Data structure [B14]

김종규, PhD

2017-06-07

Transpose of a graph G^T

- Adjacency matrix 를 transpose 한 것
- $G^T = (V, E^T)$
  - $E^T = \{(v, u) | (u, v) \in E\}$

What attributes a DFS vertex has?

- color, parent, name, node number, adjacency list
- d, f
- Compare it with BFS
  - color, parent, name, node number, adjacency list
  - d

Color Constant

```
WHITE = 0
GRAY = 1
BLACK = 2
```

Color contant (C)


```
#define WHITE 0
#define GRAY  1
#define BLACK 2
```

Vertex extension

```
class Vertex:
    def __init__(self):
        # color, parent, name, n, first
        ...
    def add(self, v):
        ...

class DFSVertex(Vertex):
    def __init__(self):
        super().__init__()
        self.d = 0
        self.f = 0
```

Vertex extension (C)

```
typedef struct {
  /* color, parent, name, n, first */
  ...
} Vertex;

void Vertex_add(Vertex* self, Vertex* v) {
  ...
}

typedef struct {
  Vertex super;
  int d, f;
} DFSVertex;
```

Global variable

```
class DepthFirstSearch:
    def __init__(self):
        self.time = 0;
        self.vertices = None
```

Global variable (C)

```
int dfs_time = 0;
```

Algorithm

```
class DepthFirstSearch:
    def dfs(self):
        for u in self.vertices:
            u.color = WHITE
            u.parent = -1
        self.time = 0
        for u in self.vertices:
            if u.color == WHITE:
                self.dfs_visit(u)
```

Algorithm (C)

```
void dfs(DFSVertex* vertices, int nelem) {
  for(int u = 0; u < nelem; u++) {
    vertices[u].super.color = WHITE;
    vertices[u].super.parent = -1;
  }
  dfs_time = 0;
  for(int u = 0; u < nelem; u++) {
    if (vertices[u].super.color == WHITE) {
      dfs_visit(vertices, nelem, u);
    }
  }
}
```

Algorithm

```
def dfs_visit(self, u):
    self.time = self.time + 1
    u.d = self.time
    u.color = GRAY
    v = u.first
    while v:
        if self.vertices[v.n].color == WHITE:
            self.vertices[v.n].parent = u.n
            self.dfs_visit(self.vertices[v.n])
        v = v.next;
    u.color = BLACK
    self.time = self.time + 1
    u.f = self.time
```

Algorithm (C)


```
void dfs_visit(DFSVertex* vertices, int nelem, int u)
  dfs_time++;
  vertices[u].d = dfs_time;
  vertices[u].super.color = GRAY;
  for (Adj* v = vertices[u].super.first; v; v = v->ne
    if (vertices[v->n].super.color == WHITE) {
      vertices[v->n].super.parent = u;
      dfs_visit(vertices, nelem, v->n);
    }
  }
  vertices[u].super.color = BLACK;
  dfs_time++;
  vertices[u].f = dfs_time;
```

Example

- 다음은 그래프 $G = (V, E)$ 의 adjacency list 를 출력한 것이다. G^T 의 adjacency list 를 출력하시오.

```
r:v s
s:w r
t:u x w
u:y x t
v:r
w:x t s
x:y u t w
y:u x
```

Strongly connected component

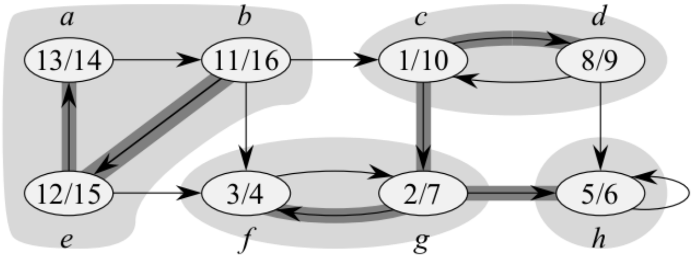

그림: Strongly connected component

Strongly connected component

STRONGLY-CONNECTED-COMPONENTS(G)

```
1  call DFS(G) to compute finishing times u.f for each vertex u
2  compute G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

그림: Strongly connected component algorithm

DFS result


```
    c  p  d  f
0:a 2 -1  1 16:b
1:b 2  0  2 15:e f c
2:c 2  1 11 14:g d
3:d 2  2 12 13:h c
4:e 2  1  3 10:f a
5:f 2  4  4  9:g
6:g 2  5  5  8:h f
7:h 2  6  6  7:h
```

Sorted by f, transpose


```
0 a 16:e
1 b 15:a
2 c 14:d b
3 d 13:c
4 e 10:b
5 f  9:g e b
6 g  8:f c
7 h  7:h g d
```

Strongly connected components

```
a b c d e f g h
0 1 2 3 4 5 6 7


0 a 16:e        a
1 b 15:a        b
2 c 14:d b         c
3 d 13:c           d
4 e 10:b        e
5 f  9:g e b         f
6 g  8:f c           g
7 h  7:h g d            h
```

Strongly connected component

```
def scc(self):
    self.dfs()
    self.transpose()
    sorted = self.sort_by_f()
    vset = self.vertices
    for v in vset:
        v.color = WHITE
        v.parent = -1
    for n in sorted:
        if self.vertices[n].color == WHITE:
            self.scc_find(vset[n])
```

Wrap-up

- Breadth first search (BFS) is useful for undirected graphs
  - We can find the shortest path of a graph based on BFS
- Depth first search (DFS) is useful for analyzing directed graph
  - We can find the strongly connected components of a graph based on DFS
- DFS and BFS can share many operations on V and E
  - We understand them as abstract data types
  - We implement them using Object Oriented Programming (OOP)
  - We can emulate OOP using structure and pointer in C