CS 341 – Algorithms

Lecture 5 – Breadth First Search

26 May 2021

Today’s Plan

1. Graph Connectivity
2. Breadth First Search
3. BFS Tree and Shortest Path
4. Bipartite Graphs

1. First tutorial on Tuesday

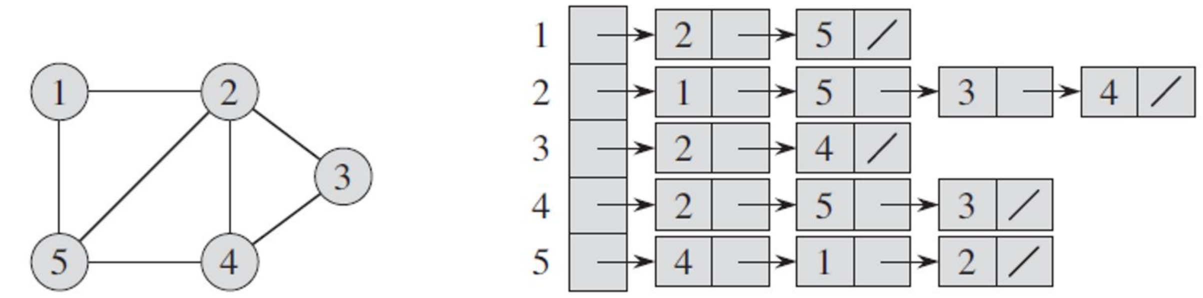
Graph Representation

Many problems in computer science can be modeled as graph problems.

There are two standard representations of an undirected graph $G = (V, E)$ with $n = |V|$ and $m = |E|$.

One is the **adjacency matrix**, and the other is the **adjacency list**.

× $O(1)$ edge



$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

$O(1)$ $\exists$ edge i, j

[CLRS]

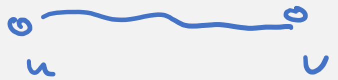
$\Omega(n^2)$ space

$\Rightarrow \Omega(n^2)$ time

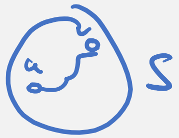
We will mostly use adjacency list, as this allows us to design $O(n + m)$-time algorithm.

Graph Connectivity

We say two vertices u, v are connected if there is a path from u to v.

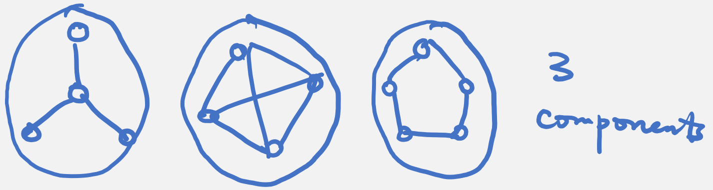


A subset of vertices $S \subseteq V$ is connected if u, v are connected for all $u, v \in S$.



A graph is connected if u, v are connected for all $u, v \in V$.

A connected component is a maximally connected subset of vertices.



Some of the most basic algorithmic questions about a graph are:

1. To determine whether a given graph is connected.
2. To find all the connected components of a given graph.
3. To determine whether two vertices u, v are connected.
4. To find a shortest path between two vertices u, v.

Breadth first search (BFS) can be used to answer all these questions in $O(n + m)$ time.

Today’s Plan

1. Graph Connectivity
2. Breadth First Search
3. BFS Tree and Shortest Path
4. Bipartite Graphs

Breadth First Search

To motivate BFS, imagine that we are searching a person in a social network.

A natural strategy is to ask for friends, then friends of friends, then friends of friends of friends, etc.

Input: $G = (V, E)$, $s \in V$. source vertex

Output: all vertices reachable from s.

Initialization: visited[v] = false for all $v \in V$.

queue Q is empty. enqueue(Q, s). visited[s] = true.

