inzva Algorithm Programme 2018-2019

Bundle 11

Graph - 3

Editor
Ekrem BAL

Reviewers
Bahadır ALTUN
Yasin KAYA
Burak BUĞRUL

Contents

1 Bridges and Articulation Points

1.1 DFS order

Dfs order verilen bir çizgeyi root düğümünü sabitleyerek dfs algoritmasındaki ile aynı şekilde fakat keşfedilmiş bir düğüme bir daha gitmeyerek tüm düğümleri gezmektir. Burada önemli bir gözlem ise kullandığımız kenarlar ve düğümler bir **tree** yapısı oluşturacaktır. Çünkü her düğüme(**root hariç**) yalnızca başka bir düğümden geldiğimiz için ve **root** düğümüne ise herhangi başka bir düğümden gelmediğimizden bu yapı bir **tree** yapısı oluşturur.

```
1  void dfs(int node){
2          used[node] = true;
3          for(auto it : g[node])
4                  if(!used[it])
5                          dfs(it);
6  }
```

1.2 Edge Çeşitleri

Bir çizgeyi dfs order ile gezerken görülebilecek birkaç tane kenar tipi vardır. Bu kenarlar bazı çizge algoritmalarını anlamamızda bize çok faydalı olacaktır.

Edge çeşitleri:

- **Tree edge** : Bu kenarlar çizgeyi gezerken kullandığımız ana kenarlardır.
- **Forward edge** : Bu kenarlar daha önce gittiğimiz ve kendi alt ağacımızda bulunan bir düğüme giden kenarlardır.
- **Back edge** : Bu kenarlar daha önce uğradığımız ama dfs işlemini tamamlamamış düğümlere giden kenarlardır.
- **Cross edge** : Bu kenarlar daha önce uğradığımız ve dfs işlemini tamamlamış düğümlere giden kenarlardır.

Bu kenarlar için önemli bir gözlem ise çift-yönlü bir çizgede **cross edge** olmasının imkansız olduğudur. Çünkü dfs işlemini tamamlamış bir düğümden çıkan bir kenarın gezilmemesi olanaksızdır.

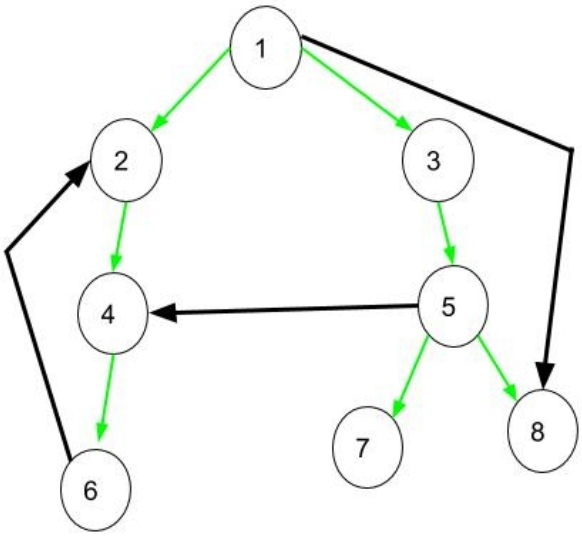

Figure 1: Yeşil renkli kenarler **tree edge**. (1,8) **forward edge**. (6,4) **back edge**. (5,4) **cross edge**.

1.3 Bridge

Yönsüz ve connected bir çizgede eğer bir kenar çıkardığımızda o çizge connected olmaktan çıkıyorsa bu kenara **bridge** denmektedir.

1.4 Bridge Bulma

Bridge bulmak için birden fazla algoritma olmasına rağmen (Chain decomposition gibi) aralarında en kolay yazılabilen ve hızlı olan Tarjan'ın algoritmasını ele alacağız.

Bir çizgeyi dfs ile gezerken eğer bir kenarın aşağıdaki ucunun alt ağacından yukarı çıkan bir **back edge** var ise, o kenar **bridge** değildir. Çünkü **back edge**, bu kenarı çıkardığımızda çizgenin alt ağacının ve atalarının ayrılmasını engellemektedir.

Bu algoritma da tam olarak buna dayanarak her düğüm için o düğümün alt ağacındaki **back edge**lerin gittiği minimum derinliği tutmaktadır.

