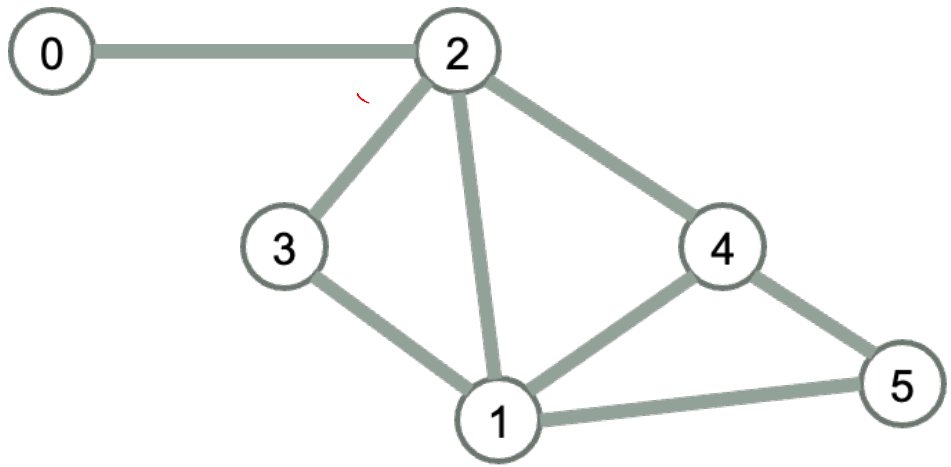
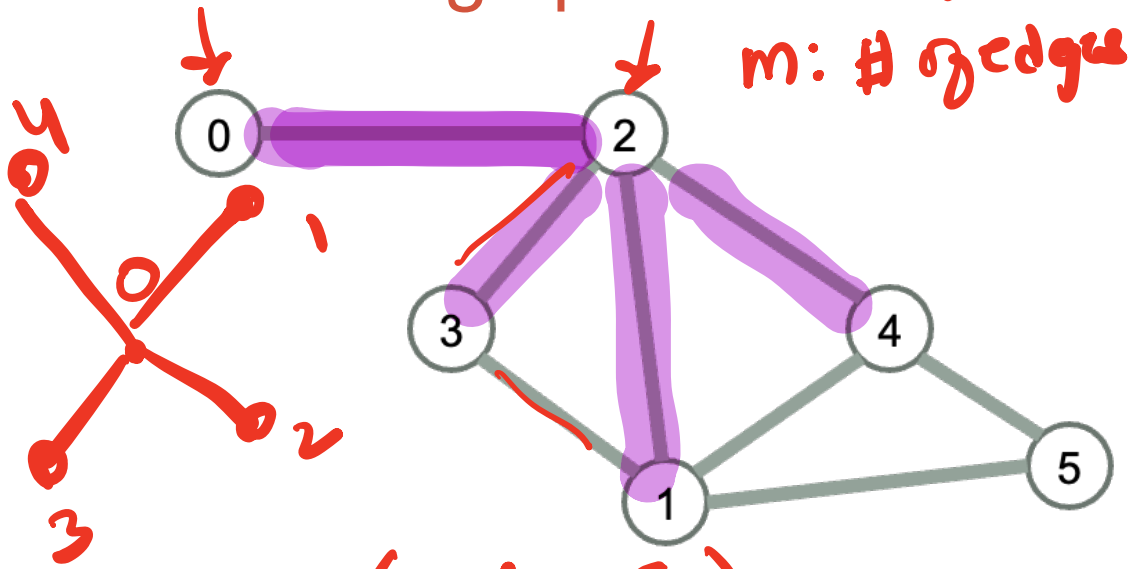
GRAPH SEARCH



