Materi GRAPH

Graph adalah struktur data yang terdiri dari kumpulan simpul (vertice) dan sisi (edge) yang menghubungkan simpul-simpul tersebut. Graph dapat digunakan untuk merepresentasikan berbagai macam masalah dalam kehidupan nyata, seperti jejaring sosial, peta, jaringan komputer, dll2.

Ada beberapa jenis graph yang perlu Anda ketahui, yaitu:

- Undirected graph: graph yang sisi-sisinya tidak memiliki arah, sehingga simpul-simpul yang terhubung dapat dijelajahi dari dua arah.
- Directed graph: graph yang sisi-sisinya memiliki arah, sehingga simpul-simpul yang terhubung hanya dapat dijelajahi dari satu arah.
- Weighted graph: graph yang sisi-sisinya memiliki nilai atau bobot yang menunjukkan biaya, jarak, waktu, dll.
- Unweighted graph: graph yang sisi-sisinya tidak memiliki nilai atau bobot.
- Cyclic graph: graph yang memiliki setidaknya satu siklus atau jalur tertutup.
- Acyclic graph: graph yang tidak memiliki siklus atau jalur tertutup.

Selain itu, ada beberapa operasi atau algoritma yang dapat dilakukan pada graph, seperti:

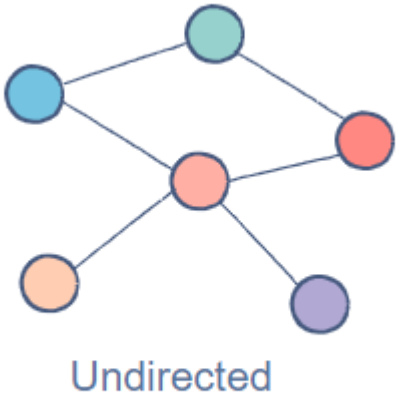
- Menambah atau menghapus simpul atau sisi3.
- Menelusuri graph dengan menggunakan metode Depth-First Search (DFS) atau Breadth-First Search (BFS)4.
- Menentukan jalur terpendek antara dua simpul dengan menggunakan algoritma Dijkstra, Bellman-Ford, Floyd-Warshall, dll4.
- Menentukan apakah graph bersifat terhubung, bipartit, planar, eulerian, hamiltonian, dll4.

Pengertian Graph

Graph adalah jenis struktur data umum yang susunan datanya tidak berdekatan satu sama lain (non-linier). Graph terdiri dari kumpulan simpul berhingga untuk menyimpan data dan antara dua buah simpul terdapat hubungan saling keterkaitan.

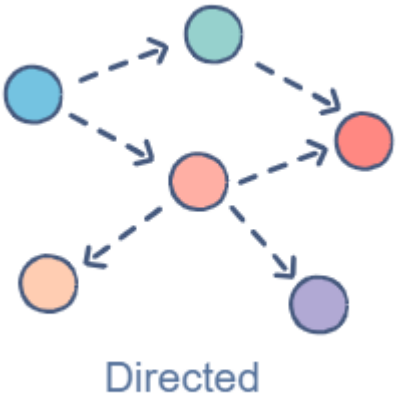
Jenis-jenis Graph

1. **Undirected Graph**



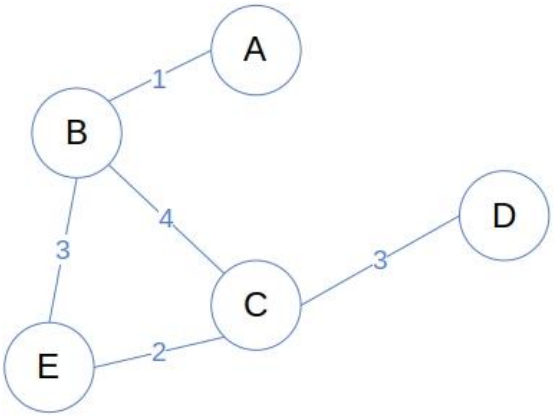
Pada undirected graph, simpul-simpulnya terhubung dengan edge yang sifatnya dua arah. Misalnya kita punya simpul 1 dan 2 yang saling terhubung, kita bisa menjelajah dari simpul 1 ke simpul 2, begitu juga sebaliknya.