Eğer bir kenarın alt ucunun alt ağacındaki **back edge**lerin gittiği minimum derinlik kenarın üst ucunun derinliğinden büyük veya eşitse bu kenar **bridge**dir. Çünkü bu kenarın alt ucunun alt ağacındaki hiçbir **back edge**, şuanki kenarımızın üstünde bir düğüme çıkmamaktadır. Dolayısıyla bu kenarı çıkardığımızda çizgenin alt ağacı ve ataları birbirinden ayrılır.

Tarjanın algoritmasıyla çizgedeki bridge'leri O(V + E) zaman karmaşıklığında bulabiliriz. Burada V çizgedeki düğüm sayısını, E ise çizgedeki kenar sayısını temsil etmektedir.

```cpp
int dfs(int node, int parent, int depth){
        int minDepth = depth;
        dep[node] = depth; // dep dizisi her dugumun derinligini tutmaktadir.
        used[node] = true;
        for(auto it : g[node]){
                if(it == parent)
                        continue;
                if(used[it]){
                        minDepth = min(minDepth, dep[it]);
                        // Eger komsu dugum daha once kullanilmis ise
                        // Bu edge back edge veya forward edgedir.
                        continue;
                }
                int val = dfs(it, node, depth + 1);
                  //val degeri alt agacindan yukari cikan minimum derinliktir.
                if(val >= depth + 1)
                        bridges.push_back({node, it});
                minDepth = min(minDepth, val);
        }
        return minDepth;
}
```

1.5 Articulation Point

Yönsüz bir çizgede eğer bir düğümü çıkardığımızda o çizgenin connected çizge sayısı artıyorsa o düğüme **articulation point** veya **cut point** denir.

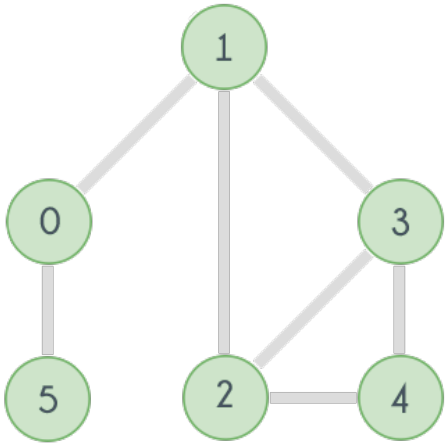

Figure 2: Örneğin 0 düğümünü çıkardığımızda düğümler 5 ve $1, 2, 3, 4$ olmak üzere iki gruba ayrılıyor. Aynı şekilde 1 düğümünü çıkardığımızda düğümler $5, 0$ ve $2, 3, 4$ olmak üzere iki gruba ayrılıyor. Bu nedenle 0 ve 1 düğümleri birer **articulation point**'dir.

1.6 Articulation Point Bulma

Articulation point'leri bulmak için kullanacağımız Tarjan'ın algoritması:

- Çizgeyi dfs order ile gez.
- Her düğüm için o düğümden ve alt ağacından yukarı çıkan kenarlardan minimum derinklikli düğüme gidenin derinliğini hesapla. Bu değere düğümün alt ağacından ulaşabileceğimiz minimum derinlik diyebiliriz. (Buna **low** değeri diyeceğiz.)
- Eğer root olmayan herhangi bir düğümün çocuklarından birinin **low** değeri şu anki düğümün derinliğinden büyük veya eşitse bu düğüm **articulation point**dir. Çünkü bu düğümün alt ağacındaki hiçbir **back edge**, şu anki düğümün üstünde bir düğüme çıkmamaktadır. Dolayısıyla bu düğümü çıkardığımızda bu düğümün alt ağacı ve ataları birbirinden ayrılır.
- Eger şu anki düğümümüz root(Dfs order'a başladığımız düğüm) ise, dfs ile gezerken birden fazla kez dallanmamız root'un **Articulation Point** olmasına sebep olur. Çünkü root'un bağlı olduğu birden fazla ayrık alt-çizgesi vardır.

