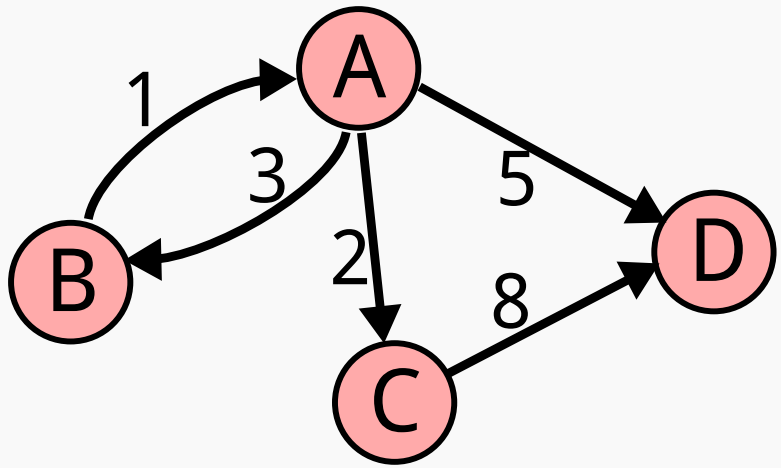
- **Graf tak berbobot**, yaitu graf dengan *edge* yang bobotnya seragam dan hanya bermakna terdapat hubungan antar *node*.
- **Graf berbobot**, yaitu graf dengan *edge* yang dapat memiliki bobot berbeda-beda. Bobot pada *edge* ini bisa jadi berupa biaya, jarak, atau waktu yang harus ditempuh jika menggunakan *edge* tersebut.



Graf tak berbobot dan graf berbobot

Jenis Graf (lanj.)

Tentu saja, suatu graf dapat memiliki kombinasi dari sifat-sifat tersebut.
Misalnya graf berbobot berarah:



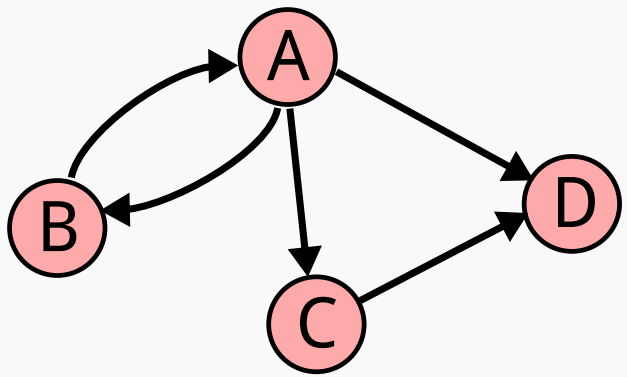
Representasi Graf pada Pemrograman

- Dalam pemrograman, dibutuhkan sebuah struktur agar data mengenai graf dapat disimpan dan diolah.
- Representasi yang akan kita pelajari adalah *adjacency matrix*, *adjacency list*, dan *edge list*.
- Masing-masing representasi memiliki keuntungan dan kerugiannya.
- Penggunaan representasi graf bergantung dengan masalah yang sedang dihadapi.

Adjacency Matrix

- Kita akan menggunakan matriks dengan ukuran $N \times N$ dengan N merupakan banyaknya *node*.
- Pada graf tidak berbobot:
  - Jika terdapat *edge* dari A ke B, maka $matrix[A][B] = 1$.
  - Jika tidak ada, maka $matrix[A][B] = 0$.

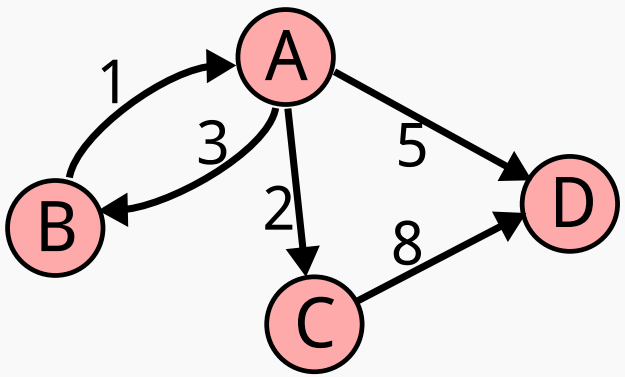
	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	0	0	0	1
D	0	0	0	0



Adjacency Matrix (lanj.)

