Exploring a graph

Visit all nodes or all nodes reachable from some “source”. Further: find shortest paths, connected components, . . .

Breadth-First/Depth-First Search

Breadth First Search — a cautious search where we check all vertices one edge away from the start node, then two edges away, etc.
**Depth First Search** — a bold search where we go as far as possible until there’s nothing new to explore, and then back up to find something new.

Adjacency lists
- 1: 2, 3, 4
- 2: 5, 7, 4, 3, 1
- 3: 1, 4, 2
- 4: 3, 1, 2
- 5: 2, 6, 7
- 6: 5
- 7: 5, 2

Order in which vertices are discovered: 1, 2, 5, 6, 7, 4, 3

Order in which vertices are finished: 6, 7, 5, 3, 4, 2, 1

Note: these orders depend on the ordering of the adjacency lists.

Use a stack to store vertices that have been discovered but not yet explored.
Depth First Search as a recursive routine (the stack is implicit)

\[
\text{DFS}(v) \\
\text{mark}(v) := \text{discovered} \\
\text{for each vertex } u \text{ in } \text{Adjacency}(v) \\
\quad \text{if } u \text{ is undiscovered then} \\
\quad\quad \text{parent}(u) \leftarrow v; \quad (v, u) \text{ is a tree edge.} \\
\quad \text{DFS}(u) \\
\quad \text{else if } u \neq \text{parent}(v) \text{ then} \quad (v, u) \text{ is a non-tree edge.} \\
\quad \text{mark}(v) \leftarrow \text{finished}
\]

\[
\text{DFS} \quad \# \text{ to handle disconnected graphs} \\
\text{initialize: mark every vertex undiscovered} \\
\text{for each vertex } v \\
\quad \text{if } v \text{ is undiscovered then} \quad \# \text{ start a new tree rooted at } v \\
\quad \text{DFS}(v)
\]

As with BFS, we should store more information as we do the search. Store parent pointers, distinguish \textbf{tree} edges and \textbf{non-tree} edges, store component numbers.
Depth First Search as a recursive routine (the stack is implicit)

\[
\text{DFS}(v) \\
\begin{align*}
\text{mark}(v) &:= \text{discovered} \\
\text{for each vertex } u \text{ in } \text{Adjacency}(v) &\quad \text{if } u \text{ is undiscovered then} \\
&\quad \quad \text{parent}(u) := v; \ (v,u) \text{ is a tree edge;} \\
&\quad \quad \text{DFS}(u) \\
&\quad \text{else if } u \neq \text{parent}(v) \text{ then } (v,u) \text{ is a non-tree edge} \\
\text{mark}(v) &:= \text{finished}
\end{align*}
\]

\[
\text{DFS} \quad \text{# to handle disconnected graphs} \\
\begin{align*}
\text{initialize: mark every vertex undiscovered} \\
\text{for each vertex } v &\quad \text{if } v \text{ is undiscovered then} \quad \text{# start a new tree rooted at } v \\
&\quad \quad \text{DFS}(v)
\end{align*}
\]

\[
\#\text{vertices} \quad \#\text{edges}
\]

Runtime: $O(n + m)$ — same analysis as for BFS

Exercise: Enhance this to number the connected components and record the component number of each vertex.
Properties of DFS

DFS gives rich structure:
- partition into one tree for each connected component
- edge classification (tree and non-tree edges)
- vertex orderings: order of discovery, order of “finishing” (more on this later)

Lemma. DFS(v₀) reaches all vertices connected to v₀.
Proof. Suppose there is a path v₀, v₁, . . . , vᵢ .
Look at the last vertex of this path that is discovered, say vᵢ .
Then we explore (and discover) all neighbours of vᵢ including vᵢ₊₁ .
(This can be made more formal by induction.)

Lemma. Any non-tree edge joins an ancestor and a descendant.

Definition. v is an ancestor of u (and u is a descendant of v) if we can follow parent pointers from u to reach v

Diagram:
Lemma. Any non-tree edge joins an ancestor and a descendant.

Proof. Consider a non-tree edge \((a, b)\).
Suppose vertex \(a\) discovered first.
In DFS \((a)\) we examine neighbour \(b\).
We will discover \(b\) before \(a\) is finished.
So \(b\) appears in sub-tree rooted at \(a\).
So \(a\) is ancestor of \(b\).
Enhancing DFS to compute discover and finish times

d(v) = discover time    f(v) = finish time

initialize time := 1
DFS(v)
    mark(v) := discovered
    d(v) := time; time := time + 1
    for each vertex u in Adjacency(v)
        if u is undiscovered then
            DFS(u)
    f(v) := time; time := time + 1

For multiple connected components, we should initialize time := 1 only at the very beginning.