2. **Directed Graph**



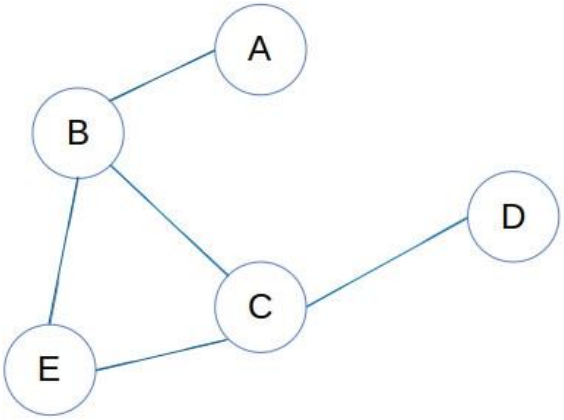
Directed graph, graph jenis ini simpul-simpulnya terhubung oleh edge yang hanya bisa melakukan jelajah satu arah pada simpul yang ditunjuk. Sebagai contoh jika ada simpul A yang terhubung ke simpul B, namun arah panahnya menuju simpul B, maka kita hanya bisa melakukan jelajah (traversing) dari simpul A ke simpul B, dan tidak berlaku sebaliknya.

3. **Weighted Graph**



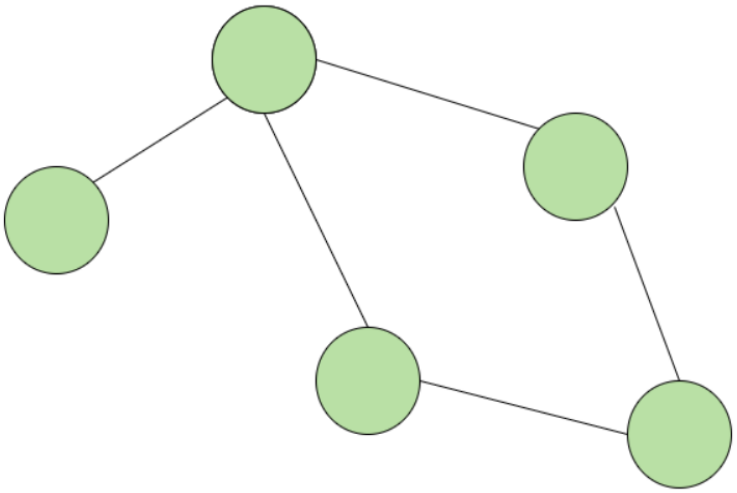
Weighted graph adalah jenis graph yang cabangnya diberi label bobot berupa bilangan numerik. Pemberian label bobot pada edge biasanya digunakan untuk memudahkan algoritma dalam menyelesaikan masalah.

4. **Unweighted Graph**



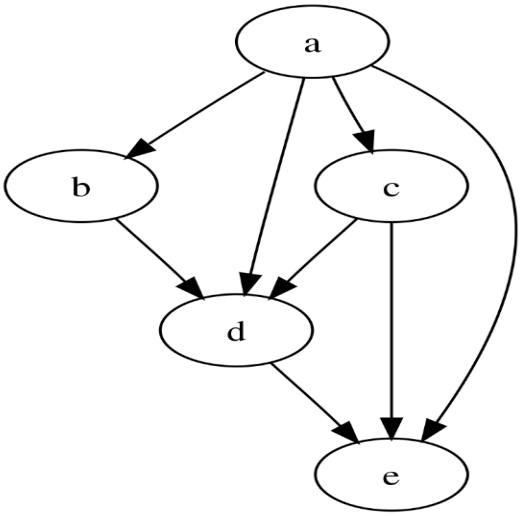
Unweighted graph dalam struktur data adalah jenis graph yang tidak memiliki bobot atau nilai numerik pada sisi-sisinya. Sisi-sisi pada unweighted graph hanya menunjukkan adanya hubungan atau keterhubungan antara simpul-simpulnya.

5. Cyclic Graph



Cyclic graph adalah graph yang mengandung setidaknya satu cycle, yaitu jalur yang dimulai dan berakhir di simpul yang sama, tanpa melewati simpul lain dua kali. Secara formal, cyclic graph didefinisikan sebagai graph G = (V, E) yang mengandung setidaknya satu cycle, di mana V adalah himpunan simpul (node) dan E adalah himpunan sisi (link) yang menghubungkannya. Cyclic graph dapat berupa directed atau undirected. Pada directed cyclic graph, sisi-sisi memiliki arah, dan cycle harus mengikuti arah sisi-sisi tersebut. Pada undirected cyclic graph, sisi-sisi tidak memiliki arah, dan cycle dapat bergerak ke arah mana saja.