- Pada graf berbobot:
  - Jika terdapat *edge* dari A ke B dengan bobot w, maka $matrix[A][B] = w$.
  - Jika tidak ada, maka dapat ditulis $matrix[A][B] = \infty$.

	A	B	C	D
A	∞	3	2	5
B	1	∞	∞	∞
C	∞	∞	∞	8
D	∞	∞	∞	∞



Analisis Adjacency Matrix

- Pada graf tak berarah, *adjacency matrix* simetris terhadap diagonalnya.
- Representasi ini mudah diimplementasikan.
- Menambah atau menghapus *edge* dapat dilakukan dalam $O(1)$.
- Untuk memeriksa apakah dua *node* terhubung juga dapat dilakukan dalam $O(1)$.
- Untuk mendapatkan daftar tetangga dari suatu *node*, dapat dilakukan iterasi $O(V)$, dengan V adalah banyaknya *node*.

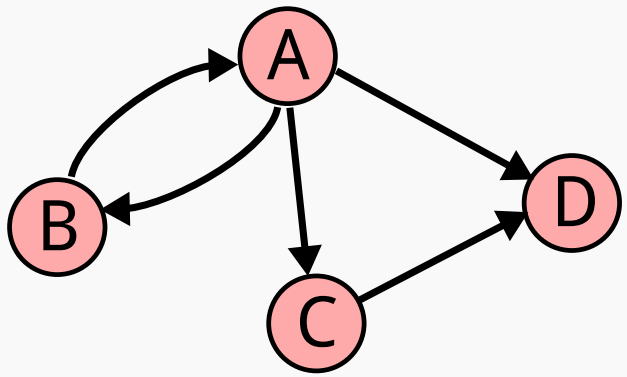
Kekurangan Adjacency Matrix

- Kekurangan dari representasi ini adalah boros memori.
- Memori yang dibutuhkan selalu $O(V^2)$, tidak dapat digunakan untuk graf dengan *node* mencapai ratusan ribu.
- Jika banyaknya *edge* jauh lebih sedikit dari $O(V^2)$, maka banyak memori yang terbuang.

Adjacency List

- Merupakan salah satu alternatif representasi graf.
- Untuk setiap *node*, buat sebuah *list* yang berisi keterangan mengenai tetangga *node* tersebut.
- Misalnya untuk graf tak berbobot, kita cukup menyimpan *node*-*node* tetangga untuk setiap *node*.

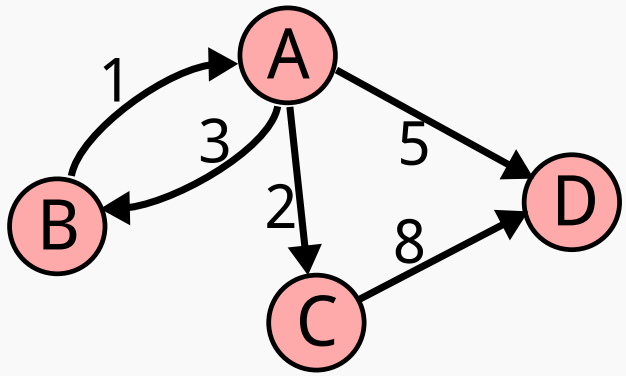
A	$[B, C, D]$
B	$[A]$
C	$[D]$
D	$[\,]$



Adjacency List (lanj.)

- Untuk graf berbobot, kita dapat menyimpan *node-node* tetangga beserta bobotnya.