Review: Adjacency list representation of graph

n: # of vertices
m: # of edges

Vertex	adjList
0	2
1	2, 3, 4, 5
2	0, 1, 3, 4
3	1, 2
4	1, 2, 5
5	1, 4



$G = (V, E)$

V: set of vertices

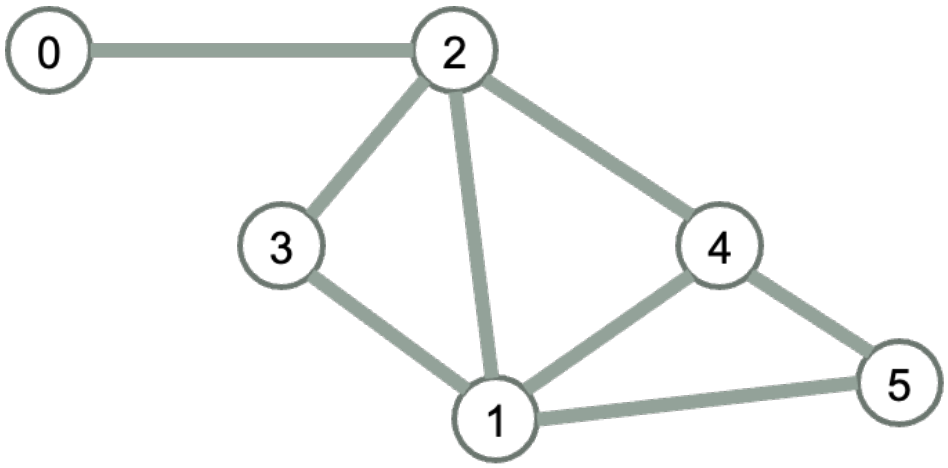
E: set of edges

O(n)

Space = $O(n + m)$

Which of these functions did you implement from last lecture's handout?

```
class graph{
  public:
    graph(int n = 0) { // n is the number of vertices
        adjList = vector<list<int>>(n);
    }
    void addEdge(int from, int to);
A   bool hasEdge(int i, int j) const;
B   vector<bool> bfs(int source) const;
C   bool isValidPath(const vector<int> & path) const; // returns true if the input path exists
D   bool isReachable(int source, int dest) const;  // returns true if a path exists from source to dest
  private:
    vector<list<int>> adjList;
};
```



E: All of them!

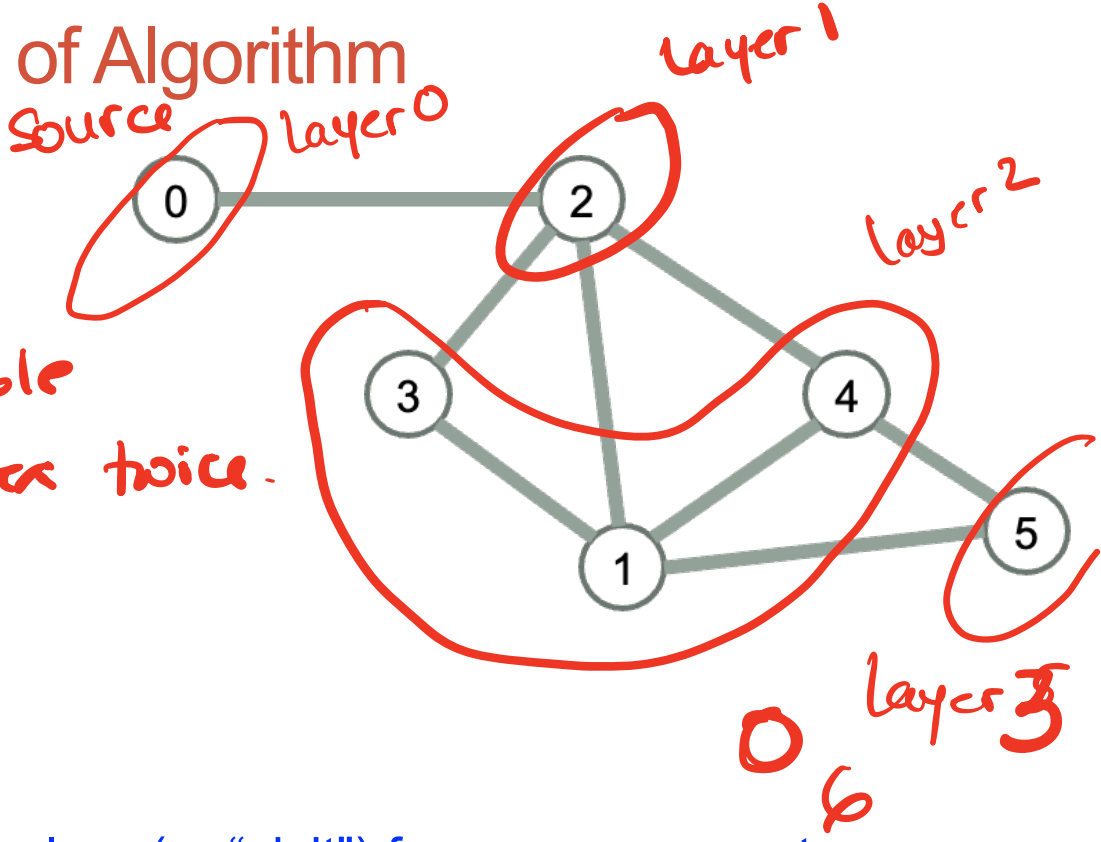
Link to hand out: https://bit.ly/CS24F23GraphsHandout