Tarjanın algoritmasıyla çizgedeki articulation pointleri O(V + E) zaman karmaşıklığında bulabiliriz.

```
int dfs(int node, int parent, int depth){
        int minDepth = depth, children = 0;
        dep[node] = depth;// dep dizisi her dugumun derinligini tutmaktadir.
        used[node] = true;
        for(auto it : g[node]){
                if(it == parent)
                        continue;
                if(used[it]){
                        minDepth = min(minDepth, dep[it]);
                        continue;
                }
                int val = dfs(it, node, depth + 1);
                if(val >= depth and parent != -1)
                        isCutPoint[node] = true;
                minDepth = min(minDepth, val);
                children++;
        }
        //Bu if yukarida belirttigimiz root olma durumunu kontrol ediyor.
        if(parent == -1 and children >= 2)
                isCutPoint[node] = true;
        return minDepth;
}
```

2 Strong Connectivity and Biconnectivity

2.1 Strong Connectivity

Bir düğümden hedef bir düğüme ulaşılabilmesi için, sonlu sayıda düğümden geçerek hedef düğüme varılması gerekmektedir.

Yönsüz bir çizgede her düğümden diğer her düğüme ulaşılabiliyorsa bu çizgeye **connected** denir. Aynı konsept yönlü çizgelere uygulandığında bu sefer ismi **strongly connected** olmaktadır.

Yani bir yönlü çizgenin **strongly connected** olması için her düğümden diğer bütün düğümlere ulaşılabilmesi gerekmektedir.

2.2 Biconnectivity

Yönsüz bir çizgede herhangi bir düğüm silindiğinde eğer geriye kalan çizge **connected** oluyorsa bu çizgeye **biconnected** denir. Başka bir deyişle çizgede **articulation point** yoksa bu çizge **biconnected** bir çizge olmaktadır.

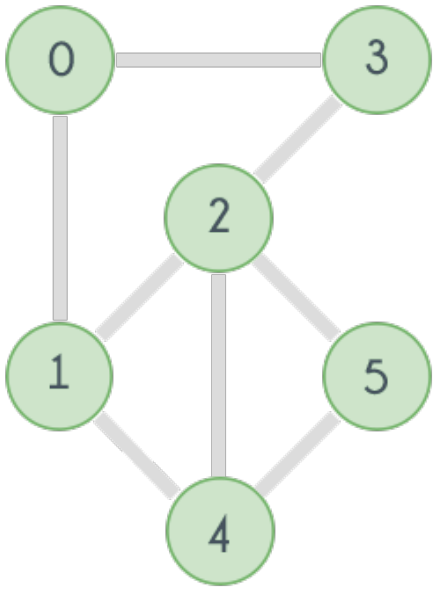

Figure 3: Biconnected bir graph örneği.

3 Connected Components

3.1 Strongly Connected Components

Bütün yönlü çizgeler **strongly connected** olan ayrık alt-çizgelere bölünebilir. İki alt-çizgenin ayrık olması için ortak kenar ve düğüm içermemesi gerekmektedir. Oluşan alt-çizgeleri tek bir düğüm olarak düşünüp yeni bir çizge oluşturursak, oluşan yeni çizge, içinde cycle olmayan yönlü bir çizge oluşturur.

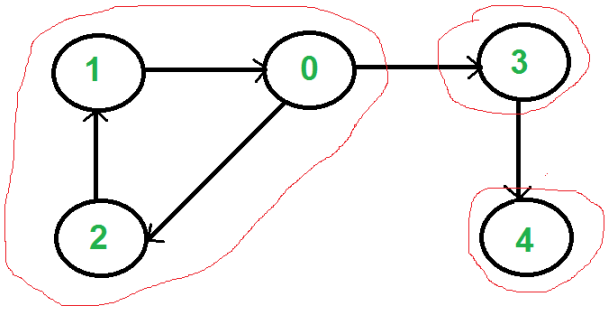

Figure 4: Kırmızı ile işaretli olan alt-çizgeler **strongly connected component**lerdir.

Figure 5: Oluşan yeni çizge içinde cycle olmayan yönlü bir çizge oluşturur.