A	$[< B, 3 >, < C, 2 >, < D, 5 >]$
B	$[< A, 1 >]$
C	$[< D, 8 >]$
D	$[\,]$



Implementasi Adjacency List

- Kita dapat menggunakan struktur data *array of lists*.
- Tiap *list* berisi keterangan mengenai tetangga suatu *node*.
- Ukuran dari *array* merupakan V, yang mana V merupakan banyaknya *node*.
- Dengan menggunakan *list*, banyaknya memori yang digunakan untuk setiap *node* hanya sebatas banyak tetangganya.
- Secara keseluruhan jika graf memiliki E *edge*, maka total memori yang dibutuhkan adalah $O(E)$.

Implementasi Adjacency List (lanj.)

- *List* yang dimaksud bisa berupa *linked list* atau *resizable array*.
- Bagi pengguna C++ atau Java, struktur *list* yang dapat digunakan adalah vector atau ArrayList.
- Untuk pengguna C atau Pascal, struktur *linked list* perlu dibuat terlebih dahulu.

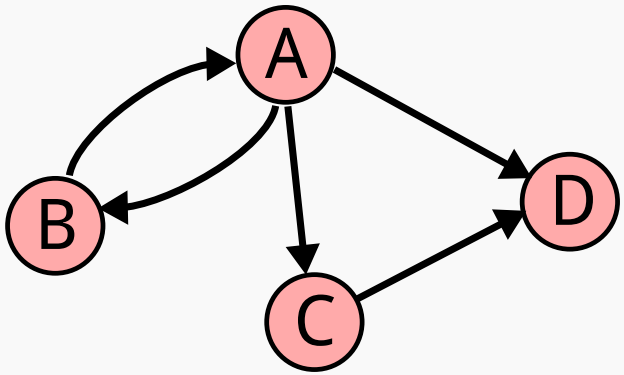
Analisis Adjacency List

- Kompleksitas menambah *edge* adalah $O(1)$, dan menghapus adalah $O(K)$ dengan K adalah banyaknya tetangga dari *node* yang *edge*-nya dihapus.
- Memeriksa apakah dua *node* terhubung oleh *edge* juga dilakukan dalam $O(K)$.
- Demikian juga untuk mendapatkan daftar tetangga dari *node*, kompleksitasnya adalah $O(K)$. Perhatikan bahwa pencarian daftar tetangga ini sudah paling efisien.

Edge List

- Sesuai namanya, kita merepresentasikan graf dengan sebuah *list*.
- Seluruh keterangan *edge* dimasukkan kedalam *list* tersebut.
- Berbeda dengan *adjacency list* yang membutuhkan *array of list*, representasi ini hanya butuh sebuah *list*.

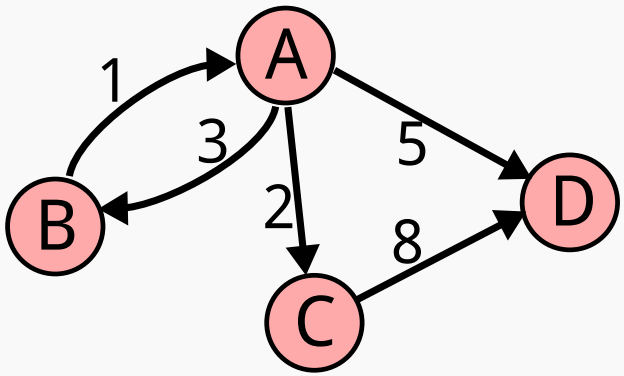
$< A, B >,$
$< A, C >,$
$< A, D >,$
$< B, A >,$
$< C, D >$



Edge List (lanj.)

- Untuk graf berbobot, kita juga menyimpan bobot dari setiap *edge*.

$< A, B, 3 >,$
$< A, C, 2 >,$
$< A, D, 5 >,$
$< B, A, 1 >,$
$< C, D, 8 >$



Implementasi Edge List

- Untuk implementasinya, kita membutuhkan struktur data sebuah *list* (atau *array*).
- Jelas bahwa memori yang dibutuhkan adalah $O(E)$, dengan E adalah banyaknya *edge* pada keseluruhan graf.

- Pada beberapa kasus, dilakukan pengurutan terhadap *edge list* atau digunakan struktur data *binary search tree*, yang memungkinkan implementasinya lebih efisien. Namun untuk saat ini kita tidak mempelajari hal tersebut.