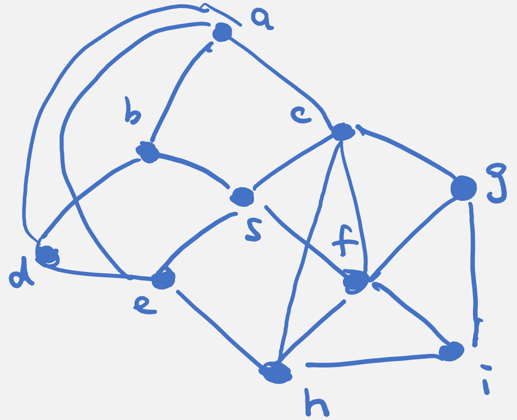
While $Q \neq$ empty do

u = dequeue(Q)

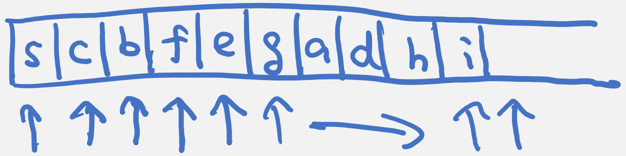
for each neighbor v of u

if visited[v] = false

enqueue(Q, v). visited[v] = true.



Q



Time Complexity

Each vertex is enqueued at most once.

When we dequeue a vertex u, run for loop for $\deg(u)$ steps

total time complexity : $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$. $\square$

MATH 239 : handshaking lemma

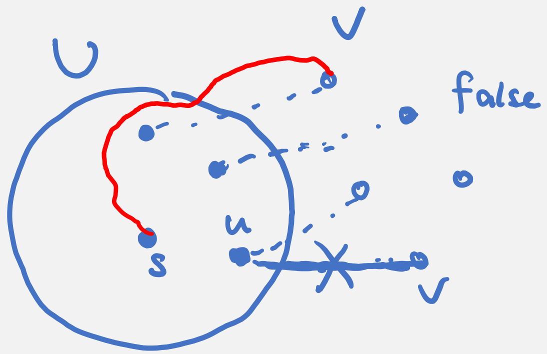
$$\sum_{v \in V} \deg(v) = 2m$$

Correctness

Lemma. There is a path from s to v if and only if $visited[v] = true$ at the end.

$\Rightarrow$ contrapositive: if $visited[v] = false$, then $\nexists$ path from s to v.

Consider U set of all vertices that $visited[v] = true$ $\forall v \in U$



Claim: no edges with one endpoint in U and another endpoint in $V \setminus U$.

proof Suppose uv exists.
In the for loop of u,
we would have marked $visited[v] = true$.
contra. $\square$

so, for any $v \in V \setminus U$, $\nexists$ a path from s to v
otherwise it would violate the claim. $\square$

Correctness

Lemma. There is a path from s to v if and only if $visited[v] = true$ at the end.

Proof $\Leftarrow$ if visited[v]=true, then there is a path from s to v.

induction: base case : visited[s] = true.

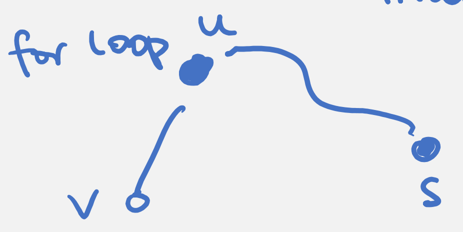
induction step: visited[v] = true. within a for loop for a vertex u

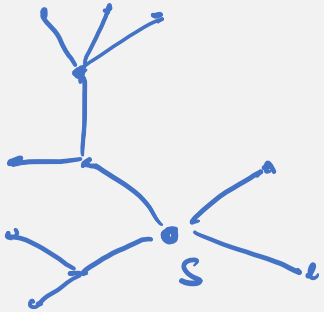
at that time, u was dequeued, was enqueued earlier

visited[u] = true

by I.H. visited[u] = true $\Rightarrow$ $\exists$ a path from s to u

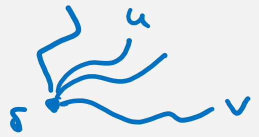
$\Rightarrow$ $\exists$ a path from s to v, by extending by the edge uv.





Graph Connectivity

1. To determine whether a given graph is connected.



pick s. run BFS. graph is connected iff visited[v] = true for $v \in V$.
$O(m+n)$

2. To find all the connected components of a given graph.

Exercise. add an outer for loop. time complexity $O(n+m)$.

3. To determine whether two vertices u, v are connected.

pick u, run BFS. visited[v] = true $\Leftrightarrow$ u,v connected.

