DFS and Topological Sort

Sam McCauley

February 20, 2025

Welcome Back!

- Problem set 0 back; Problem set 2 out later today; groups already posted
- Graded homework back on table
- Tapia info session today at 4pm in CS commmon room
- CoSSNACs tonight at 7pm! :) In CS common room
- Any questions before we start?

BFS Properties (Review/reference)

Useful shorthand: if $x \in L_i$, we also write $i = L[x]$.

Lemma

For any undirected graph G, if $(x, y) \in E$, then for any BFS tree on G, $|L[x] - L[y]| \leq 1$.

Theorem

In a connected undirected graph G, BFS starting at any vertex s will visit every vertex.

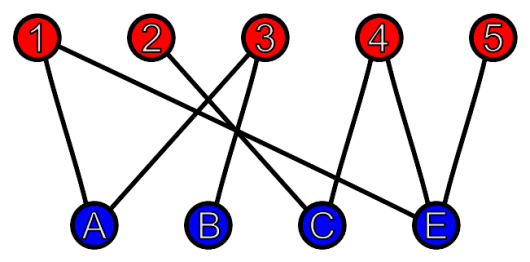
Theorem

In any graph G, for any vertex v explored using BFS, $L[v]$ is the distance from s to v.

Theorem

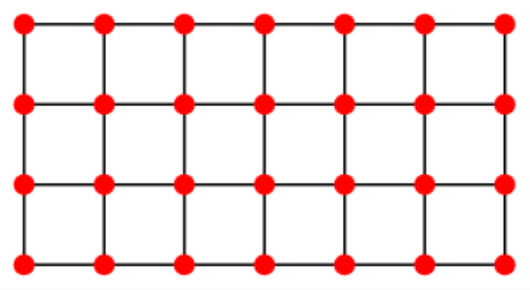
BFS runs in $O(n + m)$ time on any graph with n vertices and m edges.

Last BFS Application: Bipartite Testing



- Bipartite graph: graph G whose vertices can be partitioned into V_1, V_2 where every edge e has one endpoint in V_1 and one endpoint in V_2.

Last BFS Application: Bipartite Testing



- How can we test if a given undirected graph is bipartite?
  - Maybe greedily assign vertices to one set or the other? Does this always work?
  - Today: use BFS
  - Run BFS from any start vertex. If there is an edge between two vertices at the same level, return “not bipartite.” Otherwise, return “bipartite.”

Bipartite Testing

Theorem

The BFS bipartite testing algorithm is correct.

Proof (part 1: correct if returns "bipartite"): If the algorithm returns "bipartite," then G is bipartite.

Let V_1 be all vertices at even levels, and V_2 be all vertices at odd levels. We must show that every edge is between a vertex in V_1 and a vertex in V_2.

Consider an edge $e = (u, v)$. We must have that $|L[u] - L[v]| \leq 1$ by BFS property. We cannot have $L[u] = L[v]$, so $|L[u] - L[v]| = 1$. But then $u \in V_1$ and $v \in V_2$ (or vice versa). $\square$

Bipartite Testing

Theorem

The BFS bipartite testing algorithm is correct.

Proof (part 2: correct if returns "not bipartite"): If the algorithm returns "not bipartite," there is an edge e between two vertices v_1 and v_2 at the same level k (for some k). Assume by contradiction that G is bipartite. Then v_1 and v_2 are in different partitions.

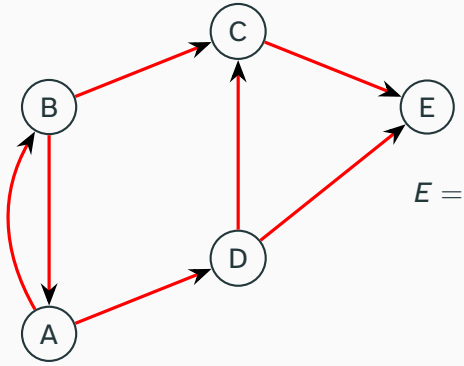
Consider the case where k is even. There is a path from s to v_1 of length k, so s and v_1 are in the same partition. Similarly, s and v_2 are in the same partition; contradicting that v_1 and v_2 are in different partitions.

The case where k is odd is almost exactly the same.

BFS is a simple algorithm, but—with careful analysis—it can accomplish quite a lot!