Analisis Edge List

- Kompleksitas menambah *edge* adalah $O(1)$.
- Bergantung dari implementasi, kompleksitas menghapus *edge* dan memeriksa keterhubungan sepasang *node* bisa berupa $O(\log E)$ sampai $O(E)$.
- Demikian juga untuk mendapatkan daftar tetangga dari *node*, kompleksitasnya bisa berkisar antara $O(\log E + K)$ sampai $O(E)$, dengan K adalah banyaknya tetangga dari *node* tersebut.

Keuntungan dan Kerugian Representasi Graf

Untuk graf dengan V *node* dan E *edge*:

	Adj.Matrix	*Adj.List*	*Edge List*
Tambah *edge*	$O(1)$	$O(1)$	$O(1)$
Hapus *edge*	$O(1)$	$O(K)$	$O(E)$
Cek keterhubungan	$O(1)$	$O(K)$	$O(E)$
Daftar tetangga	$O(V)$	$O(K)$	$O(E)$
Kebutuhan memori	$O(V^2)$	$O(E)$	$O(E)$

Dengan K adalah banyaknya *node* yang bertetangga dengan *node* yang sedang kita periksa.

Bagian 2

Penjelajahan Graf

Penjelajahan Graf

- Representasi graf saja belum berguna karena belum dapat mencari informasi mengenai suatu graf
- Penjelajahan graf merupakan penelusuran *node-node* pada suatu graf.

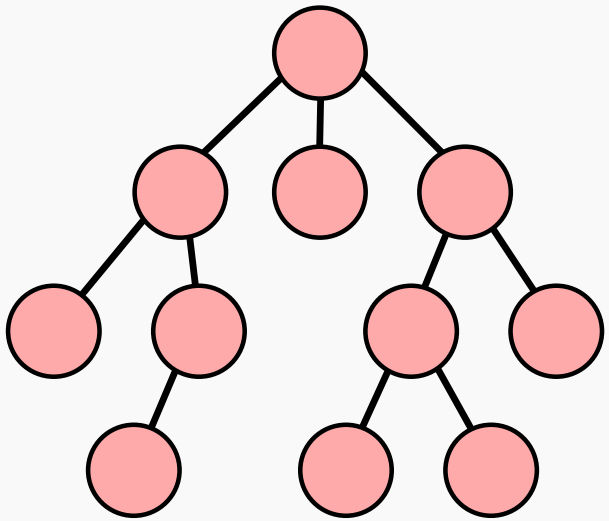
Penjelajahan Graf (lanj.)

- Diberikan *node* A dan *node* B, apakah dari *node* A kita dapat pergi ke *node* B dengan *edge* yang ada?
- Permasalahan tersebut dapat diselesaikan menggunakan penjelajahan graf.

- Terdapat 2 metode yang dapat digunakan, yaitu **DFS** dan **BFS**.

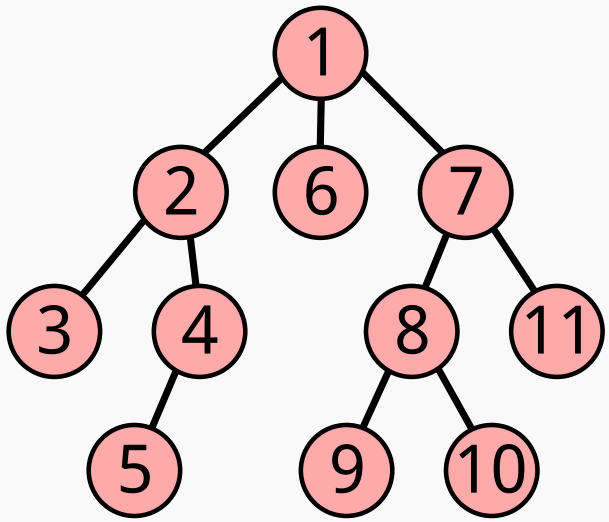
DFS: Depth-First Search

Penelusuran dilakukan terhadap *node* yang lebih dalam terlebih dahulu (*depth-first*).
Sebagai contoh, misal terdapat graf berikut:



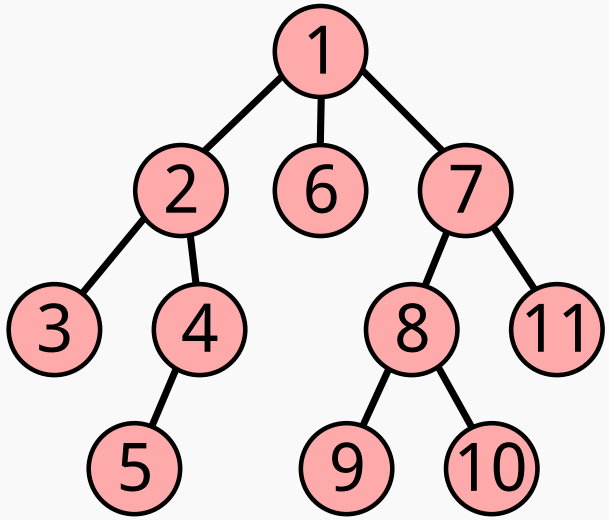
DFS: Depth-First Search (lanj.)

Penelusuran secara DFS akan dilakukan dengan cara berikut:



- Angka pada *node* menunjukkan urutan *node* tersebut dikunjungi.
- *Node* 1 dikunjungi pertama, *node* 2 dikunjungi kedua, dan seterusnya).

Penelusuran DFS



- Dapat dilihat bahwa DFS mencoba menelusuri *node* yang dalam terlebih dahulu.
- *Node* yang dekat dengan *node* pertama (seperti *node* 7 dan 8) akan dikunjungi setelah DFS selesai mengunjungi *node* yang lebih dalam (seperti *node* 3, 4, dan 5)
- Dalam pemrograman, DFS biasa dilakukan dengan rekursi atau struktur data *stack*.

Implementasi DFS (Rekursif)

Asumsikan:

- Setiap *node* dinomori dari 1 sampai V
- $adj(x)$ menyatakan himpunan tetangga dari *node* x.
- $visited[x]$ bernilai *true* hanya jika x telah dikunjungi.

$\mathrm{DFS}(curNode)$

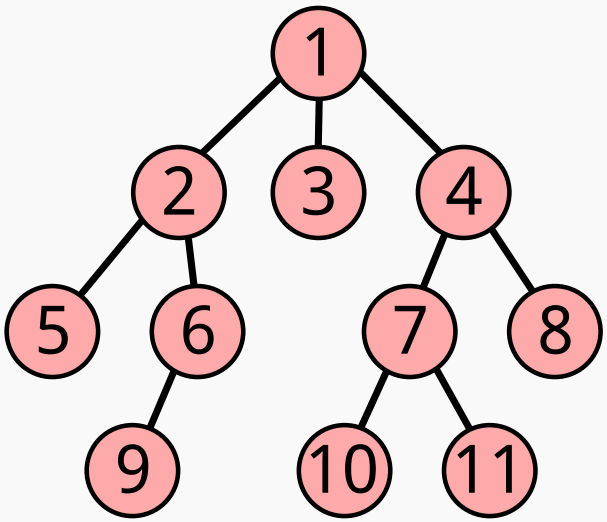
```
1   print "mengunjungi curNode"
2   visited[curNode] = true
3   for each adjNode ∈ adj(curNode)
4       if not visited[adjNode]
5           DFS(adjNode)
```

Implementasi DFS (Stack)