Strongly connected componentleri bulmak için kullanacağımız Tarjan'ın algoritması(Alternatif olarak Kosaraju Algoritması da bulunmaktadır fakat pratikte daha yavaş olduğundan ve anlamasının zor olmasından kaynaklı tarjanın algoritmasını ele alacağız):

- Herhangi bir düğümden dfs order ile gezmeye başla ve uğradığı düğümleri bir stack(yığın)'a at. Her düğümün keşfedilme zamanını hesapla. (Keşfedilme zamanı, dfs order ile gezerken düğüme ilk geldiğimiz birim zamandır ve buna **index** diyeceğiz)

- Eğer bir düğüm stack'in içindeyse henüz herhangi bir strongly connected component'e ait değildir. Bunun sebebi ise strongly connected component bulduğumuz zaman stack'deki strongly connected componente ait olan düğümleri çıkarmamızdır.

- Her düğüm için o düğümden ve alt ağacından yukarı çıkan, herhangi bir strongly connected component'e ait olmayan düğümlere giden kenarlardan, minimum indexe sahip düğüme gidenin indexini hesapla. Bu değere düğümün alt ağacından ulaşabileceğimiz minimum derinlik diyebiliriz. (Buna **low** değeri diyeceğiz.)

- Eger bir düğümün low değeri kendi indexine eşit ise bu düğüm ve bu düğümün altındaki stack'deki tüm düğümler strongly connected component'dir. Bunun sebebi ise eğer bu düğüme **u** dersek; **u**'nun alt-ağacından kendisine gelen bir kenar mutlaka olmak zorundadır çünkü olmaması durumunda **u**'nun low değerinin kendi index'inden daha küçük bir değer olacağı barizdir.

- Strongly connected component bulunduğunda(bir önceki adımda nasıl bulunduğu açıklanmıştır) stack'den şuanki strongly connected componente ait tüm düğümleri çıkar.

Tarjanın algoritmasıyla çizgedeki strongly connected compenentleri O(V + E) zaman karmaşıklığında bulabiliriz.

```
void dfs(int node){
        low[node] = index[node] = ++curTime;
        // curTime her dugumun kesfedilme zamanini bulmamizi sagliyor.
        used[node] = true;

        st.push(node);
        inStack[node] = true;
    // Bir dugumun stackte olup olmadigini inStack dizisiyle tutmamiz gerekiyor.
        for(auto it : g[node]){
                if(!used[it]){
                        dfs(it);
                        low[node] = min(low[node], low[it]);
                }
                else if(inStack[it])
                        low[node] = min(low[node], index[it]);
//Eger komsu dugum stack icindeyse o zaman bu edge back edge olabilir.
        }
        if(low[node] == index[node]){
                while(1){
                        int x = st.top();
                        st.pop();
                        cout << x << " ";
                        inStack[x] = false;
                        if(x == node)
                                break;
                }
                cout << endl;
        }
}

void scc() {
        for(int i = 0; i < n; i++)
                if(!used[i])
                        dfs(i);
}
```

4 Cycle Finding

4.1 Cycle bulma

Cycle: başladığı düğüme geri dönen, her düğümü en fazla bir defa gezen ve en az 2 düğüm içeren düğüm dizisidir.

Bir çizgede cycle olup olmadığını bulmak için Dfs order kullanabiliriz.

Eğer Dfs order ile gezerken back edge ile karşılaşırsak o zaman çizgede cycle vardır. Çünkü **back edge** üst ve alt ucundaki düğümü birbirine bağlayıp cycle oluşmasına sebep olmaktadır.

Yönlü bir çizgede cycle bulmak için kullancağımız algoritma:

- Çizgeyi dfs order ile gez.
- Bir düğüme geldiğinde o düğümü griye boya ve komşularını gezmeye başla.
- Eğer şu anki düğümün komşularından bir tanesi gri ise o zaman çizgede cycle vardır. Çünkü gri olan bir düğüm kesinlikle şu anki düğümün atalarındandır ve kendi atalarından bir düğüme giden bir kenarın back edge olduğu kesindir.
- Komşuları gezmeyi tamamladıktan sonra düğümü siyaha boya.

```
bool dfs(int node){
    //color dizizi her nodeun rengini tutuyor.
    //Eger color degeri 0 ise beyaz, 1 ise gri, 2 ise siyah rengini temsil ediyor.
        color[node] = 1;
        for(int i = 0; i < g[node].size(); i++){
                int child = g[node][i];
                if(color[child] == 1)
                        return true;
                if(!color[child])
                        if(dfs(child))
                                return true;
        }
        color[node] = 2;
        return false;
}
```

5 Max Flow

5.1 Flow Network

Flow network bir adet **kaynak** ve bir adet **hedef** içeren özel bir yönlü çizge çeşididir.