Directed Graphs

Directed Graphs

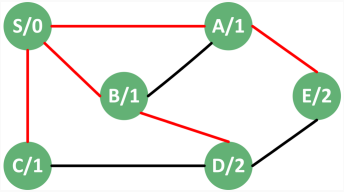


$$V = \{A, B, C, D, E\}$$

$$E = \{(A,B), (B,A), (A,D), (B,C), (D,C), (C,E), (D,E)\}$$

- In a directed graph, edges have an ordering: an edge (u, v) is *from* u *to* v.
- Called *directed edges* (some call them arcs; I won't however)
- Good for capturing some kinds of data (website links, etc.)
- Notion of a path, etc., is the same

BFS Properties Summary (Directed Graphs)



- Starts at some node s
- Partitions vertices into levels $L_0, L_1, \ldots$
- Gives a BFS tree T; a vertex at height h in the tree is in L_h
- ~~If $(x, y) \in E$, the level of x and y differ by ≤ 1~~ [This is only true for undirected graphs!]
  - **In pairs:** can you give a directed graph where $(x, y) \in E$, but the level of x and y differ by more than 1?
- A vertex is at height h in T if and only if its shortest path from s has distance h

Storing a Graph

How to store a graph?

Goal: Use data structures we know to store a graph to allow things like traversals

- *Adjacency List* representation
- *Adjacency Matrix* representation

Adjacency List

- For each vertex, store all neighbor edges/vertices in a linked list
- Works well for:
  - Can find all d_v neighbors of v in $O(1 + d_v)$ time
  - Only requires $O(n + m)$ space (why?)
- Does not work well for:
  - Given an edge $e = (u, v)$, is $e \in E$?
  - Must scan through neighbors of u; requires $\Omega(d_u)$ time.

Example [On Board #1]

Adjacency Matrix

- Store an $n \times n$ matrix
- Store a 1 in entry (i, j) if there is an edge from the ith to the jth vertex
- Works well for:
  - Given an edge $e = (u, v)$, is $e \in E$ in $O(1)$ time.
- Does not work well for:
  - Space if graph has few edges (requires $\Omega(n^2)$ space)
  - Finding all d_v neighbors of v takes $\Omega(n)$ time
- Used much less often

Example [On Board #2]

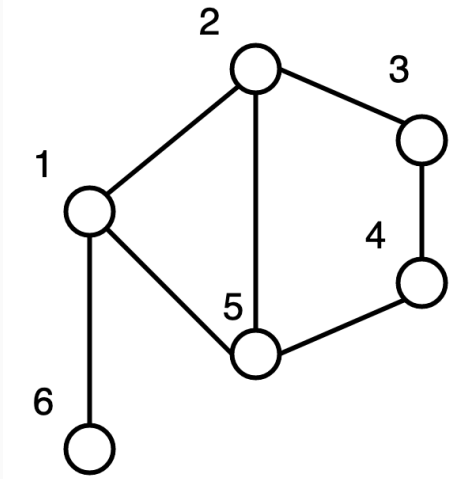
Depth-First Search

DFS: Stack Definition

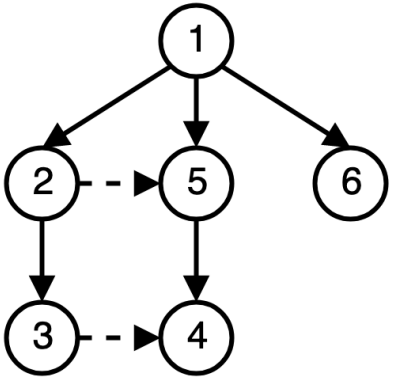
```
1  DFS(G, s):
2      Put s in a stack S
3      while S is not empty:
4          v = S.pop() # take the top vertex from S
5          if v is not marked as visited:
6              mark v as visited
7              for each edge (v,w):
8                  S.push(w) # add w to the top of S
```

- We can obtain DFS by using a stack rather than a queue in BFS.
- Define a DFS tree: the parent edge of a node is the edge that marked it visited.

