Graf Lanjutan

Tim Olimpiade Komputer Indonesia

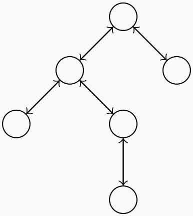
Pendahuluan

Melalui dokumen ini, kalian akan:

- Memahami Euler Tour Tree dan penggunaannya
- Memahami konsep Bridge, Articulation Point, dan Strongly Connected Component
- Memahami konsep Centroid Decomposition

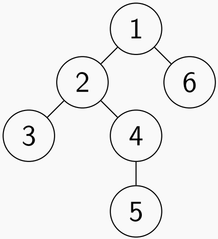
Euler Tour Tree

- **Euler Tour** pada sebuah graf adalah sebuah *trail* yang berawal di sebuah *node* dan berakhir di *node* yang sama dengan melalui setiap *edge* tepat satu kali.
- **Euler Tour Tree** adalah salah satu cara mepresentasikan sebuah *tree* dengan mengasumsikan sebuah jalan yang awalnya tidak berarah menjadi dua buah jalan berarah bolak balik, dan mencari *euler tour* yang dimulai dari *root*.



Euler Tour Tree (lanj.)

- Jika kita menomori seluruh node sesuai dengan urutan *node* yang dikunjungi pada *euler tour tree*, maka untuk setiap *node* u, nomor-nomor seluruh *node* yang berada pada *subtree node* u berurutan.
- Sebagai contoh, nomor-nomor seluruh *node* yang berada pada *subtree node* 2 pada ilustrasi berikut adalah $\{2, 3, 4, 5\}$.



Euler Tour Tree (lanj.)

- Karena itu, jika setiap *node* memiliki sebuah nilai, maka kita dapat menjumlahkan nilai seluruh *node* yang berada pada *subtree* sebuah *node* menggunakan data struktur yang menyelesaikan persoalan *range sum query*.
- Karena tidak terdapat *cycle*, menentukan urutan *node* yang dikunjungi pada *euler tour tree* dapat diperoleh dengan DFS sederhana.

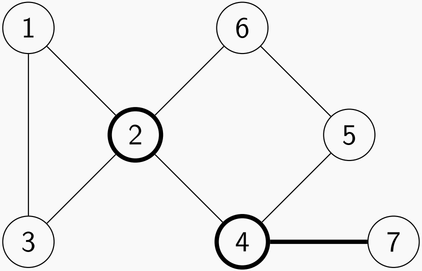
```
// berisi pemetaan node ke urutan kunjungan
// pada euler tour tree.
map<Node, int> nomor;

void dfs(Node u, Node parent) {
  nomor[u] = nomor.size();
  for (Node v : u.neighbours()) {
    if (v != parent) {
      dfs(v, u);
    }
  }
}
```

Graph Connectivity dalam graf tidak berarah

- Beberapa definisi dalam graf tidak berarah
  - **Connected component** adalah subgraf maksimal yang mana tiap dua vertexnya bisa dicapai satu sama lain melalui sebuah *path*.
  - Sebuah *edge* merupakan **bridge** apabila jika edge tersebut dihapus maka banyaknya *connected component* dari graf tersebut bertambah.
    - Dengan kata lain, tidak terdapat *path* lain yang menghubungkn dua *node* yang merupakan ujung dari *edge* tersebut.
  - Sebuah *node* merupakan **articulation point** apabila jika *node* tersebut dihapus maka banyaknya *connected component* dari graf tersebut bertambah.

Graph Connectivity dalam graf tidak berarah (lanj.)

- Sebagai contoh, pada graf berikut, *edge* yang dicetak tebal merupakan *bridge*, sedangkan *node* yang dicetak tebal merupakan *articulation point*.

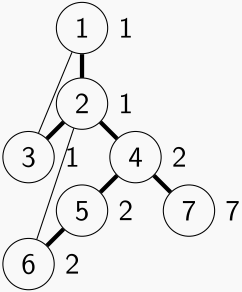


Graph Connectivity dalam graf tidak berarah (lanj.)

