Basic Graph Algorithms

Graph: $G = (V, E)$, $E \subseteq V \times V$

Notation: $|V| = n$, $|E| = m$

Size of input: $O(n + m)$

Graph representations:

Adjacency matrix:

$A \in \{0, 1\}^{n \times n}$, $A_{ij} = 1$ if $(i, j) \in E$, $A_{ij} = 0$ if $\{(i, j)\} \notin E$

$\begin{pmatrix}
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0
\end{pmatrix}$

Adjacency list:

$AC(u)$ is linked list of vertices adjacent to $u$
Adjacency Matrix:

**Pros:**
- Can check if two nodes are adjacent in \(O(1)\) time.

**Cons:**
- Space \(\Theta(n^2)\).
- \(\Theta(n)\) to iterate through neighbors.

Adjacency List:

**Pros:**
- Iterate through neighbors at any given time.
- Space \(O(|V| + |E|)\).

**Cons:**
- Checking if \((v, u)\) exists takes \(\Theta(|V|)\) time.

---

**Breadth-First Search (BFS):**

Explore by “levels” from starting node.
Pseudocode in book, but basically:

```plaintext
BFS(G, s) {
    Set mark(v) = False ∀ v ∈ V \ {s}
    Enqueue(s)
    while (queue not empty) {
        v = Dequeue
        For all neighbors u of v {
            if mark(u) = False {
                mark(u) = True
                Enqueue(u)
            }
        }
    }
}
```

**Running time: O(n + m)**

Examine each edge ≤ 2 times (once for each endpoint dequeued)
**Def:** Distance $d(u, v)$ from $u$ to $v$ is min # edges in any path from $u$ to $v$.

**Thm (informal):** BFS$(s)$ gives shortest paths from $s$ to all other nodes.

**PF (sketch):** Contradiction. Assume false.

- BFS path
- Closest node to $s$ where $\not\exists$ BFS$(s)$ doesn't give shortest path.
- Shortest path
Depth-First Search (DFS)

Init: mark(v) = False \( \forall v \in V \)

DFS(v) {
    mark(v) = True
    For each \((u,v)\) in \(A(u)\) {
        if mark(u) = False then DFS(u)
    }
}

Running Time: \(O(n+m)\)

Ex:

Def: \(u\) is reachable from \(v\) if there is a path \(v = v_0, v_1, v_2, \ldots, v_k = u\) s.t. \((v_i, v_{i+1}) \in E\) \(\forall i \in \{0,1,\ldots, k-1\}\)
Then: When DFS(u) terminates, it has visited (marked) all nodes reachable from v

Pro: Terminates; 

spy u reachable from v, but not marked

\[ \rightarrow a \rightarrow x \rightarrow y \rightarrow a \rightarrow \rightarrow u \]

Contradiction: would have marked y.

DFS(u) {
  set mark(v) = False \forall v \in V
  while \text{ unmarked node } v, \text{ DFS}(v)
}

"Timestamps": keep track of start/finish times.
Replaces mark.
DFS (v) {
    t = 0
    set start(v) = t and (w) = 0 ∀u ∈ V
    while ∃ node v with start(v) = 0, DFS(v)
}

DFS(v) {
    t = t + 1
    start(v) = t
    for each (v, u) ∈ A(v) {
        if start(u) = 0 then DFS(u)
    }
    t = t + 1
    finish (v) = t
}

Edge types:
- (v, u) e.d.
- Tree edge: DFS(v) calls DFS(u)

Diagram:
[Diagram with nodes and edges labeled]
**Forward edge**: \((uv)\) s.t. \(u\) descendent of \(v\)

\[
\text{start}(u) < \text{start}(v) \land \text{finish}(v) < \text{finish}(u)
\]

**Back edge**: \((uv)\) s.t. \(u\) ancestor of \(v\)

\[
\text{start}(u) < \text{start}(v) \land \text{finish}(v) < \text{finish}(u)
\]

**Cross edge**: \((uv)\) s.t. \(u\) neither ancestor nor descendent of \(v\)

\[
\text{start}(u) < \text{finish}(v) \land \text{start}(v) < \text{finish}(u)
\]

**Topological sort**:

**Def**: Directed graph \(G\) is a Directed Acyclic Graph (DAG) if no directed cycles

**Def**: A topological sort \(v_1, v_2, \ldots, v_n\) of a DAG is an ordering of vertices s.t. all edges of the form \((v_i, v_j), i < j\)

**Thm**: \(G\) is a DAG iff DFS(G) has no back edges
**Proof:** (⇒) contrapositive. If DFS(h) has a back edge, then G is not a DAG.

(⇐) contrapositive. Say G is not a DAG, let C be a directed cycle.

Let u ∈ C be node of min start time.

⇒ all nodes in C descendents of u
⇒ (u, w) a back edge

Use this to compute topological sort:

Do DFS(G).

When node v finished (DFS(v) returns), put at head of list.

since no back edges, if (v, w) ∈ E, finish (w) < finish (v)
⇒ w already in list
Running Time: O(|V|n)

Strongly Connected Components (SCCs):
Kosaraju's Algorithm!

Def: C ∈ V is a strongly connected component if maximal subset s.t.

∀ u, v ∈ C, u can reach v and vice versa.

Fact: Given directed graph G, there is a unique partition of V into SCCs.
Ref: Reachability is an equivalence relation.
Problem: Given \( G \), compute \( \text{SCCs} \)

Trivial Alg: Do DFS/BFS from each node, keep track of what’s reachable from where. \( O(n+m) \)

Def: Let \( \hat{G} \) be graph of \( \text{SCCs} \):
- vertex \( u(c) \) for each \( \text{SCC} \) \( C \)
- edge \( (u(c), u(c')) \) if \( \exists u(c), u(c') \) \( (u, v) \in E \)

Thm: \( \hat{G} \) is a \( \text{DAG} \)

Pr.:
Since $G$ is a DAG, has a topological sort

Let $C$ be a sink SCC (no outgoing edges).
Let $u$ be any vertex in $C$.

**Def:** $\text{Finish}(C) = \max_{u \in C} \text{Finish}(u)$

**Lemma:** Run DFS($G$). Let $C_1$, $C_2$ distinct SCCs s.t. $(u(C_1), u(C_2)) \in E(G)$. Then $\text{Finish}(C_1) > \text{Finish}(C_2)$
Let $x \in C_1 \cup C_2$ be first node encountered by DFS.

If $x \in C_2$:

IF $x \in C_1$:

So node with largest finishing time in a source SCC (no incoming edges)

Want node in sink SCC.
Flip the graph!
\( G^T = G \) with all directions flipped

Kosaraju's Alg:

DFS\((G^T)\) to get finishing times

while \( G \) non-empty 

Let \( v \) be vertex in \( G \) with largest finish time

Run DFS\((v)\), let \( C \) be all nodes found

Delete \( C \) from \( G \)

Running Time: \( O(n + m) \)
Correctness:

Let $C_1, C_2, \ldots, C_k$ be sets identified by alg $\text{in order}$

Claim: $C_i$ is sink SCC of $G \setminus \bigcup_{j=1}^{i-1} C_j$

PF: Induction on $i$

Base Case:

Inductive Step:

In $G \setminus \bigcup_{j=1}^{i-1} C_j$, let $v$ be vertex with largest finish time

\[ i \longrightarrow \]