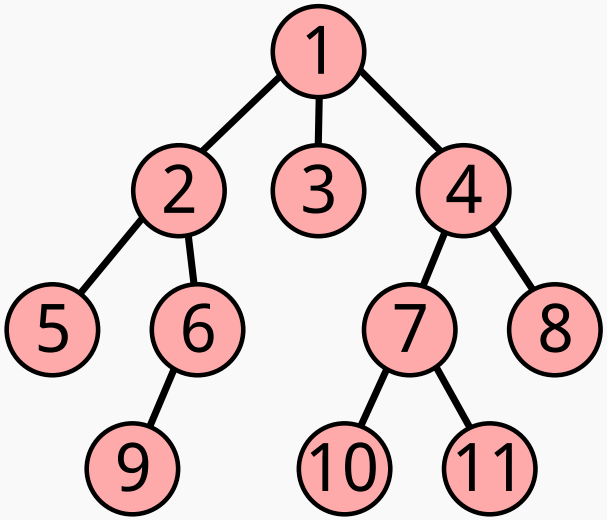
```
DFS()
 1  // Inisialisasi stack sebagai stack kosong.
 2  stack.push(initialNode)
 3  visited[initialNode] = true
 4  while not stack.empty()
 5        curNode = stack.top()
 6        stack.pop()
 7        print "mengunjungi curNode"
 8        visited[adjNode] = true
 9        for each adjNode ∈ adj(curNode)
10              if not visited[adjNode]
11                    stack.push(adjNode)
```

BFS: Breadth-First Search

Penelusuran *node* pada graf dilakukan lapis demi lapis.
Semakin dekat suatu *node* dengan *node* awal, *node* tersebut akan dikunjungi terlebih dahulu.



Penelusuran BFS



- Angka pada gambar menunjukkan urutan *node* tersebut dikunjungi.
- Dalam pemrograman, BFS biasa diimplementasikan dengan bantuan struktur data *queue*.

Implementasi BFS

```
BFS()
 1  // Inisialisasi queue sebagai queue kosong.
 2  queue.push(initialNode)
 3  visited[initialNode] = true
 4  while not queue.empty()
 5        curNode = queue.front()
 6        queue.pop()
 7        print "mengunjungi curNode"
 8        for each adjNode ∈ adj(curNode)
 9              if not visited[adjNode]
10                    visited[adjNode] = true
11                    queue.push(adjNode)
```

Analisis Kompleksitas

- Baik DFS maupun BFS sama-sama mengunjungi setiap *node* tepat satu kali, dengan memanfaatkan seluruh *edge*.
- Kompleksitas dari kedua metode adalah:
  - $O(V^2)$, jika digunakan *adjacency matrix*.
  - $O(V + E)$, jika digunakan *adjacency list*.

- Penggunaan DFS atau BFS dapat disesuaikan dengan persoalan yang dihadapi.

Contoh Permasalahan (lanj.)

Pak Dengklek tinggal di kota A. Suatu hari, beliau ingin pergi ke kota B. Terdapat beberapa ruas jalan yang menghubungkan kota-kota dalam negara tempat beliau tinggal. Namun karena sudah tua, Pak Dengklek ingin melewati sesedikit mungkin ruas jalan untuk sampai ke kota B.

Diberikan informasi mengenai struktur kota dan ruas jalan, tentukan berapa banyak ruas jalan yang perlu beliau lewati untuk pergi dari kota A ke kota B!

Solusi

- Permasalahan ini dapat diselesaikan dengan BFS.
- Karena sifat BFS yang menelusuri *node* lapis demi lapis, maka dapat disimpulkan jika suatu *node* dikunjungi, maka jarak yang ditempuh dari awal sampai *node* tersebut pasti jarak terpendek.
- Hal ini selalu benar untuk segala graf tak berbobot, BFS selalu menemukan *shortest path* dari suatu *node* ke seluruh *node* lainnya.