6. Acyclic Graph



Acyclic graph adalah graph yang tidak memiliki cycle, yaitu jalur yang dimulai dan berakhir di simpul yang sama, tanpa melewati simpul lain dua kali. Acyclic graph itu bipartite, yaitu dapat dibagi menjadi dua himpunan simpul sedemikian rupa sehingga tidak ada sisi yang menghubungkan simpul dalam himpunan yang sama

Jenis Algoritma Pada Graph

1. BFS (Breadth-First Search):

   Algoritma ini digunakan untuk menjelajahi atau mencari jalur dalam graf secara meluas, yaitu dengan mengunjungi semua tetangga suatu simpul sebelum beralih ke simpul lain yang lebih dalam.

2. DFS (Depth-First Search):

   Algoritma ini digunakan untuk menjelajahi atau mencari jalur dalam graf secara dalam, yaitu dengan mengunjungi satu cabang pohon jalur sebanyak mungkin sebelum kembali dan mencari cabang lain.

3. Dijkstra:

   Algoritma ini digunakan untuk menemukan jalur terpendek antara dua simpul dalam graf berbobot positif.

4. Floyd-Warshall:

   Algoritma ini digunakan untuk mencari jalur terpendek antara semua pasangan simpul dalam graf berbobot positif atau negatif.

5. Ford-Fulkerson:

   Algoritma ini digunakan dalam teori aliran jaringan untuk mencari aliran maksimum di antara dua simpul dalam graf berbobot.

6. Kruskal:

   Algoritma ini digunakan untuk mencari pohon rentang minimum (Minimum Spanning Tree) dalam graf berbobot positif.

7. Prim:

   Algoritma ini juga digunakan untuk mencari pohon rentang minimum dalam graf berbobot positif, tetapi dengan pendekatan yang berbeda dari Kruskal.

Operasi-operasi Pada Graph

1. `addNode(Object data)`: menambahkan simpul baru ke graph dengan data tertentu

   Contoh code pada bahasa java:

```java
// Metode untuk menambahkan simpul baru ke graph
  public void addNode(Object data) {
    // Buat simpul baru dengan data tertentu
    Node newNode = new Node(data);
    // Tambahkan simpul baru ke daftar simpul
    nodes.add(newNode);
  }
```

2. `addEdge(Object source, Object destination)`: menambahkan sisi baru ke graph yang menghubungkan simpul source dan destination.

   Contoh code pada bahasa java:

```java
// Metode untuk menambahkan sisi baru ke graph
  public void addEdge(Object source, Object destination) {
    // Cari simpul source dan destination di daftar simpul
    Node sourceNode = findNode(source);
    Node destinationNode = findNode(destination);
    // Jika simpul source dan destination ditemukan
    if (sourceNode != null && destinationNode != null) {
      // Tambahkan simpul destination ke daftar simpul yang terhubung dengan simpul source
      sourceNode.adjList.add(destinationNode);
      // Tambahkan simpul source ke daftar simpul yang terhubung dengan simpul destination
      destinationNode.adjList.add(sourceNode);
    }
  }
```

3. `removeNode(Object data)`: menghapus simpul dari graph yang memiliki data tertentu.

   Contoh code pada bahasa java:

```java
// Metode untuk menghapus simpul dari graph
  public void removeNode(Object data) {
    // Cari simpul dengan data tertentu di daftar simpul
    Node node = findNode(data);
    // Jika simpul ditemukan
    if (node != null) {
      // Iterasi setiap simpul di daftar simpul
      for (Node n : nodes) {
        // Hapus simpul dari daftar simpul yang terhubung dengan simpul lain
        n.adjList.remove(node);
      }
      // Hapus simpul dari daftar simpul
      nodes.remove(node);
    }
  }
```

4. `removeEdge(Object source, Object destination)`: menghapus sisi dari graph yang menghubungkan simpul source dan destination.

   Contoh code dalam bahasa java:

```java
// Metode untuk menghapus sisi dari graph
  public void removeEdge(Object source, Object destination) {
    // Cari simpul source dan destination di daftar simpul
    Node sourceNode = findNode(source);
    Node destinationNode = findNode(destination);
    // Jika simpul source dan destination ditemukan
    if (sourceNode != null && destinationNode != null) {
      // Hapus simpul destination dari daftar simpul yang terhubung dengan simpul source
      sourceNode.adjList.remove(destinationNode);
      // Hapus simpul source dari daftar simpul yang terhubung dengan simpul destination
      destinationNode.adjList.remove(sourceNode);
    }
```

5. `findNode(Object data)`: mencari simpul di graph yang memiliki data tertentu dan mengembalikan simpul tersebut jika ditemukan, atau null jika tidak ditemukan.