- Pada sebuah graf tidak berarah:
  - **DFS Tree** dari graf tersebut adalah tree yang dihasilkan dari menjalankan algoritma DFS pada graf tersebut.
  - Sebuah *edge* merupakan **tree edge** jika *edge* tersebut berada pada DFS *tree*.
  - Sebuah *edge* merupakan **back edge** jika *edge* tersebut menghubungkan sebuah *node* dengan leluhurnya.
    - Perhatikan bahwa seluruh *edge* yang bukan merupakan *tree edge* adalah *back edge*.
  - **Discovery time** dari sebuah *node* adalah urutan node tersebut dikunjungi pada DFS.
  - **Low link** dari sebuah *node* adalah *discovery time* terkecil yang bisa dicapai oleh sebuah *node* dengan *tree edge* ke anak-anaknya dan *back edge* paling banyak sekali.

Graph Connectivity dalam graf tidak berarah (lanj.)

- Sebagai contoh, ilustrasi berikut menunjukan DFS *tree* yang mungkin dihasilkan jika menjalankan algoritma DFS pada graf sebelumnya dimulai dengan *node* 1.
  - *edge* yang dicetak tebal merupakan *tree edge*, sedangkan yang tidak dicetak tebal merupakan *back edge*
  - Angka di dalam dan di luar *node* menyatakan *discovery time* dan *low link* dari node tersebut secara berurutan.



Bridge

- Mari kita notasikan *discovery time* dan *low link* dari *node* u sebagai $low(u)$ dan $num(u)$ secara berurutan.
- Sebuah *back edge* tidak mungkin merupakan *bridge*, karena kedua *node* yang merupakan ujung dari *edge* tersebut masih terhubung menggunakan *tree edge*.
- Sebuah *tree edge* yang menghubungkan *node* u dan v (dengan *node* u merupakan *parent* dari *node* v pada DFS *tree*) merupakan *bridge* jika $low(v) > num(u)$.
  - Jika $low(v) \leq num(u)$, maka terdapat *path* dari *node* v ke *node* u melalui *back edge* tanpa melalui *edge* yang menghubungkan kedua *node* tersebut.
- Algoritma ini memiliki kompleksitas $O(V + E)$.

Articulation point

- Jika sebuah edge menghubungkan *node* u dan v (dengan *node* u merupakan *parent* dari *node* v pada DFS *tree*), u bukan merupakan *root* dari DFS *tree*, dan $low(v) \geq num(u)$, maka *node* u merupakan *articulation point*.
  - $low(v) \geq num(u)$ berarti menghapus *node* u menyebabkan *node* v tidak terhubung dengan *parent* dari *node* u.
- Untuk setiap *node* u yang bukan merupakan *root*, jika $low(v) < num(u)$ untuk semua *node* v yang merupakan anak dari *node* u, maka *node* u bukan merupakan *articulation point*.
- *Root* dari DFS *tree* merupakan *articulation point* jika dan hanya jika ia memiliki lebih dari satu anak.
- Algoritma ini memiliki kompleksitas $O(V + E)$.

Contoh kode

- Berikut ini adalah contoh kode untuk mencari seluruh *bridge* dan *articulation point* pada sebuah graf.

```
void dfs(Node u, Node parent) {
  num[u] = low[u] = time++;
  for (Node v : adj[u]) {
    if (num[v] != UNDEFINED) {
      ++num_children[u];
      dfs(v, u);
      if (low[v] > num[u]) bridges.insert((u, v));
      if (low[v] >= num[u] && parent != UNDEFINED_NODE) {
        articulation_point.insert(u);
      }
      low[u] = min(low[u], low[v]);
    } else if (v != parent) {
      low[u] = min(low[u], num[v]);
    }
  }
}
```

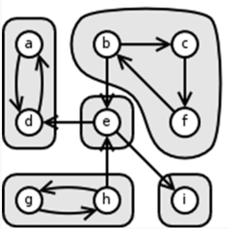
Contoh kode (lanj.)

- Berikut ini adalah contoh kode untuk mencari seluruh *bridge* dan *articulation point* pada sebuah graf (lanj.).

```
for (Node u : nodes) {
  if (num[u] == UNDEFINED) {
    dfs(u, UNDEFINED_NODE);
    if (num_children[u] > 1) {
      articulation_point.insert(root);
    }
  }
}
```

Graph Connectivity dalam graf berarah

- Dalam graf berarah, **strongly connected component** (SCC) adalah subgraf maksimal yang mana tiap dua vertexnya bisa dicapai satu sama lain melalui sebuah *path*.
- Sebagai contoh, pada graf di bawah, kumpulan *node* yang berada dalam satu daerah warna abu-abu merupakan SCC.
- Jika kita membuat graf berarah baru dengan membuat satu *node* untuk setiap SCC dan menambahkan *edge* dari *node* u ke *node* v jika terdapat *edge* dari salah satu *node* di SCC u ke salah satu *node* di SCC v, maka graf baru ini adalah sebuah DAG.