BFS Tree Example

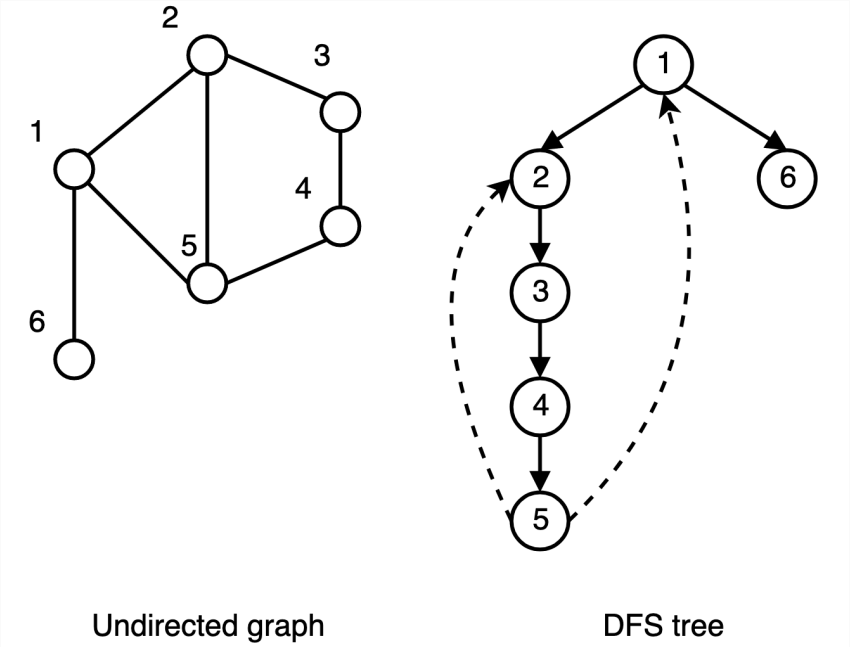


Undirected graph



BFS tree

DFS Tree Example



DFS Running Time

Theorem

DFS runs in $O(n + m)$ time.

Proof: Each time we mark a node v as visited, we spend $O(1 + d_v)$ time, where d_v is the number of neighbors of v. (This includes the time to mark all neighbors, push them onto the stack, pop them off the stack, and check if they are visited.)

Total running time (recall that $\sum_{v \in V} d_v = 2m$):

$$\sum_{v \in |V|} O(1 + d_v) = O(n) + \sum_{v \in |V|} O(d_v) = O(n + m).$$

DFS: Recursive Definition

```
DFS(s):
    mark s as visited
    for each neighbor v of s:
        if v is not visited:
            DFS(v)
```

- (Won't prove): this is the same as the stack version
- Side Discussion: Which would you rather implement? What are the upsides and downsides of each?
- Observation: Every node marked as visited in a recursive call of `DFS(x)` is a descendant of x in the DFS tree.

DFS Correctness

Theorem

DFS starting at vertex s will visit vertex v if and only if v is reachable from s.

Proof: ($\Rightarrow$) since DFS visits vertex v, v must be in the DFS tree T. Then there is a path from s to v: the path in T.

DFS Correctness

Theorem

DFS starting at vertex s will visit vertex v if and only if v is reachable from s.

Proof: ($\Leftarrow$) if v is reachable from s, there is a path $p = (s = v_0, v_1, \ldots, v_k = v)$. We show that all vertices in p are marked by induction.

I.H.: For some i, vertex v_i in p is marked at some point during DFS on s.

Base case v_0: s is marked on the first DFS iteration.

Inductive step: if v_i is marked, then $\texttt{DFS}(v_i)$ was called. During the for loop, either v_{i+1} was already marked visited (and we're done), or $\texttt{DFS}(v_{i+1})$ is called, during which v_{i+1} is marked visited.

DFS Intuition

- We showed: DFS(v) explores all nodes reachable from *v*

- So if DFS is called *recursively* on a node *x*, it will visit all nodes reachable from *x* that have not yet been marked visited. [On Board #3]

DFS Property

Theorem

For any edge $e = (u, v)$ in an undirected graph, either u is an ancestor of v or v is an ancestor of u in any DFS tree T.

Proof: Obvious if $e \in T$.

Otherwise, say $e \notin T$. First, suppose that `DFS(u)` is called before `DFS(v)`.

When we consider neighbor v of u, it must be that v was already marked visited (Why)? Answer: otherwise, we would have $e \in T$.

Therefore, v must have been marked visited between when `DFS(u)` was called, and when the `for` loop was called on v. Therefore, v was visited at some point during `DFS(u)`. By our observation, v is a descendant of u in T.

The case where `DFS(v)` is called before `DFS(u)` is identical.

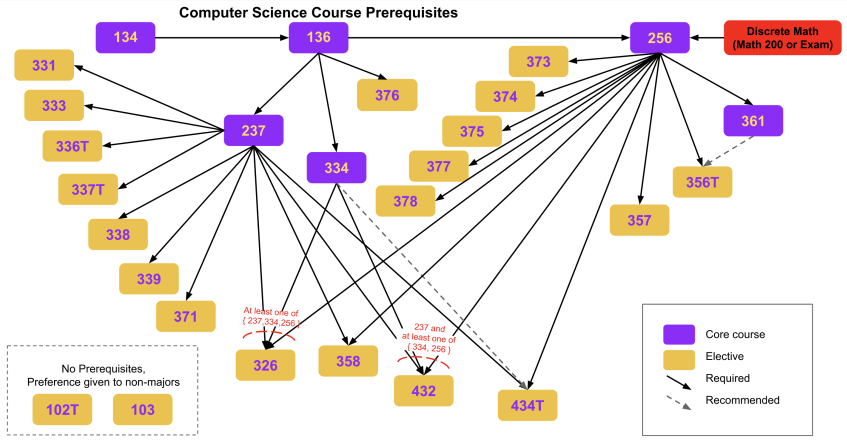
DFS: Recursive Definition

```
1 DFS(s):
2     mark s as visited
3     for each neighbor v of s:
4         if v is not visited:
5             DFS(v)
```

- Observation: Every node marked as visited in a recursive call of `DFS(x)` is a descendant of x in the DFS tree.

