Lecture 4: BFS, Graph Representations, DFS

Sam McCauley

February 14, 2024

Welcome Back!

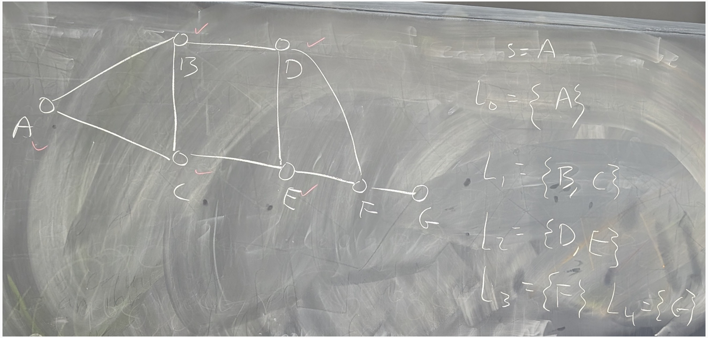
- Trying to get Assignment 0 back to you tomorrow or early Wednesday
- If office hours are over zoom tomorrow I will send an email (unlikely but possible).
- Any questions before we start?

Breadth-First Search

BFS Definition: Intuition

We define BFS using a sequence of layers

- Initialize $L_0 = \{s\}$, $i = 0$; mark s as visited
- if there exists a node in L_i with an unvisited neighbor:
    - Set L_{i+1} to be all unvisited neighbors of nodes in L_i; mark all nodes in L_{i+1} as visited; set $i = i + 1$



BFS Properties

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

Lemma

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

Theorem

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

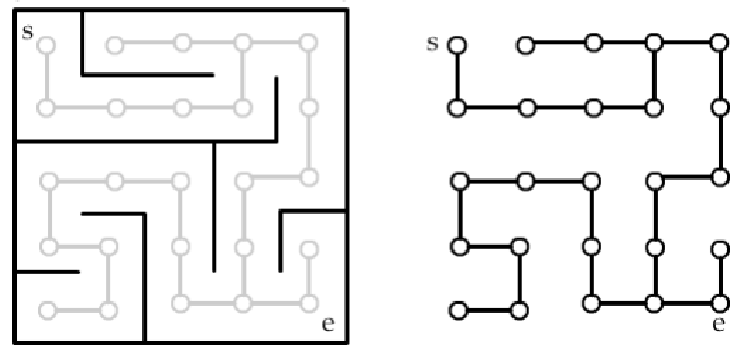
Theorem

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

The BFS Tree

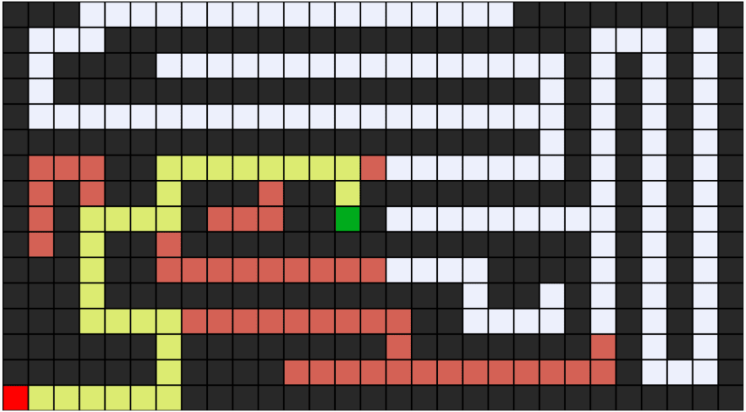
- The levels explored by the BFS are the levels of a tree (i.e. the nodes at a particular height)
- If v' is a neighbor of v that we add to some level, then v is the parent of v'.
- We can calculate the BFS tree while doing the BFS in $O(n+m)$ time

Application: Maze Solving



- BFS can find if a maze is solvable!
- Turn the maze into a graph: node for each square; edge if can get from one square to another
- How can we prove that BFS *always* solves the maze if possible?
- Animation: `https://youtu.be/zMy5MwQWwss?si=VRNW3sgRgMeK7aVd&t=129`