Algoritma Tarjan

- Untuk mencari seluruh SCC, kita dapat menggunakan algoritma Tarjan. Dapatkan DFS *tree* dari graf yang diberikan. Pada graf terarah, mungkin saja terdapat *edge* yang bukan merupakan *tree edge* ataupun *back edge*.
  - Sebuah *edge* bisa saja merupakan **forward edge**, yaitu *edge* yang berarah dari sebuah *node* ke salah satu keturunannya. Untuk mencari SCC, *edge* ini dapat kita abaikan karena tidak mempengaruhi hasil SCC.
  - Jika sebuah *edge* bukan merupakan *tree edge*, *back edge*, atau *forward edge*, maka *edge* ini merupakan **cross edge**. *Cross edge* pasti berarah dari sebuah *node* yang memiliki *discovery time* lebih tinggi ke sebuah *node* yang memiliki *discovery time* lebih rendah.

Algoritma Tarjan (lanj.)

- Apabila *node u* memiliki *back edge* ke *node v*, maka seluruh *node* pada *path* yang menghubungkan *node u* dan *node v* pada DFS *tree* pasti berada dalam satu SCC.
- Apabila *node u* memiliki *cross edge* ke *node v*, maka kita harus memeriksa apakah terdapat *path* dari *node v* ke *node u* juga.

Algoritma Tarjan (lanj.)

- Jalankan algoritma DFS. Pada akhir dari DFS di *node u*, jika semua *node* pada SCC yang berisi *node u* merupakan keturunan dari *node u*, maka kita ingin membuat SCC tersebut.
  - Untuk melakukan hal ini, sambil menjalankan algoritma DFS, kita ingin menjaga sebuah *stack* global S. Pada saat algoritma menjalakan DFS di *node u*, S berisi seluruh *node* yang dapat mengunjungi *node u*.
  - Pada DFS di *node u*, untuk setiap *cross edge* dari *node u* ke *node v*, jika *node v* berada pada stack S, maka kita dapat menganggap *edge* ini sebagai *back edge*, sehingga kita dapat memperbaharui nilai $low(u)$ dengan $\min(low(u), num(v))$.

Algoritma Tarjan (lanj.)

- Jalankan algoritma DFS. Pada akhir dari DFS di *node* u, jika semua *node* pada SCC yang berisi *node* u merupakan keturunan dari *node* u, maka kita ingin membuat SCC tersebut. (lanj.)
  - Pada akhir DFS di *node* u, jika $low(u) = num(u)$, ini berarti SCC yang berisi *node* u tidak berisi *node* lain yang bukan merupakan keturunan *node* u, sehingga kita dapat membuat sebuah SCC baru.
  - Masukkan *node* u dan seluruh *node* yang berada di atas *node* u pada stack S ke dalam satu SCC baru, dan keluarkan seluruh *node* tersebut dari S. Kumpulan *node* ini adalah seluruh *node* pada keturunan *node* u yang belum terdapat pada SCC manapun.
- Algoritma ini memiliki kompleksitas $O(V + E)$.

Contoh kode

- Berikut ini adalah contoh kode untuk mencari seluruh SCC.

```
void dfs(Node u) {
  num[u] = low[u] = time++;
  S.push(u); isInStack[u] = true;
  for (Node v : adj[u]) {
    if (num[v] != UNDEFINED) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
    } else if (isInStack[v]) {
      low[u] = min(low[u], num[v]);
    }
  }

  if (low[u] == num[u]) {
    set<Node> newScc;
    do {
      int s = stack.top(); scc.insert(s);
      stack.pop(); isInStack[s] = false;
    } while (s != u);
    sccs.insert(newScc);
  }
}
```

Algoritma Kosaraju

- Selain algoritma Tarjan, kita juga dapat menggunakan algoritma Kosaraju untuk mendapatkan seluruh SCC.
- Algoritma Kosaraju juga memiliki kompleksitas $O(V + E)$.
- Kita tidak akan membahas algoritma Kosaraju pada pembelajaran ini.

