Graph traversal

DFS, BFS, applications

beOI Training

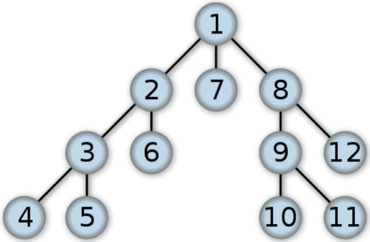
OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

January 29, 2016

Table of Contents

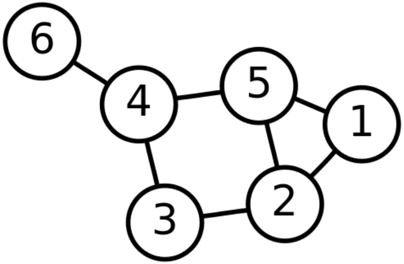
Depth-first principle

- Go in depth, backtrack when stuck
- Visit everything before switching



DFS example

Do not visit a node twice!



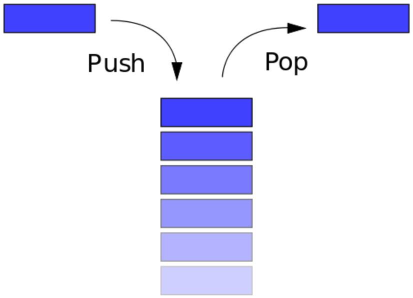
$$\mathbf{1} \rightarrow \mathbf{2} \rightarrow \mathbf{3} \rightarrow \mathbf{4} \rightarrow \mathbf{5} \rightarrow 4 \rightarrow \mathbf{6} \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1$$

DFS recursive implementation

```
vector<int> neigh[MAXN];
bitset<MAXN> visited;

void dfs(int u)
{
    if (visited[u])
        return;
    visited[u] = true;

    for (int i = 0; i < (int) neigh[u].size(); i++)
        dfs(neigh[u][i]);
}
```

DFS visit order

- Always travels locally: parent $\rightarrow$ child $\rightarrow$ parent
- The last discovered are the first visited $\Rightarrow$ we can also use a *Last In First Out* structure (stack)

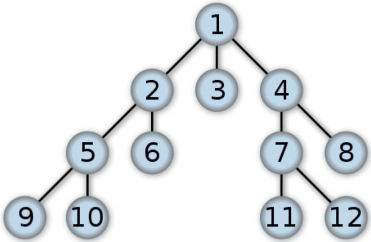


DFS stack implementation

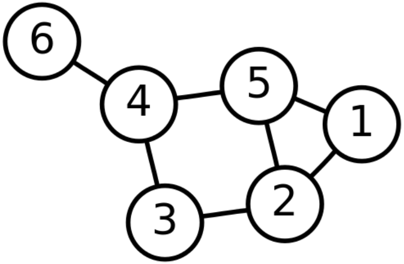
```
stack<int> st;
st.push(start);

while (!st.empty())
{
    int u = st.top();
    st.pop();

    for (int i = 0; i < (int) neigh[u].size(); i++)
    {
        int v = neigh[u][i];
        if (!visited[v])
        {
            visited[v] = true;
            st.push(v);
        }
    }
}
```

Breadth-first principle

- Go layer by layer
- Visit the closest nodes first



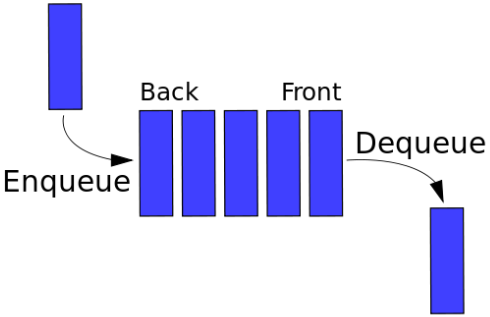
BFS example



$1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4 \rightarrow 6$

BFS visit order

The first discovered are the first visited $\Rightarrow$ we can use a *First In First Out* structure (queue)



BFS implementation

```
queue<int> q; // different
q.push(start);

while (!q.empty())
{
    int u = q.front(); // different
    q.pop();

    for (int i = 0; i < (int) neigh[u].size(); i++)
    {
        int v = neigh[u][i];
        if (!visited[v])
        {
            visited[v] = true;
            q.push(v);
        }
    }
}
```

BFS for shortest path

BFS traverses graph with nondecreasing distance!

- Add two tables `dist[]` and `parent[]`
- When adding to queue:
  - $\text{dist}[\text{v}] = \text{dist}[\text{u}] + 1$
  - $\text{parent}[\text{v}] = \text{u}$
- Trace path back using `parent[]`

Comparison

Complexity: both $O(V + E)$.