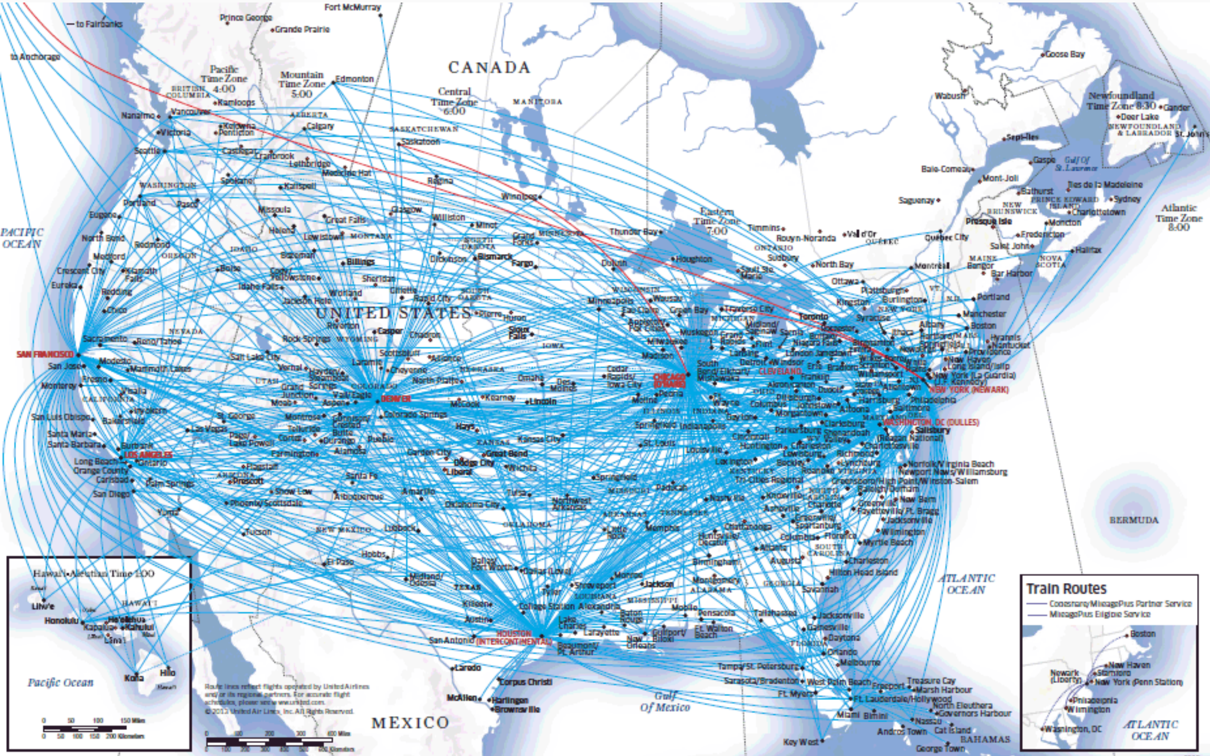
Application: Maze Solving



- How do we get the path from start to end of the maze?
- One answer: use the BFS tree!
- Path from *s* to *e* in the tree is a path from *s* to *e* in the maze

BFS to find Shortest Path

- BFS gives the *shortest path* between the initial vertex s and any other vertex v in the graph
    - We call the length of the shortest path between two vertices u and v the *distance* betwen u and v



Route lines reflect flights operated by United Airlines and/or its regional partners. For accurate flight schedules, please see www.united.com.

BFS to find Shortest Path

- BFS gives the *shortest path* between the initial vertex s and any other vertex *v* in the graph
    - We call the length of the shortest path between two vertices *u* and *v* the *distance* betwen *u* and *v*
- How can we formalize?

Theorem

For any vertex v, if v is at height d of the BFS tree rooted at s, then the shortest path from s to v has length d.

BFS to find Shortest Path

Theorem

For any vertex v, v is at depth d of the BFS tree rooted at s if and only if the shortest path from s to v has length d.

Proof: By strong induction on d. Base case: for $d = 0$, the only vertex with a shortest path of length 0 from s is s; we have that $L_0 = \{s\}$ by definition of BFS.

Now, assume that for some d, for all $0 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d + 1$.)

First, we show that if a vertex v is in L_{d+1}, its shortest path from s has length $d + 1$. We break this into two parts: first we show that there exists a path of length $d + 1$; then we show that no path has length $< d + 1$.