Breadth First Search: Sketch of Algorithm

Start from a source

Explore out, explore all

- vertices that are reachable
- don't explore any vertex twice.



- In general, a search algorithm would explore (or "visit") from a source vertex
  - all the vertices reachable ,
  - never exploring out from the same vertex twice
- How does the BFS algorithm ensure this?

BFS Traverse: Time Complexity (express in terms of n, m)

Input: Graph G = (V, E), source vertex s, Let n = |V|, m = |E|
Start at source *s*; O(1)
Mark all the vertices as "not visited" O(n)
Mark *s* as visited O(1)
push *s* into a queue O(1)
while the queue is not empty: (n times)
- pop the vertex *u* from the front of the queue
  - for each of *u*'s neighbor (v)
    - If v has not yet been visited (*v*):
      - Mark *v* as visited
      - Push *v* in the queue

note that the number of times the for loop runs varies which can make time analysis tricky

What is the time complexity of BFS?
A. O(n)
B. O(m)
C. O(n + m)
D. O(n^2)
E. ~~None of the above~~ O(n·m)

- How many times does the while loop run? n times
- How many times do we check if a vertex has been visited? 2·m

BFS Traverse: Time Complexity (express in terms of n, m)

Input: Graph G = (V, E), source vertex s, Let n = |V|, m = |E|
Start at source *s*;
Mark all the vertices as "not visited"
Mark *s* as visited
push *s* into a queue
while the queue is not empty:
- pop the vertex *u* from the front of the queue
  - for each of *u*'s neighbor(v)
    - If v has not yet been visited (*v*):
      - Mark *v* as visited
      - Push *v* in the queue

O(n)

O(n) for the whole program

O(m) is run time for the whole program

- How many times does the while loop run?
- How many times do we check if a vertex has been visited?

BFS Traverse: Space Complexity (express in terms of n, m)

Input: Graph G = (V, E), source vertex s, Let n = |V|, m = |E|
Start at source *s*;
Mark all the vertices as "not visited"
Mark *s* as visited
push *s* into a queue
while the queue is not empty:
- pop the vertex *u* from the front of the queue
  - for each of *u*'s neighbor (v)
    - If v has not yet been visited (*v*):
      - Mark *v* as visited
      - Push *v* in the queue

