Chapter 11
Basic of graphs

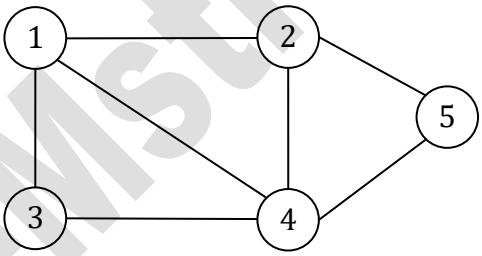
Banyak masalah pemrograman dapat diselesaikan dengan memodelkan masalah sebagai masalah graf dan menggunakan algoritma graf yang sesuai. Contoh khas dari graf adalah jaringan jalan dan kota dalam suatu negara. Namun, terkadang graf tersembunyi dalam masalah dan mungkin sulit untuk mendeteksinya.

Bagian buku ini membahas algoritma graf, dengan fokus khusus pada topik-topik yang penting dalam *competitive programming*. Dalam bab ini, kita akan membahas konsep-konsep yang berkaitan dengan graf dan mempelajari berbagai cara untuk merepresentasikan graf dalam algoritma.

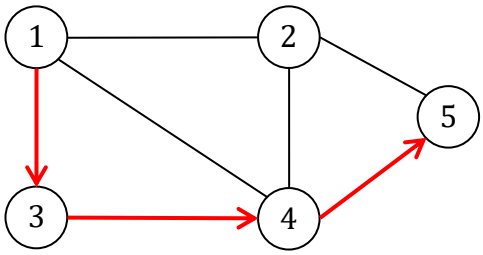
Terminologi Graf

Sebuah **graf** terdiri dari **node** dan **edge**. Dalam buku ini, variabel n menyatakan jumlah node dalam graf, dan variabel m menyatakan jumlah edge. Node diberi nomor menggunakan bilangan bulat $1,2,\ldots,n$.

Sebagai contoh, graf berikut terdiri dari 5 node dan 7 edge:



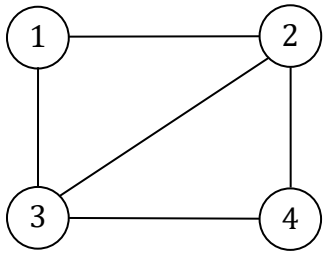
Sebuah **path** menghubungkan node a ke node b melalui edge-edge dalam graf. Panjang atau **length** dari sebuah *path* adalah jumlah edge di dalamnya. Sebagai contoh, graf di atas mengandung *path* $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ dengan panjang 3 dari node 1 ke node 5:



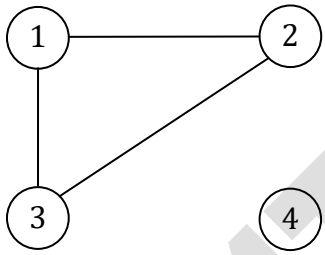
Sebuah *path* disebut *cycle* jika node pertama dan terakhir adalah sama. Sebagai contoh, graf di atas mengandung *cycle* $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. Sebuah *path* disebut **simple** jika setiap node muncul paling banyak satu kali dalam *path* tersebut.

Connectivity

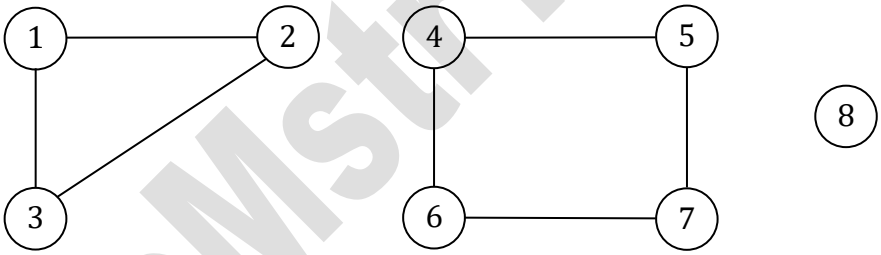
Sebuah graf disebut **connected** jika terdapat *path* antara setiap pasangan node. Sebagai contoh, graf berikut adalah *connected*:



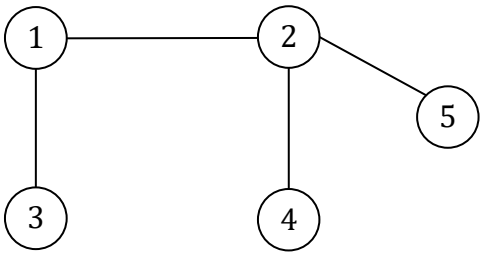
Graf berikut tidak *connected*, karena tidak memungkinkan untuk mencapai node 4 dari node lainnya:



Bagian-bagian *connected* dari sebuah graf disebut **components**. Sebagai contoh, graf berikut mengandung tiga *components*: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$, dan $\{8\}$.

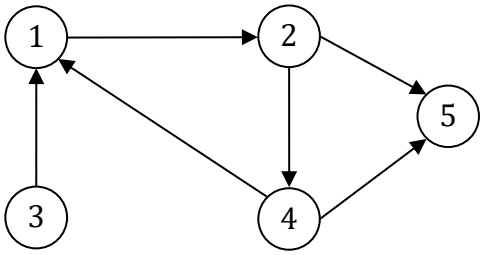


Sebuah **tree** adalah graf *connected* yang terdiri dari n node dan $n - 1$ edge. Terdapat tepat satu *path* yang unik antara setiap pasangan node dalam sebuah *tree*. Sebagai contoh, graf berikut adalah sebuah *tree*:



Edge Directions

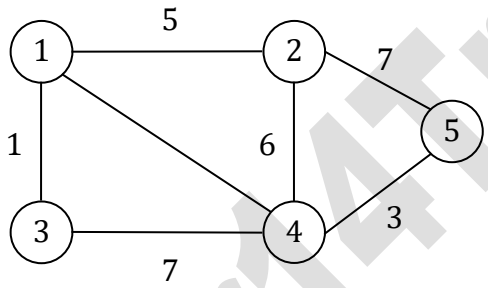
Sebuah graf disebut **directed** jika edge hanya dapat dilalui dalam satu arah saja. Sebagai contoh, graf berikut adalah *directed*:



Graf di atas mengandung *path* $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ dari node 3 ke node 5, tetapi tidak ada *path* dari node 5 ke node 3.

Edge Weights

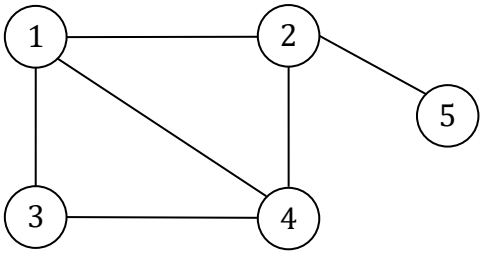
Dalam sebuah **weighted** graph, setiap edge memiliki **weight**. *Weight* ini sering diinterpretasikan sebagai panjang edge. Sebagai contoh, graf berikut adalah *weighted*:



Panjang sebuah *path* dalam *weighted graph* adalah jumlah dari *weight* semua edge pada *path* tersebut. Sebagai contoh, dalam graf di atas, panjang *path* $1 \rightarrow 2 \rightarrow 5$ adalah 12, dan panjang *path* $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ adalah 11. *Path* yang terakhir adalah **shortest path** dari node 1 ke node 5.

Neighbors and Degrees

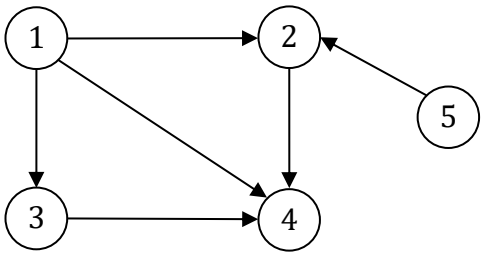
Dua node disebut **neighbors** atau **adjacent** jika terdapat sebuah edge di antara mereka. **Degree** dari sebuah node adalah jumlah *neighbors*-nya. Sebagai contoh, dalam graf berikut, *neighbors* dari node 2 adalah 1, 4, dan 5, sehingga *degree*-nya adalah 3.