4. To find a shortest path between two vertices u, v. $\Leftarrow$ next time.

Today’s Plan

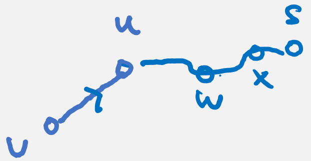
1. Graph Connectivity
2. Breadth First Search
3. BFS Tree and Shortest Path
4. Bipartite Graphs

1. Homework 1 due on Monday
2. Homework 2 to be posted early next week

BFS Tree

How to trace out a path from s to v (if such a path exists)?

proof



parent[v] = u

for loop to trace out the path

BFS tree : all the edges (v, parent[v])

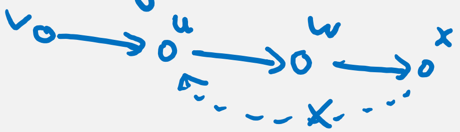
Why the edges (v, parent[v]) form a tree?

(1) every vertex v has one edge (v, parent[v])

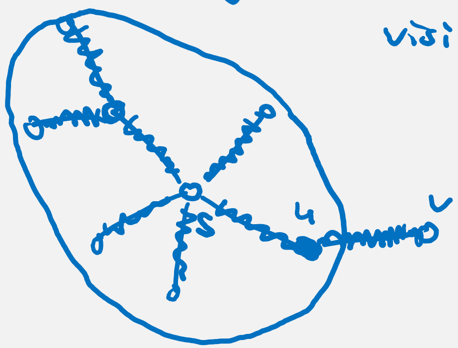
but not s

total n-1 edges (1) + (2) $\Rightarrow$ tree (MATH 239)

no cycles because: parent[v] is visited earlier than v



(2) inductively. tree on visited[v] = true

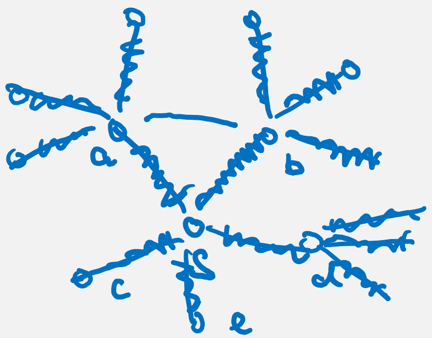


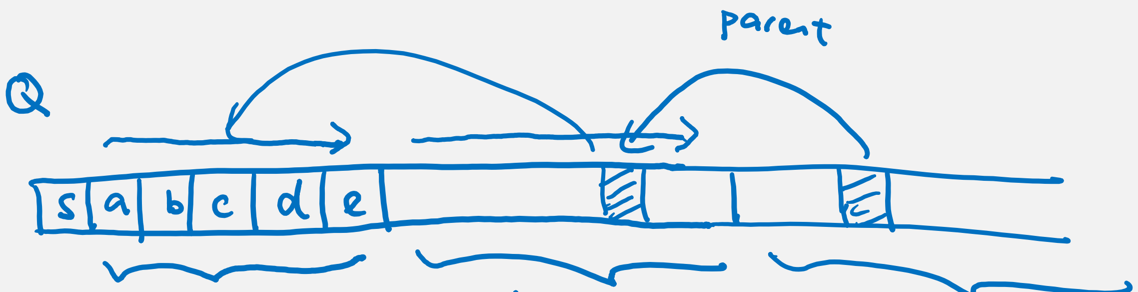
Shortest Path

Proposition. The path from v to s on a ~~DFS~~ BFS tree is a shortest path from v to s.

induction

v is of distance k to s if the shortest paths from v to s have length k.





distance 1 before all other vertices of distance > 1

distance 2 before vertices of distance > 2

distance 3

invariant: Shortest path distances computed correctly for visited[v] and keep a shortest path in the tree so far

$\square$

BFS with Shortest Path Distances

Input: $G = (V, E)$, $s \in V$.

Output: all vertices reachable from s and their shortest path distance from s.

Initialization: visited[v] = false for all $v \in V$.

queue Q is empty. enqueue (Q, s). visited [s] = true. distance [s] = 0.

While Q ≠ empty do

u = dequeue (Q)

for each neighbor v of u

if visited [v] = false

enqueue (Q, v). visited[v] = true. parent[v] = u. distance[v] = distance [u] + 1.