   Contoh code pada bahasa java:

```java
// Metode untuk mencari simpul di graph
 public Node findNode(Object data) {
   // Iterasi setiap simpul di daftar simpul
   for (Node node : nodes) {
     // Jika simpul memiliki data yang sama dengan data yang dicari
     if (node.data.equals(data)) {
       // Kembalikan simpul tersebut
       return node;
     }
   }
   // Jika simpul tidak ditemukan, kembalikan null
   return null;
 }
```

6. `findEdge(Object source, Object destination)`: mencari sisi di graph yang menghubungkan simpul source dan destination dan mengembalikan sisi tersebut jika ditemukan, atau null jika tidak ditemukan.

   Contoh code pada bahasa java:

```java
// Metode untuk mencari sisi di graph
  public boolean findEdge(Object source, Object destination) {
    // Cari simpul source dan destination di daftar simpul
    Node sourceNode = findNode(source);
    Node destinationNode = findNode(destination);
    // Jika simpul source dan destination ditemukan
    if (sourceNode != null && destinationNode != null) {
      // Cek apakah simpul destination ada di daftar simpul yang terhubung dengan simpul source
      return sourceNode.adjList.contains(destinationNode);
    }
    // Jika simpul source atau destination tidak ditemukan, kembalikan false
    return false;
  }
```

7. `traverse(Object start, String method)`: menelusuri graph dari simpul start dengan metode tertentu.

Contoh code pada bahasa java:

```java
// Metode untuk menelusuri graph
 public List<Node> traverse(Object start, String method) {
   // Buat daftar untuk menyimpan simpul yang dikunjungi
   List<Node> visited = new ArrayList<>();
   // Cari simpul start di daftar simpul
   Node startNode = findNode(start);
   // Jika simpul start ditemukan
   if (startNode != null) {
     // Jika metode adalah BFS
     if (method.equals("BFS")) {
       // Buat antrian untuk menyimpan simpul yang akan dikunjungi
       Queue<Node> queue = new LinkedList<>();
       // Masukkan simpul start ke antrian
       queue.add(startNode);
       // Selama antrian tidak kosong
       while (!queue.isEmpty()) {
         // Keluarkan simpul dari antrian
         Node node = queue.poll();
         // Jika simpul belum dikunjungi
         if (!visited.contains(node)) {
           // Tambahkan simpul ke daftar simpul yang dikunjungi
           visited.add(node);
           // Iterasi setiap simpul yang terhubung dengan simpul tersebut
           for (Node n : node.adjList) {
             // Masukkan simpul yang terhubung ke antrian
             queue.add(n);
           }
         }
       }
     }
```

Fungsi dan Kegunaan Graph

Fungsi dan kegunaan graph di antaranya:

- Graph digunakan untuk merepresentasikan aliran komputasi.
- Digunakan dalam pemodelan grafik.
- Graph dipakai pada sistem operasi untuk alokasi sumber daya.
- Google maps menggunakan graph untuk menemukan rute terpendek.
- Graph digunakan dalam sistem penerbangan untuk optimasi rute yang efektif.
- Pada state-transition diagram, graph digunakan untuk mewakili state dan transisinya.
- Di sirkuit, graph dapat digunakan untuk mewakili titik sirkuit sebagai node dan kabel sebagai edge.
- Graph digunakan dalam memecahkan teka-teki dengan hanya satu solusi, seperti labirin.
- Graph digunakan dalam jaringan komputer untuk aplikasi Peer to peer (P2P).
- Umumnya graph dalam bentuk DAG (Directed acyclic graph) digunakan sebagai alternatif blockchain untuk cryptocurrency. Misalnya crypto seperti IOTA

Kelebihan Graph:

Keunggulan dari struktur data graph adalah sbb:

- Dengan menggunakan graph kita dapat dengan mudah menemukan jalur terpendek dan tetangga dari node
- Graph digunakan untuk mengimplementasikan algoritma seperti DFS dan BFS.
- Graph membantu dalam mengatur data.
- Karena strukturnya yang non-linier, membantu dalam memahami masalah yang kompleks dan visualisasinya.

Kekurangan:

kekurangan dari struktur data graph di antaranya

- Graph menggunakan banyak pointer yang bisa rumit untuk ditangani.
- Memiliki kompleksitas memori yang besar.
- Jika graph direpresentasikan dengan adjacency matrix maka edge tidak memungkinkan untuk sejajar dan operasi perkalian graph juga sulit dilakukan.