Implementasi Solusi

```
SHORTESTPATH(A, B)
 1  // Inisialisasi queue sebagai queue kosong.
 2  // Inisialisasi array visitTime dengan -1.
 3  queue.push(A)
 4  visitTime[A] = 0
 5  while not queue.empty()
 6        curNode = queue.front()
 7        queue.pop()
 8        for each adjNode ∈ adj(curNode)
 9              // Jika adjNode belum pernah dikunjungi...
10              if visitTime[adjNode] == −1
11                    visitTime[adjNode] = visitTime[curNode] + 1
12                    queue.push(adjNode)
13  return visitTime[B]
```

Bagian 3

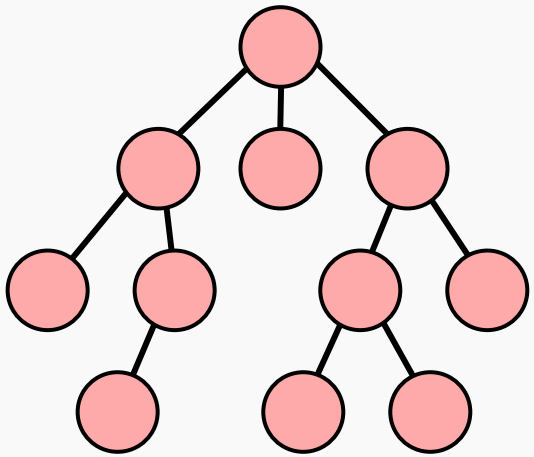
Macam-Macam Graf

Macam-Macam Graf

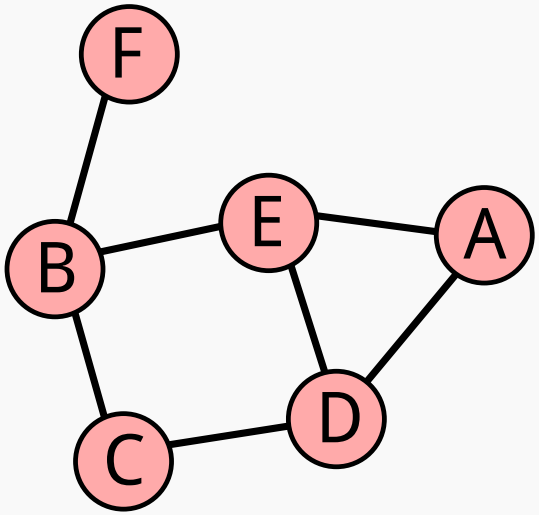
- Terdapat Graf yang memiliki suatu karakteristik khusus, sehingga penyelesaian masalah yang melibatkan graf ini dapat memanfaatkan karakter tersebut.
- Contoh macam-macam graf adalah **tree**, **directed acyclic graph**, dan **bipartite graph**.
- Kali ini kita akan menyinggung *tree* dan *directed acyclic graph*.

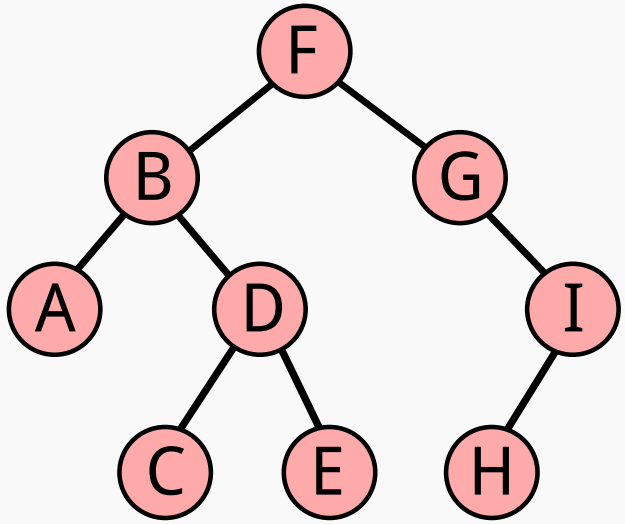
Tree

- *Tree* merupakan bentuk khusus dari graf.
- Seluruh *node* pada *tree* terhubung (tidak ada *node* yang tidak dapat dikunjungi dari *node* lain) dan tidak terdapat cycle.
- Banyaknya *edge* dalam sebuah *tree* pasti $V - 1$, dengan V adalah banyaknya *node*.