BFS to find Shortest Path

Proof: Now, assume that for some d, for all $1 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d + 1$.)

First, we show that if a vertex v is in L_{d+1}, its shortest path from s has length $d + 1$. We break this into two parts: first we show that there exists a path of length $d + 1$; then we show that no path has length $< d + 1$.

Since $v \in L_{d+1}$, v has a neighbor $v' \in L_d$. By the I.H., the shortest path from s to v' has length d. Therefore, there is a path from s to v of length $d + 1$, so the shortest path from s to v has length *at most* $d + 1$.

Now, we show that no path from s to v has length $< d + 1$. Consider a path of length k, $p = s, v_1, \dots, v_{k-1}, v$ for $k < d + 1$. By the I.H., v_{k-1} is in level L_{k-1}; but since there is an edge from v_{k-1} to v, v must be in L_k or earlier, contradicting our assumption that $v \in L_{d+1}$.

BFS to find Shortest Path

Proof: *Recall:* Proof by strong induction on d.

Now, assume that for some d, for all $1 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d + 1$.)

Now, we show that if the shortest path from s to v has length $d + 1$, then $v \in L_{d+1}$. By I.H., $v \notin L_j$ for $j < d + 1$.

Let $p = s, v_1, \ldots, v_d, v$ be a path of length $d + 1$ from s to v. By the I.H., $v_d \in L_d$. When we explore the neighbors of v_d, we cannot have already explored v since $v \notin L_j$ for $j < d + 1$; thus $v \in L_{d+1}$

BFS to find Shortest Path (wrapup)

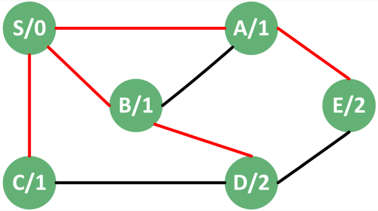
Theorem

For any vertex v, v is at depth d of the BFS tree rooted at s if and only if the shortest path from s to v has length d.

Proof: By strong induction on d. Base case: for $d = 0$, the only vertex with a shortest path of length 0 from s is s; we have that $L_0 = \{s\}$ by definition of BFS.

Summary: We have shown that assuming the I.H. for all $1 \leq k \leq d$, if $v \in L_{d+1}$, then the shortest path from s to v has length $d + 1$; furthermore, if the shortest path from s to v has length $d + 1$, then $v \in L_{d+1}$. Therefore the inductive step is complete.

BFS Properties Summary



- 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
- A vertex is at height h in T if and only if its shortest path from s has distance h

Implementing BFS

Implementing BFS

- Can we be more specific about how BFS works?
- Maybe give pseudocode?
- Do we need to store the levels explicitly? How should we store them?
- Key insight: we can explore the nodes in level L_{i+1} in the same order they were added to L_{i+1}. (And note that they were added before any node in L_{i+2}
- So: explore nodes in the same order they were visited!

BFS Pseudocode

```
BFS(G, s):
    Put s in a queue Q
    while Q is not empty:
        v = Q.dequeue() # take the first vertex from Q
        if v is not marked as visited:
            mark v as visited
            for each edge (v,w):
                Q.enqueue(w) # add w to Q
```

Note: this algorithm only works if at start all vertices in *G* are not marked as visited!

- Question: How can we calculate the BFS tree *T*?
- Can we *guarantee* that this is equivalent to the level-by-level version of BFS?

Proof that BFS Algorithms are Equivalent

Theorem

In `BFS(G, s)`, all nodes in level L_i are explored (removed from the queue) before any node in level L_{i+1}

We'll use the following *invariant*: if at any time the first instance of the univisted nodes in the queue are in order $v_1, v_2, \ldots, v_k$, then $L[v_1] \leq L[v_2] \leq \cdots \leq L[v_k] \leq L[v_1] + 1$.

If this invariant holds, then the theorem is true.

Proof that BFS Algorithms are Equivalent

Inductive Hypothesis: if after x iterations of the while loop, the order of the first instance of univisted nodes in the queue $v_1, v_2, \ldots, v_k$, then $L[v_1] \leq L[v_2] \leq \cdots \leq L[v_k] \leq L[v_1] + 1$.