Contoh aplikasi: 2-SAT

- Persoalan 2-SAT adalah menentukan apakah ekspresi dalam bentuk konjungsi (*and*) dari beberapa klausa, dengan setiap klausa merupakan disjungsi (*or*) dari dua literal, dapat bernilai benar. Setiap literal bisa saja merupakan suatu variabel atau negasi dari suatu variabel.
  - Sebagai contoh, ekspresi $(x_2 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$ dapat bernilai benar. Salah satu caranya dengan memberikan x_1 dan x_2 nilai salah dan memberikan x_3 dan x_4 nilai benar.
  - Sebagai contoh lainnya, ekspresi $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$ tidak dapat bernilai benar untuk seluruh kemungkinan pemberian nilai x_1, x_2, dan x_3.

Contoh aplikasi: Solusi 2-SAT

- Persoalan 2-SAT dapat diselesaikan dengan algoritma berikut:
  - Buatlah sebuah graf baru G. Untuk setiap variabel x pada ekspresi yang diberikan, tambahkan *node* x dan $\neg x$ pada G.
  - Untuk setiap klausa $(a \vee b)$ (ekuivalen dengan $\neg a \Rightarrow b$ dan $\neg b \Rightarrow a$), tambahkan *edge* berarah dari *node* $\neg a$ ke *node* b dan dari *node* $\neg b$ ke *node* a pada G.
  - Ekspresi yang diberikan dapat bernilai benar jika dan hanya jika kita dapat memberikan nilai salah/benar kepada setiap *node* pada G dengan syarat:
    - *node* x dan *node* $\neg x$ harus bernilai berbeda, dan
    - untuk setiap *edge* dari *node* a ke *node* b, jika *node* a bernilai benar, maka *node* b juga harus bernilai benar.
  - Jika terdapat sebuah SCC yang berisi *node* a dan *node* b, maka kedua node ini harus bernilai sama. Karenanya, jika terdapat sebuah SCC yang berisi *node* x dan *node* $\neg x$, maka ekspresi yang diberikan tidak dapat bernilai benar. Jika tidak, maka ekspresi yang diberikan dapat bernilai benar.

Centroid Decomposition

- Pada sebuah *tree* berisi N *node*, **centroid** dari *tree* tersebut adalah sebuah *node* yang jika *node* tersebut dihapus, seluruh *tree* sisanya memiliki paling banyak $\frac{N}{2}$ *node*. Mencari *centroid* dapat dilakukan dalam kompleksitas $O(N)$.
- **Centroid decomposition** (penguraian *centroid*) adalah proses mencari *centroid* pada sebuah tree, menghapus *centroid* tersebut dari *tree*, lalu melakukan proses yang sama pada seluruh *tree* yang dihasilkan secara rekursif.
  - Karena setiap penghapusan *centroid* dari *tree* menyisakan *tree* yang memiliki *node* paling banyak setengah dari banyak *node* awal, proses rekursif ini memiliki kedalaman paling banyak $\lceil \log_2 N \rceil$.
  - Untuk setiap tingkat kedalaman, jumlah banyaknya *node* pada *tree* yang tersisa tidak lebih dari N. Sehingga, proses penguraian *centroid* dapat dilakukan dalam kompleksitas $O(N \log N)$.

Centroid Decomposition: Contoh penggunaan

- Diberikan sebuah *tree* berbobot dan bilangan bulat K. Kita ingin menghitung berapa pasang *node* yang dihubungkan oleh *path* yang memiliki jarak (total bobot) tepat K.
- Persoalan ini dapat diselesaikan menggunakan penguraian *centroid*:
    - Misalkan *node* c adalah *centroid* dari *tree* tersebut. Kita akan menghitung berapa pasang *node* yang dihubungkan oleh *path* yang melewati *node* c dan memiliki jarak K. Asumsikan menghapus *node* C menghasilkan dua *tree* terpisah T_1 dan T_2. Kita ingin menghitung berapa pasang *node* (a, b) sedemikian sehingga $a \in T_1$, $b \in T_2$, dan jumlah jarak *node* c dan *node* a dan jarak *node* c dan *node* b adalah K. Hal ini dapat dihitung dalam kompleksitas $O(N)$.
    - Berikutnya, hapus *node* c dan lakukan algoritma ini secara rekursif pada seluruh *tree* yang tersisa untuk juga menghitung banyaknya pasang *node* yang dihubungkan oleh *path* yang tidak melewati *node* c.
    - Kompleksitas algoritma ini adalah $O(N \log N)$.