Contoh Tree

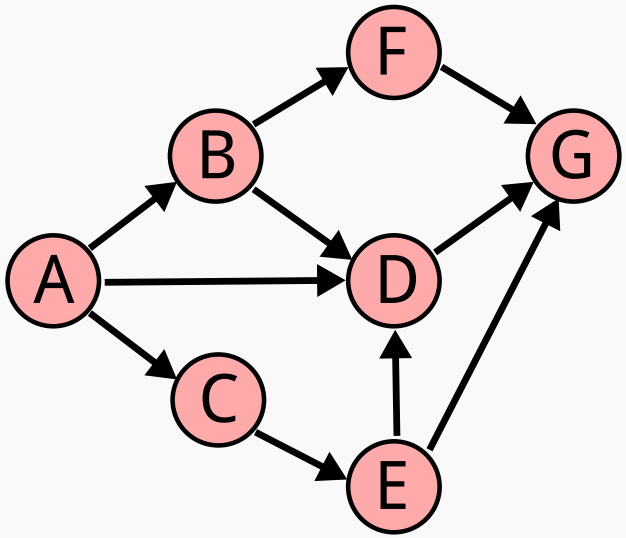




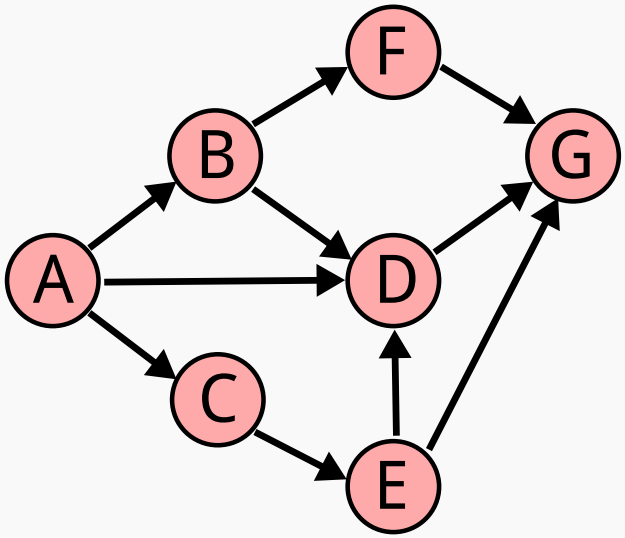
Gambar kiri bukan *tree* karena memiliki *cycle*, sedangkan gambar kanan merupakan *tree*.

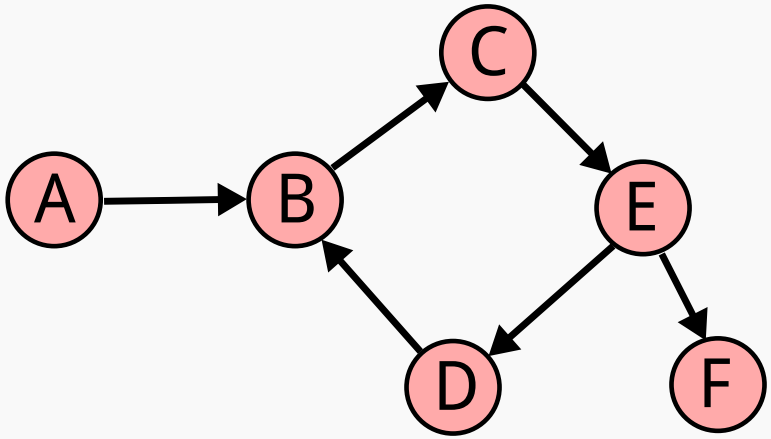
Directed Acyclic Graph

- *directed acyclic graph (DAG)* merupakan bentuk khusus dari *directed* graf.
- DAG tidak memiliki cycle.
- Berbeda dengan *tree* yang mana setiap *node* harus dapat dikunjungi dari *node* lainnya, sifat tersebut tidak berlaku pada DAG.



Contoh Directed Acyclic Graph





Pada gambar di atas, gambar kiri merupakan DAG, sedangkan gambar kanan bukan DAG karena memiliki *cycle*.

Penutup

- Hampir setiap kompetisi pasti memiliki soal yang bertemakan graf.
- Mampu menguasai dasar penyelesaian masalah graf menjadi kemampuan penting dalam dunia kompetisi.