Decomposition of Graphs: Graph Basics

Graph

An (undirected) graph is a collection V of vertices, and a collection E of edges of each which connects a pair of vertices

Loop

Loop is an edge which connects a vertex to itself

There can be multiple edges between same set of vertices

Decomposition of Graphs: Representing Graphs

Edge list: A list of all edges

Adjacency matrix: A matrix with entries 1 if there is an edge, 0 if not

Adjacency list: A list of adjacent vertices

Op.	Is Edge?	List Edge	List Nbrs
Adj. Matrix	$\Theta(1)$	$\Theta(\|V\|^2)$	$\Theta(\|V\|)$
Edge List	$\Theta(\|E\|)$	$\Theta(\|E\|)$	$\Theta(\|E\|)$
Adj. List	$\Theta(\deg)$	$\Theta(\|E\|)$	$\Theta(\deg)$

In dense graphs, $|E| \approx |V|^2$ because there are many edges between vertices

In sparse graphs, $|E| \approx |V|$ because there are few edges between vertices

Decomposition of Graphs: Exploring Graphs

Path

A path in a graph G is a sequence of vertices $v_0, v_1, \ldots, v_n$ so that for all i, (v_i, v_{i+1}) is an edge of G

Reachability

Input: Graph G and vertex s

Output: The collection of vertices v of G so that there is a path from s to v

```
Component(s):
DiscoveredNodes <- {s}
while there is an edge e leaving DiscoveredNodes that has not been
explored:
    add vertex at other end of e to DiscoveredNodes
return DiscoveredNodes
```

```
Explore(v):
visited(v) <- true
for (v, w) in E:
    if not visited(w):
        Explore(w)
```

Remark: It is a depth first search and it requires adjacency list representation.

Theorem

If all vertices start unvisited, `Explore(v)` marks as visited exactly the vertices reachable from v.

Proof:

- `Explore(v)` only explores things reachable from v
- w is not marked as visited unless explored
- If w is explored, then all neighbors will also be explored
- Suppose u is reachable from v by path
- Assume w furthest along path explored
- It must explore the next item in the path

```
DFS(G):
for all v in V:
    mark v as unvisited
for v in V:
    if not visited(v):
        Explore(v)
```

Running time:

- Each explored vertex is marked visited. No vertex is explored after visited once. Each vertex is explored exactly once, $O(|V|)$
- For each vertex, its neighborhood are checked. Total number of neighborhood over all vertices is $O(|E|)$
- Total $O(|V| + |E|)$

Theorem

The vertices of a graph G can be partitioned into Connected Components so that v is reachable from w if and only if they are in the same connected component.

Proof:

Need to show reachability is an equivalence relation. Namely:

- v is reachable from v
- If v is reachable from w, w is reachable from v
- If v is reachable from u, and w is reachable from v, w is reachable from u

Connected Components

Input: Graph G

Output: The connected components of G

```
Explore(v):
visited(v) <- true
CCnum(v) <- cc
for (v, w) in E:
    if not visited(w):
        Explore(w)
```

```
DFS(G):
for all v in V:
    mark v as unvisited
cc <- 1
for v in V:
    if not visited(v):
        Explore(v)
        cc <- cc + 1
```

Decomposition of Graphs: Previsit and Postvisit Orders

```
Explore(v):
visited(v) <- true
previsit(v)
for (v, w) in E:
    if not visited(w):
        Explore(w)
postvisit(v)
```

```
previsit(v): # Initialize clock to 1
pre(v) <- clock
clock <- clock + 1
```

```
postvisit(v):
post(v) <- clock
clock <- clock + 1
```

Lemma

For any vertices u and v the intervals $[\text{pre}(u), \text{post}(u)]$ and $[\text{pre}(v), \text{post}(v)]$ are either nested or disjoint

Proof:

Assume that u is visited before v. Two cases:

Find v while exploring u . It is a nested case because $\text{pre}(u) < \text{pre}(v)$ and $\text{post}(u) > \text{post}(v)$.