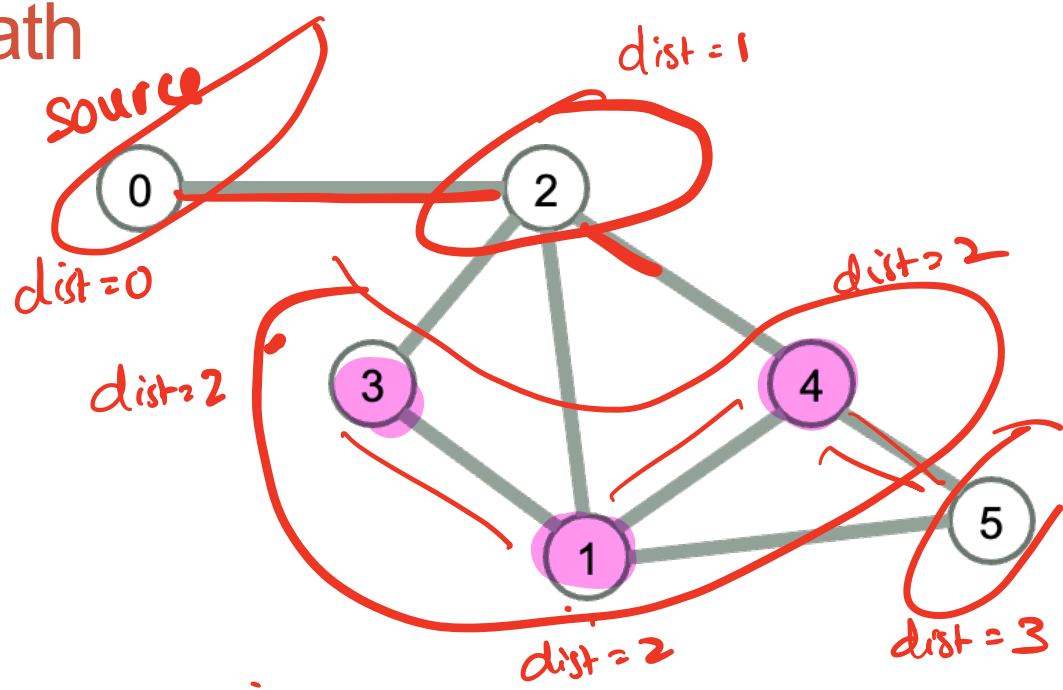
What is the space complexity of BFS?
A. O(n)
B. O(m)
C. O(n + m)
D. O(n^2)
E. None of the above

- Space complexity: Peak (additional) space usage expressed as big O

Application of BFS: shortest path

Vertex	dist	prev	adjList
0 (source)	0	-1	2
1	2	2	2, 3, 4, 5
2	1	0	0, 1, 3, 4
3	2	2	1, 2
4	2	2	1, 2, 5
5	3	4	1, 4

P X 0 2 4 5



dist(5) : shortest distance from 0 to 5

shortest path : 0, 2, 4, 5

Goal: Compute dist(v): fewest number of edges from the path from vertex **s** to **v**

BFS Shortest Path

Input: Graph G = (V, E), source vertex s, Let n = |V|, m = |E|
Start at source *s*;
Mark all the vertices as "not visited",
Mark *s* as visited
push *s* into a queue
while the queue is not empty:
- pop the vertex *u* from the front of the queue
  - for each of *u*'s neighbor (v)
    - If v has not yet been visited (*v*):
      - Mark *v* as visited
      - Push *v* in the queue

$dist(i) = \begin{cases} 0, & \text{if } i = s \\ \infty, & \text{otherwise} \end{cases}$