Discover and finish times form a parenthesis system

If d(v) < d(u) then

\[
\begin{bmatrix}
  d(v) & d(u) & f(u) & f(v)
\end{bmatrix}
\quad \text{OR}\quad
\begin{bmatrix}
  d(v) & f(u) & d(u) & f(u)
\end{bmatrix}
\]

because the interval d(v), f(v) is the time when v is on the stack
Summary of Lecture 13, part 1

- Depth First Search, discover and finish times

What you should know from Lecture 13, part 1:

- how to do DFS
- property of non-tree edges

Next:

- Applications of Depth First Search
Applications of DFS (for undirected graphs)

Finding 2-connected ("biconnected") components

This graph is connected but removing one vertex (2 or 5) disconnects it.

Vertex v is a cut vertex if removing v (and its incident edges) makes the graph disconnected.
Cut vertices are bad in networks.

Biconnected components

These are the “pieces” you get by separating at the cut vertices.

DFS from vertex 5

Adjacency lists

1: 2,3,4
2: 5,7,4,3,1
3: 1,4,2
4: 3,1,2
5: 2,6,7
6: 5
7: 5,2
Finding 2-connected ("biconnected") components

linear time $O(n+m)$ algorithm using DFS
due to Hopcroft and Tarjan 1973

photos from the 70s
Finding 2-connected (“biconnected”) components.

First find cut vertices.
Characterizing cut vertices in terms of the DFS tree

Claim. The root is a cut vertex iff it has >1 child

Lemma. A non-root vertex v is a cut vertex iff
v has a subtree T with no non-tree edge going from the subtree to an ancestor of v.

*removing v separates T from the rest of G.

*since removing v disconnects G some subtree must disconnected
  • no non-tree edge from subtree to ancestor of v
Finding cut vertices.

Making the lemma into a fast algorithm to find cut vertices

Define \( \text{low}(u) = \min \{d(w): \text{there is an edge } (u,w) \text{ or } u \text{ has a descendant } x \text{ and there is an edge } (x,w) \} \)

interpretation: \( \text{low}(u) \) = how high in the tree can we get to from \( u \) by going down (0 or more) and then up 1 edge.

The lemma rephrased: a non-root vertex \( v \) is a cut vertex iff \( v \) has a child \( u \) with \( \text{low}(u) \geq d(v) \)

Example

d = 1 2 3 4 5 6 7
\( d(v) = 7 2 6 5 1 12 3 \)
\( \text{low}(v) = 2 1 2 2 1 12 1 \)

Test vertex 2 — child 4 has \( \text{low}(4) = 2 \geq d(2) \)
\( d(2) = 2 \) — vertex 2 is a cut vertex

Test vertex 3, \( d(3) = 6 \). Child 1, \( \text{low}(1) = 2 < d(3) \) — so 3 is not cut vertex
Finding cut vertices.
Define \( \text{low}(u) = \min \{d(w): \text{there is an edge } (u,w) \text{ or } u \text{ has a descendant } x \text{ and there is an edge } (x,w) \} \)

interpretation: \( \text{low}(u) = \) how high in the tree can we get to from \( u \) by going down (0 or more) and then up 1 edge.

Computing low recursively:
\[
\text{low}(u) = \min \left\{ \begin{array}{l}
\min \{d(w) : (u,w) \in E \} \\
\min \{\text{low}(x): x \text{ a child of } u \}
\end{array} \right. 
\]

Order of computation: do children of \( u \) before \( u \). So order by finish time.
i.e., When we finish a vertex \( u \) in DFS, use the recursive formula to compute \( \text{low}(u) \)

Advanced Exercise. Enhance DFS to compute \( \text{low} \). \( O(n+m) \)

Finding cut vertices:
After computing low, run this test:
If root has >1 child then root is a cut vertex
For every vertex \( v \neq \text{root} \)
if \( v \) has a child \( u \) with \( \text{low}(u) \geq d(v) \) then \( v \) is a cut vertex

runtime: \( O(n+m) \)
Finding 2-connected ("biconnected") components.

So far, we found the cut vertices.

Advanced Exercise. Find the 2-connected components. O(n+m)
Summary of Lecture 13

- Depth First Search, discover and finish times
- DFS to find cut vertices

What you should know from Lecture 13:

- how to do DFS
- property of non-tree edges
- what are cut vertices, 2-connected components
- appreciate the application of DFS to finding cut vertices

Next:

- Depth First Search for directed graphs