Flow network çizgesinde her kenarın bir kapasitesi vardır. Bu kapasiteler bu kenarden geçebilecek flow(akış) miktarını belirtmektedir.

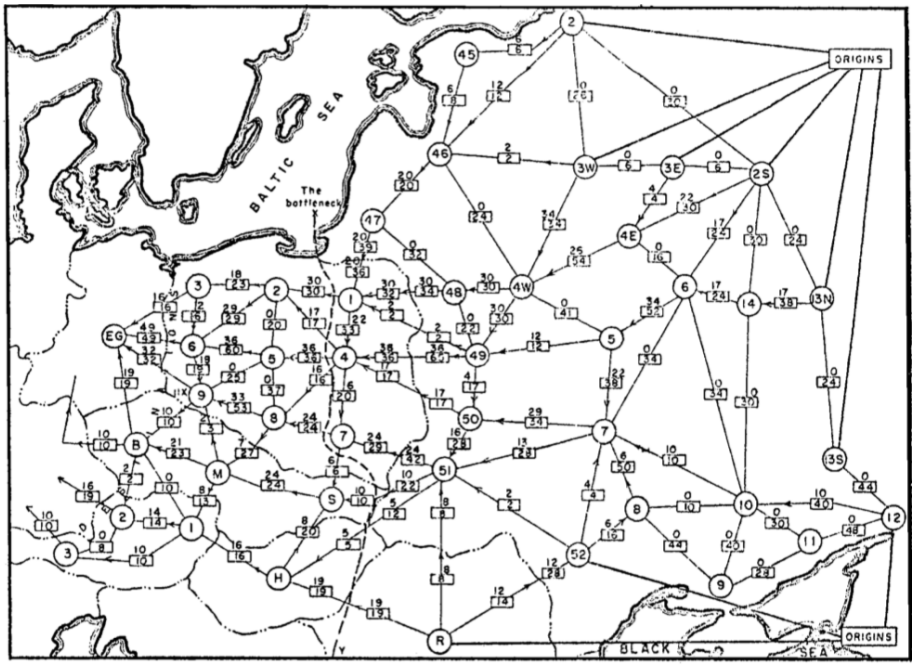

Figure 6: Tarihteki ilk flow network örneklerinden biri.

5.2 Maximum Flow

Maximum flow belirli bir flow networkünde kaynaktan hedefe devamlı bir akış varken birim zamanda hedefe ulaşabilecek maximum akış miktarını hesaplayan algoritmadır.

Maximum Flow problemini çözmeye yarayan bir çok algoritma vardır. Bunlardan popüler olanların zaman karmaşıklıkları:

- Ford-Fulkerson algoritması : $O(E * flowSayisi)$
- Edmonds–Karp algoritması : $O(V * E^2)$
- Dinic algoritması : $O(E * V^2)$

5.3 Ford Fulkerson

Ford Fulkerson'un maximum flow algoritmasının adımları şöyledir:

- Kaynaktan hedefe bir yol bul.
- Bulduğumuz yoldaki minimum kapasiteli kenar bu yoldan geçebilecek flowa eşittir.
- Yoldaki kenarların kapasitelerini bulduğumuz flow(adım 2 de bulduğumuz minimum kapasite) kadar eksilt ve kullandığımız kenarların tersini bulduğumuz flow kadar kapasiteli olacak şekilde çizgeye ekle.
- Kaynaktan hedefe yol kalmayana kadar devam et.

Peki bu neden çalışıyor?

Örneğin **u**'dan **v**'ye olan kenarı içeren **x** büyüklüğünde bir flow bulduğumuzu varsayalım.

Bulduğumuz yol $a \to ... \to u \to v \to ... \to b$ olsun.