Base Case: For $x = 0$, the queue only contains s. ✓

Inductive Step: Assume I.H. after some $x \geq 0$ iterations of the while loop. During $(x+1)$st iteration, v_1 is removed from the queue and its neighbors are added to the queue; let $u_1, \ldots, u_r$ be the unvisited neighbors that are not already in the queue. We have that $L[u_1] = L[u_2] = \cdots = L[u_r] = L[v_1] + 1$.

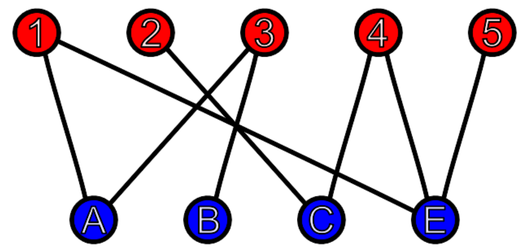
The queue now contains $v_2, v_3, \ldots, v_k, u_1, u_2, \ldots, u_r$. By I.H. and the above,

$$L[v_2] \leq L[v_3] \leq \cdots \leq L[v_k] \leq L[u_1] \leq \cdots \leq L[u_r] \leq L[v_1] + 1$$

Since we also had $L[v_1] \leq L[v_2]$ from I.H., we are done:

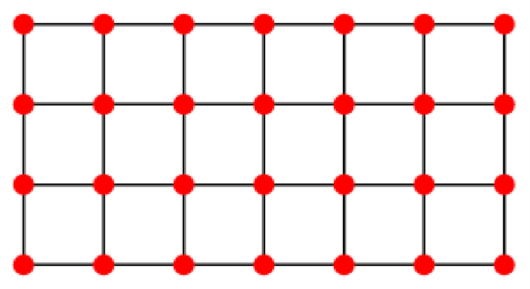
$$L[v_2] \leq L[v_3] \leq \cdots \leq L[v_k] \leq L[u_1] \leq \cdots \leq L[u_r] \leq L[v_2] + 1$$

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 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). □

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; let’s say $v_1 \in V_1$ and $v_2 \in V_2$.

Let p_1 be the path from s to v_1 in the BFS tree T, and let p_2 be the path from v_2 to s in T. Both p_1 and p_2 have length k.

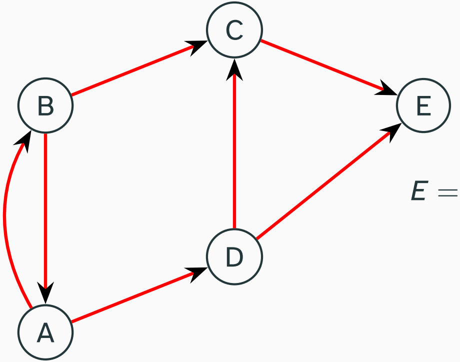
Let $p_1 = (s = u_0, u_1, u_2, \ldots, u_k = v_1)$. We know that $u_k \in V_1$, so $u_{k-1} \in V_2$; and so on. So if k is odd, $s \in V_2$; if k is even then $s \in V_1$.

Let $p_2 = (v_2 = w_0, w_2, \ldots, w_k = s)$. We know that $w_1 \in V_2$, so $w_2 \in V_1$; and so on. So if k is odd, $s \in V_1$; if k is even then $s \in V_2$. In either case (k odd or even) we have a contradiction.

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

Directed Graphs

Directed Graphs

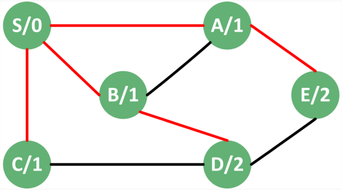


$$V = \{A, B, C, D, E\}$$

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

- 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; see lecture 5]
- 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

From BFS to DFS

- BFS explores “carefully,” creates wide trees
- Depth-first search (DFS): explore as deep as possible
- We’ll define DFS two ways

DFS: Stack Definition

```
DFS(G, s):
    Put s in a stack S
    while S is not empty:
        v = S.pop() # take the top vertex from S
        if v is not marked as visited:
            mark v as visited
            for each edge (v,w):
                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.

Let’s do an example [On Board #3]