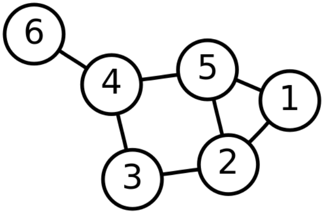
How to choose:

- If distance / shortest path necessary $\Rightarrow$ BFS
- Otherwise $\Rightarrow$ DFS (shorter)

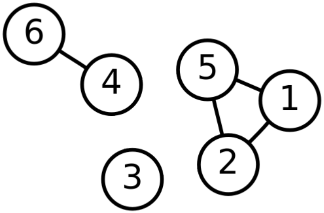
Table of Contents

Connected components

- On **undirected graphs**, "*u* connected to *v*" is an equivalence relation
- So it forms a partition of the vertices



1 connected component



3 connected components

Finding connected components

One run of DFS/BFS visits a whole component!

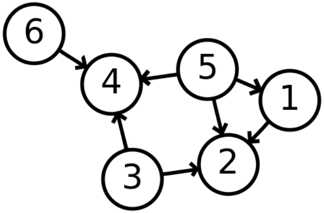
- Go through nodes and check if visited
- In `dfs()`, store nodes in a vector

```
for (int i = 0; i < n; i++)
{
    if (!visited[i]) // new CC
    {
        vector<int> cc;
        dfs(i, &cc); // adds nodes to cc
        // process cc
    }
}
```

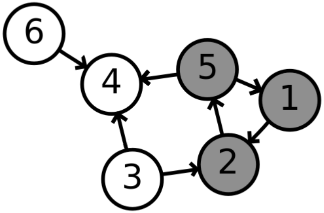
Table of Contents

Directed acyclic graphs

Directed graphs without cycles



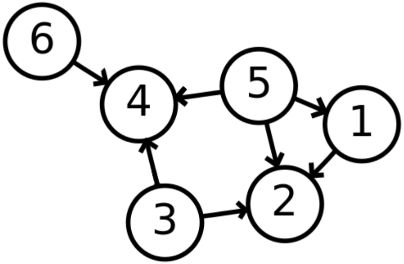
DAG



not DAG

Topological sort

- There are no cycles, so we can order the nodes so that edge $u \to v \Rightarrow u$ before v
- Example: course prerequisites, how to take them all in order
- Not unique: $\{3, 5, 6, 4, 1, 2\}, \{5, 3, 1, 6, 2, 4\}, \ldots$



Toposort with zero in-degree

If u does not have edges pointing to it, it can go first!

- Start at nodes with $\deg_{\text{in}} = 0$
- Remove edges as we find them
- Continue until no node is left

Zero in-degree implementation

```
int in_degree[MAXN];    // pre-filled
queue<int> zero_in;     // pre-filled
vector<int> toposort;

while (!zero_in.empty())
{
    int u = zero_in.front();
    zero_in.pop();
    toposort.push_back(u);

    for (int i = 0; i < (int) neigh[u].size(); i++)
    {
        int v = neigh[u][i];
        in_degree[v]--;
        if (in_degree[v] == 0)
            zero_in.push(v);
    }
}
```

Toposort with DFS

Use DFS to build the toposort backwards

- Call `dfs()` on every node
- Add a node *after* recursing

Thus every node is

- after its children in the *backward* toposort
- before its children in the *forward* toposort

Toposort DFS implementation

```
int dfs(int u, vector<int> *toposort)
{
    if (visited[u]) break;
    visited[u] = true;
    // recurse as usual...
    toposort->push_back(u);
}

vector<int> toposort;
for (int i = 0; i < n; i++)
    dfs(i, &toposort); // no need to check visited[i]

for (int i = n-1; i >= 0; i--)
    // use toposort[i]
```

Comparison

Complexity: both $O(V + E)$

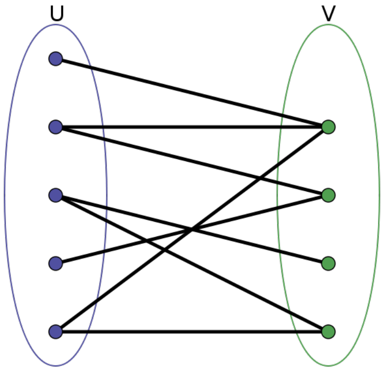
Use the zero in-degree method only if needed:

- Additional conditions on the order
- Enumerate all possible toposorts

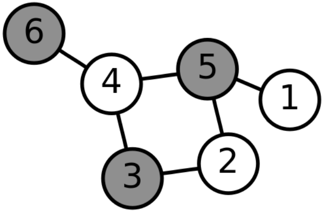
Table of Contents

Bipartite graphs

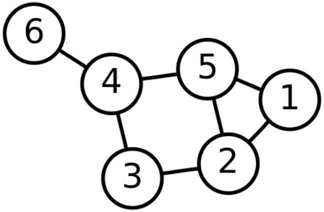
- Can be split in two without internal edges
- Equivalent: no odd-length cycle



Bipartite examples



bipartite



not bipartite

Bipartite check with DFS/BFS

Put an arbitrary node u on the left then traverse the graph:

- If v is on the left, put all its neighbors on the right
- If v is on the right, put all its neighbors on the left
- Continue until conflict or finished

Note: v on left $\Leftrightarrow \text{dist}(u, v)$ even

Bipartite check correctness

G is not bipartite $\Leftrightarrow$ the assignment fails

- If G is not bipartite the assignment will clearly fail
- If there is a conflict between u and v adjacent
  - Let p be their common parent in the DFS
  - $\text{dist}(p, u)$ and $\text{dist}(p, v)$ have the same parity
  - Cycle $p - u - v$ has odd length $\text{dist}(p, u) + 1 + \text{dist}(p, v)$
  - Thus G is not bipartite

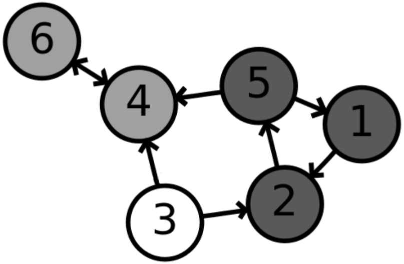
Bipartite check implementation

```
int color[MAXN];

bool dfs(int u)
{
    for (int i = 0; i < (int) neigh[u].size(); i++)
    {
        int v = neigh[u][i];
        if (color[v] == -1) // unassigned
        {
            color[v] = !color[u];
            if (!dfs(v))
                return false;
        }
        else if (color[u] == color[v]) // conflict
            return false;
    }
    return true;
}
```

Table of Contents

Strongly connected components

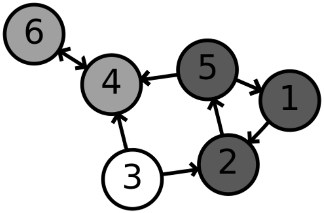
Nodes u, v in same SCC $\Leftrightarrow$ path $u \rightarrow v$ **and** path $v \rightarrow u$



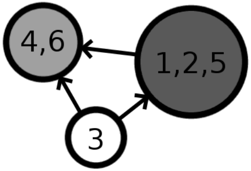
3 strongly connected components

Contracted graph

- If we contract the SCCs into one node, a DAG appears
- This is because all cycles have been contracted
- Useful for DP (as we will see later)



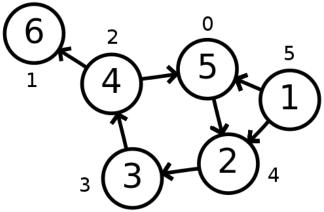
original



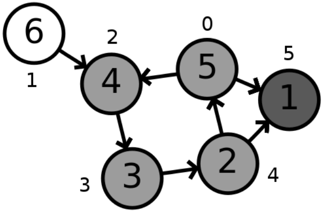
contracted

Kosaraju's algorithm

- Run the DFS toposort algorithm on G
- Use that order and flood with the transpose of G



raw toposort order



flood with decreasing index

Kosaraju implementation

```
void dfs(int u, vector<int> neigh[], vector<int> *st)
{
    // ... (use neigh[] to flood)
    st->push_back(u);
}

vector<int> toposort; // not valid toposort!
for (int i = 0; i < n; i++)
    dfs(i, neigh, &toposort);
visited.reset();
for (int i = n-1; i >= 0; i--)
{
    if (!visited[toposort[i]])
    {
        vector<int> scc;
        dfs(toposort[i], neighT, &scc);
    }
}
```

Source of figures

- https://en.wikipedia.org/wiki/File:6n-graf.svg
- http://en.wikipedia.org/wiki/File:Depth-first-tree.svg
- http://en.wikipedia.org/wiki/File:Data_stack.svg
- http://en.wikipedia.org/wiki/File:Breadth-first-tree.svg
- http://en.wikipedia.org/wiki/File:Data_Queue.svg
- https://commons.wikimedia.org/wiki/File:Simple-bipartite-graph.svg