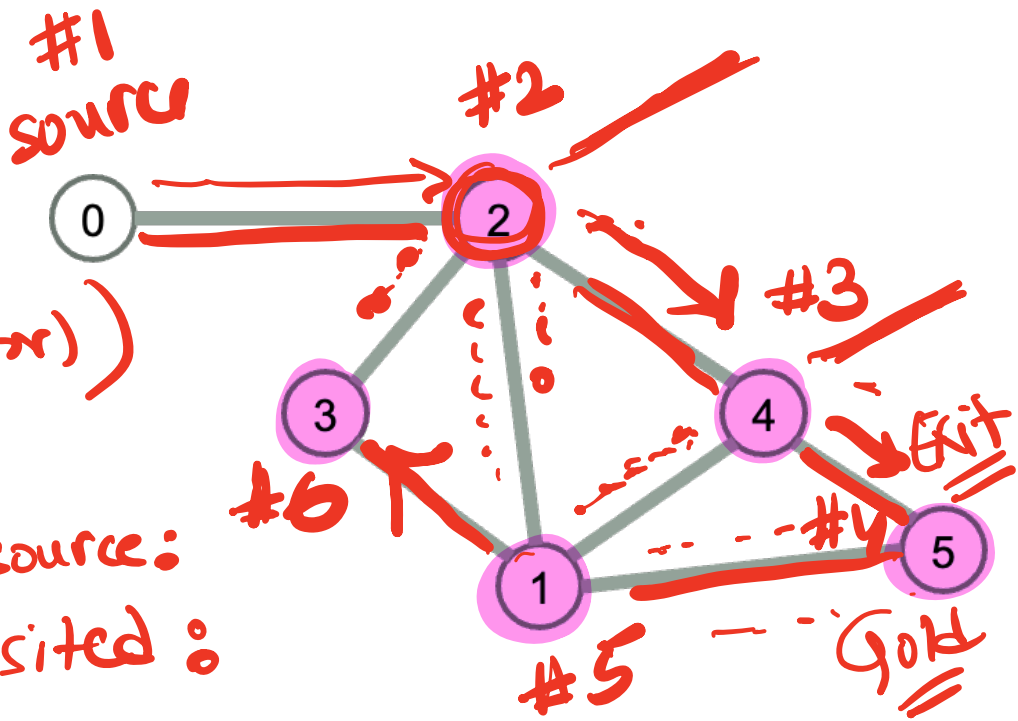
initialize prev vector to all −1

dist(v) = 1 + dist(u)
prev(v) = u

- Modify BFS to compute the shortest path from source s to all other vertices

Depth First Search

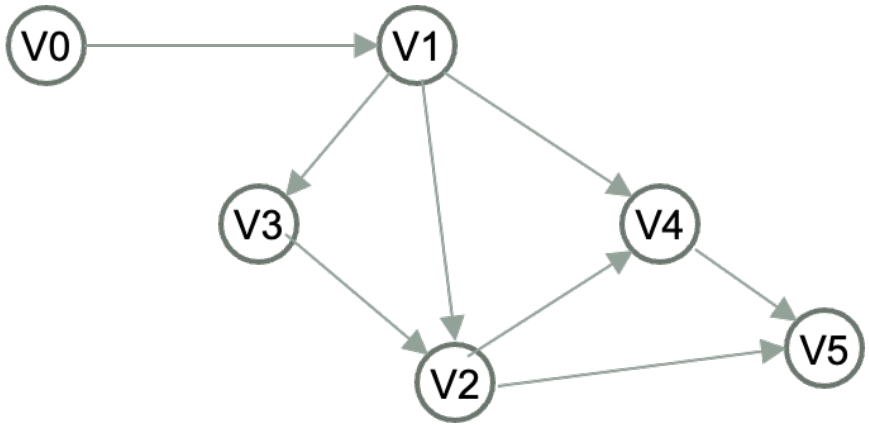
Search as far down a single path as possible, backtrack as needed




```
DFS (source, visited (vector))
    visited [source] = true
    for each neighbor V of source:
        if V is not visited:
            DFS ( V, visited )
```

Depth First Search

Search as far down a single path as possible, backtrack as needed



Assuming DFS chooses the lower number node to explore first, in what order does DFS visit the nodes in this graph?

A. V0, V1, V2, V3, V4, V5
B. V0, V1, V3, V4, V2, V5
C. V0, V1, V3, V2, V4, V5
D. V0, V1, V2, V4, V5, V3

Although C is a valid result for DFS, it doesn't follow the additional constraint stated in the question

Work to complete your handout

```
class graph{
  public:
    graph(int n = 0) { // n is the number of vertices
        adjList = vector<list<int>>(n);
    }
    void addEdge(int from, int to);
    bool hasEdge(int i, int j) const;
    vector<bool> bfs(int source) const;
    bool isValidPath(const vector<int> & path) const; // returns true if the input path exists
    bool isReachable(int source, int dest) const;  // returns true if a path exists from source to dest
    // (New!) Implement a variation of BFS to compute the shortest path from a
                source vertex to all vertices reachable from it
    // (New!) Implement depth-first search
  private:
    vector<list<int>> adjList;
};
```

Link to hand out: https://bit.ly/CS24-Graph-SearchHandout