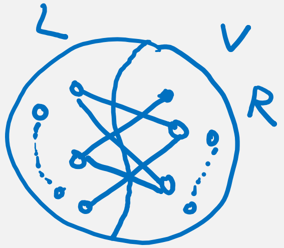
Today’s Plan

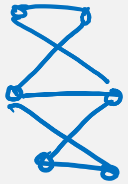
1. Graph Connectivity
2. Breadth First Search
3. BFS Tree and Shortest Path
4. Bipartite Graphs

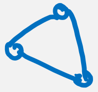
Bipartite Graphs

A graph $G = (V, E)$ is bipartite if there is a bipartition (L, R) of V such that

all edges have one endpoint in L and one endpoint in R.



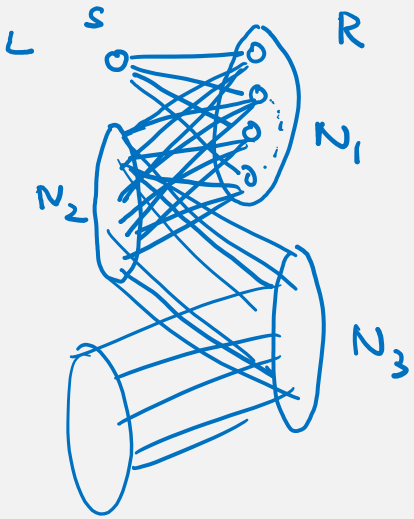




Question: Design an efficient algorithm to check whether a graph is bipartite.

trivial: try all bipartition (L,R). time: $\Omega(2^n)$

idea :



Algorithm

① $L = \{ v \mid dist(s,v) \text{ even} \}$ $R = \{ v \mid dist(s,v) \text{ odd} \}$

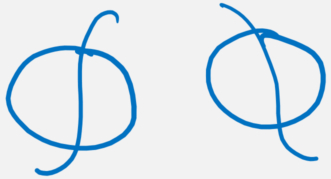
② If there is an edge with L or an edge within R,

then "NON-bipartite"

side[v] = L/R

for each uv

③ Otherwise, output "bipartite".



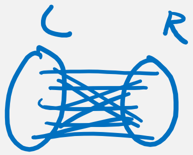
- assume graph is connected

  solve components separately

Time complexity: ① BFS $O(n+m)$ ② $O(m+n)$

Correctness

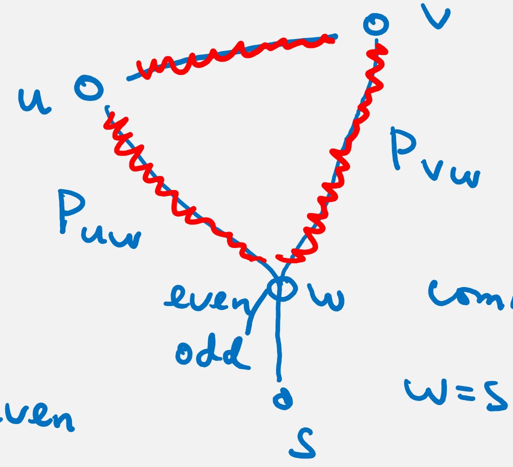
① If the algorithm says "bipartite", correct ✓



② If the " " "non-bipartite", correct?

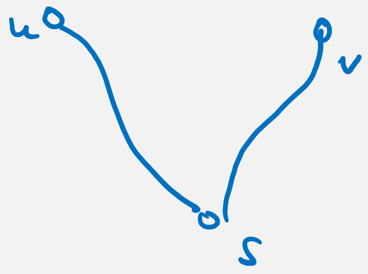
MATH 239: graph bipartite $\Leftrightarrow$ no odd cycles

graph non-bipartite $\Leftrightarrow$ $\exists$ odd cycle.



common ancestor

$w = s$ possibly



regardless of w odd/even

$\text{len}(P_{uw}) + \text{len}(P_{vw}) = \text{even}$

odd cycle $P_{uw} \cup P_{vw} \cup \{uv\}$.

non-bipartite $\Rightarrow$ $\exists$ odd cycle. □

Remarks

1. This provides an algorithmic proof that a graph is bipartite if and only if it has no odd cycles.

2. The BFS algorithm is also a linear time algorithm to find an odd cycle in an undirected graph.

   bipartite : no odd cycles   otherwise. $\exists$ an odd cycle.

3. Having an odd cycle is a "short" proof that a graph is non-bipartite.

   3-coloring

❖ Remember BFS and its main feature is to find shortest paths.