Jumlah total *degree* dalam sebuah graf selalu $2m$, di mana m adalah jumlah *edges*, karena setiap *edge* meningkatkan *degree* dari tepat dua node sebanyak satu. Oleh karena itu, jumlah total *degree* dalam sebuah graf selalu genap.

Sebuah graf disebut **regular** jika setiap node memiliki *degree* yang sama, yaitu suatu konstanta d. Sebuah graf disebut *complete* jika setiap node memiliki *degree* $n - 1$, yaitu graf tersebut mengandung semua *edge* yang mungkin di antara node-node.

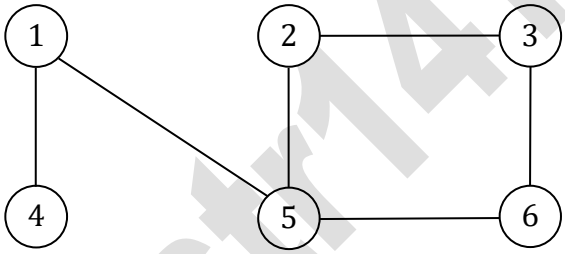
Dalam *directed graph*, **indegree** dari sebuah node adalah jumlah *edges* yang berakhir pada node tersebut, sedangkan **outdegree** dari sebuah node adalah jumlah *edges* yang berawal dari node tersebut. Sebagai contoh, dalam graf berikut, *indegree* dari node 2 adalah 2, dan *outdegree* dari node 2 adalah 1.



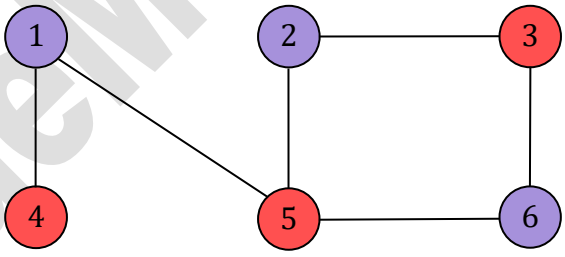
Colorings

Dalam sebuah **coloring** pada graf, setiap node diberi warna sedemikian rupa sehingga tidak ada dua *adjacent nodes* yang memiliki warna yang sama.

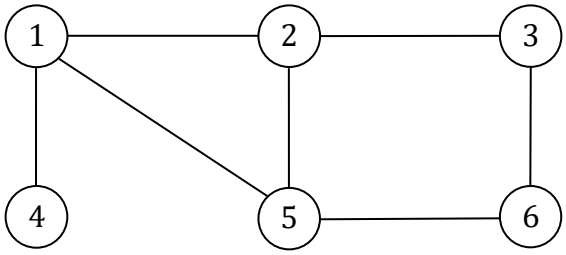
Sebuah graf disebut **bipartite** jika memungkinkan untuk mewarnainya menggunakan dua warna. Ternyata, sebuah graf adalah *bipartite* jika dan hanya jika graf tersebut tidak mengandung *cycle* dengan jumlah *edges* yang ganjil. Sebagai contoh, graf berikut:



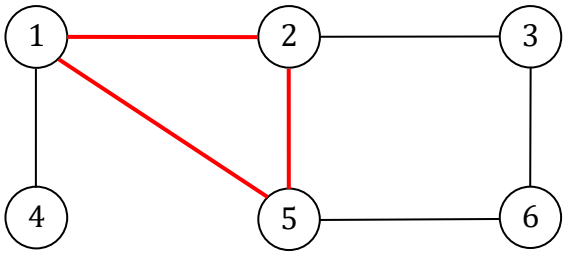
adalah *bipartite*, karena dapat diwarnai sebagai berikut:



Namun, graf berikut:



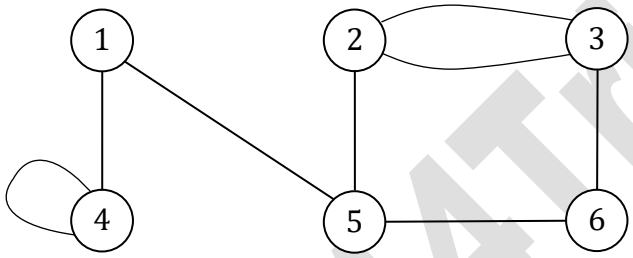
bukan *bipartite*, karena tidak memungkinkan untuk mewarnai *cycle* yang terdiri dari tiga node berikut menggunakan dua warna:



Simplicity

Sebuah graf disebut **simple** jika tidak ada *edge* yang dimulai dan berakhir pada node yang sama, serta tidak ada beberapa *edges* antara dua node yang sama. Sering kali, kita mengasumsikan bahwa graf adalah *simple*.

Sebagai contoh, graf berikut bukan *simple*:



Graph Representation

Terdapat beberapa cara untuk merepresentasikan graf dalam algoritma. Pemilihan struktur data bergantung pada ukuran graf dan cara algoritma memprosesnya. Selanjutnya, kita akan membahas tiga representasi umum.

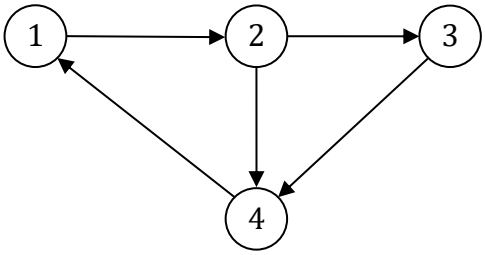
Adjacency List Representation

Dalam representasi *adjacency list*, setiap node x dalam graf diberikan sebuah **adjacency list** yang berisi node-node yang terhubung dengan *edge* dari x. *Adjacency lists* adalah cara yang paling populer untuk merepresentasikan graf, dan sebagian besar algoritma dapat diimplementasikan secara efisien menggunakan metode ini.

Cara yang nyaman untuk menyimpan *adjacency lists* adalah dengan mendeklarasikan sebuah array dari *vectors* sebagai berikut:

```
vector<int> adj[N];
```

Konstanta *N* dipilih agar semua *adjacency lists* dapat disimpan. Sebagai contoh, graf berikut:



dapat disimpan sebagai berikut:

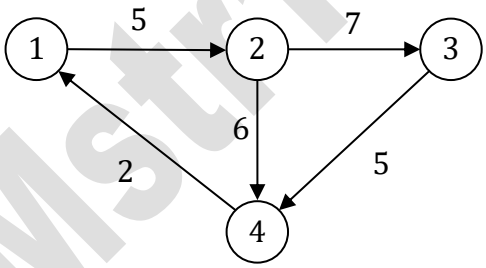
```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Jika graf bersifat *undirected*, penyimpanannya serupa, tetapi setiap *edge* ditambahkan dalam kedua arah.

Untuk *weighted graph*, struktur ini dapat diperluas sebagai berikut:

```
vector<pair<int, int>> adj[N];
```

Dalam kasus ini, *adjacency list* dari node a akan berisi pasangan (b, w) setiap kali terdapat *edge* dari node a ke node b dengan *weight* w. Sebagai contoh, graf berikut:



dapat disimpan sebagai berikut:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Keuntungan menggunakan *adjacency lists* adalah kita dapat dengan efisien menemukan node-node yang dapat dicapai dari suatu node melalui sebuah *edge*.

Sebagai contoh, *loop* berikut akan melewati semua node yang dapat dicapai dari node s:

```
for (auto u : adj[s]) {
    // process node u
}
```

Adjacency Matrix Representation

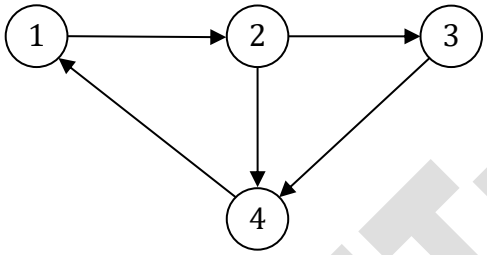
Sebuah **adjacency matrix** adalah array dua dimensi yang menunjukkan *edges* yang ada dalam graf. Dengan *adjacency matrix*, kita dapat dengan efisien memeriksa apakah terdapat *edge* antara dua node.

Matriks ini dapat disimpan sebagai sebuah array:

```
int adj[N][N];
```

Setiap nilai $\texttt{adj}[a][b]$ menunjukkan apakah terdapat *edge* dari node a ke node b. Jika *edge* tersebut ada dalam graf, maka $\texttt{adj}[a][b] = 1$, dan jika tidak ada, maka $\texttt{adj}[a][b] = 0$.

Sebagai contoh, graf berikut:

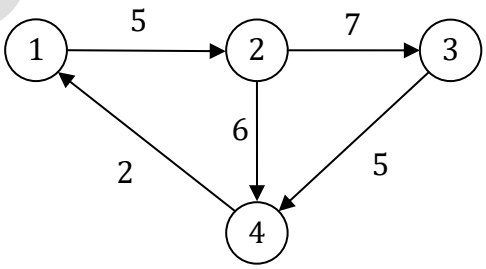


dapat direpresentasikan sebagai berikut:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Jika graf memiliki *weight*, representasi *adjacency matrix* dapat diperluas sehingga matriks menyimpan *weight* dari *edge* jika *edge* tersebut ada.

Dengan menggunakan representasi ini, graf berikut:



sesuai dengan matriks berikut:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Kelemahan dari representasi *adjacency matrix* adalah matriks tersebut memiliki n^2 elemen, dan biasanya sebagian besar bernilai nol. Oleh karena itu, representasi ini tidak dapat digunakan jika graf berukuran besar.

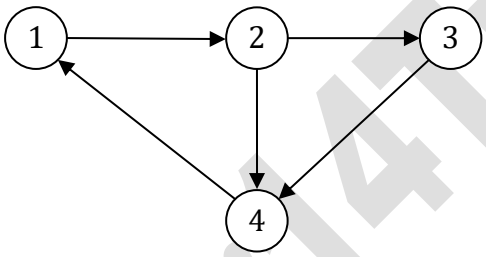
Edge List Representation

Sebuah **edge list** menyimpan semua *edges* dalam sebuah graf dalam urutan tertentu. Ini adalah cara yang nyaman untuk merepresentasikan graf jika algoritma memproses semua *edges* dalam graf dan tidak perlu mencari *edges* yang berawal dari suatu node tertentu.

Edge list dapat disimpan dalam sebuah *vector*:

```
vector<pair<int, int>> edges;
```

Setiap pasangan (a, b) menunjukkan bahwa terdapat sebuah *edge* dari node a ke node b. Dengan demikian, graf berikut:



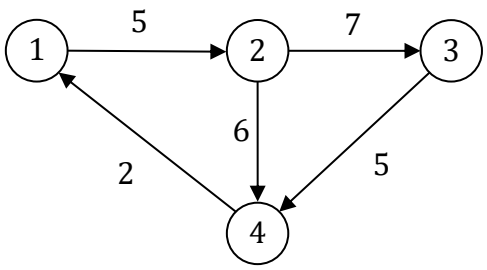
dapat direpresentasikan sebagai berikut:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

Jika graf memiliki *weight*, struktur ini dapat diperluas sebagai berikut:

```
vector<tuple<int, int, int>> edges;
```

Setiap elemen dalam daftar ini berbentuk (a, b, w), yang berarti terdapat *edge* dari node a ke node b dengan *weight* w. Sebagai contoh, graf berikut:



dapat direpresentasikan sebagai berikut:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```