Çizgemize **v**'den **u**'ya **x** kapasitelli yeni bir kenar ekleyeceğiz fakat bulduğumuz bu kenar orjinal çizgemizde bulunmuyor.

Ters kenarları ekledikten sonra bulduğumuz yeni yol da $c \to ... \to v \to u \to ... \to d$ olsun ve bulunan flow da **y** olsun.

$y \leq x$ olduğu barizdir.

$a \to ... \to u \to ... \to d$ yolunu izleyen **y** büyüklüğünde bir flow,

$c \to ... \to u \to ... \to b$ yolunu izleyen **y** büyüklüğünde bir flow,

$a \to ... \to u \to v \to ... \to d$ yolunu izleyen **x - y** büyüklüğünde bir flow olacak şekilde 3 farklı geçerli flow şeklinde gösterebiliriz.

Ford Fulkersenin genel çalışma karmaşıklığı $O(E * flowSayisi)$ çünkü en kötü durumda her yol bulduğunda flowu 1 arttırır. Her yol bulma işlemi de kenar sayısı kadar olduğuna göre karmaşıklık $O(E * flowSayisi)$ olur. Fakat Ford Fulkersen algoritmasını kodlarken BFS kullanırsak o zaman karmaşıklık; her kenar için o kenarı minimum kabul eden flowların sürekli artacak olmasından dolayı $O(V * E^2)$ oluyor. Bu implementasyona Edmond-Karp Algoritması deniyor.

Figure 7: Soldaki figür her kenardan ne kadar flow geçtiğini gösteriyor. Sağdaki ise çizgenin şu anki halini gösteriyor.

Figure 8: Flow = 7

Figure 9: Flow = 8

Figure 10: Flow = 13

Figure 11: Flow = 15

```cpp
//c matrisi her kenarin kapasitesini tutuyor.
//g komsuluk listesi ise cizgeyi gezmemizi sagliyor.
bool bfs(){
        vector < bool > visited(n, false);
        queue < int > q;
        q.push(source);
        visited[source] = true;
        while(!q.empty()){
                int node = q.front();
                q.pop();
                if(node == sink)
                        break;
                for(int i = 0; i < g[node].size(); i++){
                        int child = g[node][i];
                        if(c[node][child] <= 0 or visited[child])
                                continue;
                        visited[child] = true;
                        parent[child] = node;
                        q.push(child);
                }
        }
        return visited[sink];
}
int max_flow() {
    while(bfs()){
        int curFlow = -1, node = sink;
        while(node != source){
        //curFlow su anki yoldaki minimum kapasite yani buldugumuz flowdur.
                int len = c[parent[node]][node];
                if(curFlow == -1)
                        curFlow = len;
                else
                        curFlow = min(curFlow, len);
                node = parent[node];
        }
        flow += curFlow;
        node = sink;
        while(node != source){
                c[parent[node]][node] -= curFlow;
                //Buldugumuz yoldan buldugmuz flowu eksiltiyoruz.
                c[node][parent[node]] += curFlow;//Kenarlarin tersini ekliyoruz.
                node = parent[node];
        }
    }
    return flow;
}
```

References

[1] Finding bridges in a graph in O(N+M). CP-Algorithms.

[2] Articulation Points (or Cut Vertices) in a Graph. Geeksforgeeks website.

[3] Tree, Back, edge and Cross edges in DFS of Graph. Geeksforgeeks website.

[4] Cycle detection. Wikipedia, the free online encyclopedia.

[5] Maximum flow from hackerearth.

[6] Ford-Fulkerson Algorithm for Maximum Flow Problem. Geeksforgeeks website.

[7] Tarjan's Algorithm to find Strongly Connected Components. Geeksforgeeks website.

[8] Strongly Connected Components. Hackerearth website.

[9] Articulation Points and Bridges. Hackerearth website.

[10] Connected component (graph theory). Wikipedia, the free online encyclopedia.

[11] Tarjan's strongly connected components algorithm. Wikipedia, the free online encyclopedia.

[12] Flow Network. Wikipedia, the free online encyclopedia.

[13] On the history of the transportation and maximum flow problems

[14] Edmonds–Karp algorithm. Wikipedia, the free online encyclopedia.

[15] Biconnected graph. Wikipedia, the free online encyclopedia.