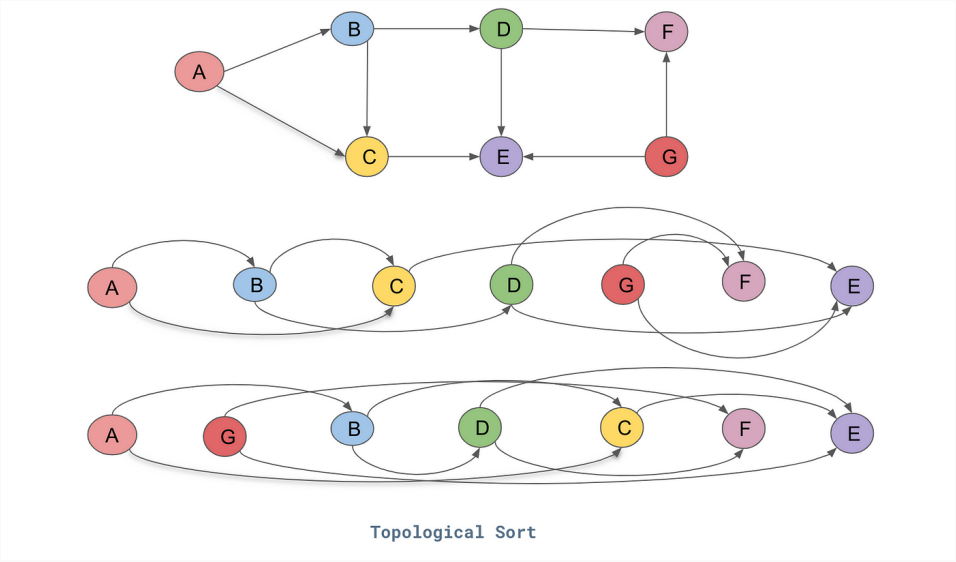
Topological Ordering

Topological Ordering



- Goal: Order the vertices of a graph so that for any edge (u, v), u comes before v in the final order
- *Example:* find a sequence of all courses satisfying prerequisites

Topological Ordering (a.k.a. Topological Sort)



Source: https://medium.com/@konduruharish/topological-sort-in-typescript-and-c-6d5ecc4bad95

DAGs and Toplogical Ordering

Theorem

A graph G has a topological ordering if and only if G is acyclic.

Proof: ($\Rightarrow$) if G has a topological ordering, G cannot have a cycle. (Why?)

DAGs and Toplogical Ordering

Theorem

A graph G has a topological ordering if and only if G is acyclic.

Proof: ($\Rightarrow$) if G has a topological ordering, G cannot have a cycle.

We'll prove ($\Leftarrow$) (if G is acyclic, then G has a topological ordering) in the next few slides.

DAGs and Toplogical Ordering

First, let's prove the following.

Lemma

Every Directed Acyclic Graph has a vertex with indegree 0.

Proof: Assume the contrary: there exists a DAG G where all vertices have indegree > 0.

Pick a vertex v_0. Find some v_1 such that $(v_1, v_0) \in G$. In general, for each v_i, find a v_{i+1} where $(v_{i+1}, v_i) \in G$.

After n steps, we have a sequence $v_0, v_1, v_2, \ldots, v_n$. One vertex must repeat (why?) (Answer: pidgeonhole principle).

Let $v_i = v_j$ for some $j > i$. Then stepping back through the sequence, $v_j, v_{j-1}, v_{j-2}, \ldots, v_i$ is a cycle. Contradiction.

Topological Ordering: Simple Algorithm

```
1 while L has length less than n:
2     find a vertex v with indegree 0 (if not exists, graph has
          a cycle)
3     add v to the end of L
4     remove v and its outgoing edges from G
```

- Can we prove that this algorithm works?

Topological Ordering: Simple Algorithm

```
while L has length less than n:
    find a vertex v with indegree 0 (if not exists, graph has
        a cycle)
    add v to the end of L
    remove v and its outgoing edges from G
```

- Running time?
- Store vertices with indegree 0?
  - Use a stack of vertices with indegree 0, and an array storing indegree of all vertices
  - Initialize array with a BFS/DFS
- Time to remove vertex and edges with adjacency list?
- Overall: $O(n + m)$ time

Finding Topological Ordering with DFS

```
1 DFS-Cycle(s):
2     mark s as active
3     for each neighbor v of s:
4         if v is not active or finished:
5             DFS-Cycle(v)
6         else:
7             report that there is a cycle
8     mark s as finished
9     add s to the front of L
```

- Running time?
- $O(n + m)$
- Why does this work?

Topological Ordering

- Useful for some applications

- Also: can find if a directed graph contains a cycle in $O(n + m)$