Find v after exploring u. It is a disjoint case because $\text{pre}(v) > \text{post}(u)$

Decomposition of Graphs: Directed Acyclic Graphs

Directed Graphs

A directed graph is a graph where each edge has a start vertex and an end vertex

Directed `DFS`

Only follow directed edges

Cycle

A cycle in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_n$ so that $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n), (v_n, v_1)$ are all edges

Theorem

If G contains a cycle, it cannot be linearly ordered.

Proof:

- Suppose G has cycle $v_1, \ldots, v_n$ and it can be linearly ordered
- Assume v_k comes first
- Then, v_k comes before v_{k-1} which leads to a contradiction

DAGs

A directed graph G is a directed acyclic graph or DAG if it has no cycle

Theorem

Any DAG can be linearly ordered

Decomposition of Graphs: Topological Sort

Source, sink

A source is a vertex with no incoming edges

A sink is a vertex with no outgoing edges

```
LinearOrder(G):
while G is non-empty:
    Follow a path until cannot extend
    Find sink v
    Put v at the end of order
    Remove v from G
```

Running time:

- The worst case is $O(|V|)$ paths (Start from source, remove the sink and start from the source again)
- Each path takes $O(|V|)$ time (For each path, we traverse $O(|V|)$ nodes)
- In total $O(|V|^2)$

```
TopologicalSort(G):
DFS(G)
sort vertices by reverse post-order
```

Theorem

If G is a DAG, with an edge u to v, $\text{post}(u) > \text{post}(v)$

Proof:

Explore v before exploring u. We can only start to explore u after v is explored. $\text{post}(u) > \text{post}(v)$

Explore v while exploring u. We must finish exploring v before u is explored. So, $\text{post}(u) > \text{post}(v)$

Explore v after exploring u. It's impossible because there is an edge pointing from u to v

Decomposition of Graphs: Strongly Connected Components

Connectedness

Two vertices v, w in a directed graph are connected if you can reach v from w and can reach w from v.

Theorem

A directed graph can be partitioned into strongly connected components where two vertices are connected if and only if they are in the same component.

Theorem

The metagraph of a graph G is always a DAG.

Proof:

Supposed not. There must be a cycle $\mathcal{C}$ in G. Any nodes in cycle can reach any others. Then, these nodes should be considered as in the same SCCs. It leads to a contradiction that it is a metagraph (every node of a metagraph represents a strongly connected component).

Strongly Connected Components

Input: Graph G

Output: The strongly connected components of G

```
EasySSC(G): # O(|V|^2 + |V||E|)
for each vertex v:
    run explore(v) to determine vertices reachable from v
for each vertex v:
    find the u reachable from v that can also reach v
these are the SCCs
```

Theorem

If $\mathcal{C}$ and $\mathcal{C}'$ are two strongly connected components with an edge from some vertex of $\mathcal{C}$ to some vertex of $\mathcal{C}'$, then the largest post in $\mathcal{C}$ is bigger than the largest post in $\mathcal{C}'$.

Proof:

Cases:

- Visit $\mathcal{C}$ before visit $\mathcal{C}'$
- Visit $\mathcal{C}'$ before visit $\mathcal{C}$

Case I - visit $\mathcal{C}$ first

- Explore all nodes in $\mathcal{C}'$ while exploring $\mathcal{C}$
- $\mathcal{C}$ has the largest post (recall how the process of depth first search works)

Case II - visit $\mathcal{C}'$ first

- We cannot reach $\mathcal{C}$ from $\mathcal{C}'$. Otherwise, they belongs to the same connected component
- Must finish exploring $\mathcal{C}'$ before exploring $\mathcal{C}$
- So, $\mathcal{C}$ has the largest post

Reverse graph

A reverse graph G^R is the graph obtained from G by reversing all of the edges

Reverse graph components

- G^R and G have the same SCCs
- Source components of G^R are sink components of G
- So, we can find sink components of G by running `DFS` on G^R

```
SCCs(G):
run DFS(G^R)
let v have the largest post number
run Explore(v)
vertices found are first SCC
Remove from G and repeat
```

```
SCCs(G): # Runtime: O(|V| + |E|)
run DFS(G^R)
for v in V in reverse postorder:
    if not visited(v):
        Explore(v)
        mark visited vertices as new SCC
```