
CSE 100: GRAPH SEARCH

Announcements

- PA3 released
 - Checkpoint due Tuesday, May 5 @10:00pm
 - Final submission due Thursday, May 14 @10:00PM

Start early! Start early! Start early! Start early! Start early!

I don't mean to nag but....

Start early, Start early...

The labs, they get jammed if you don't

Start early!

The tutors, they can't swim when the tide of students crash in, so...

Start early! Start early! Post lecture is perfect!

You will love PA3,

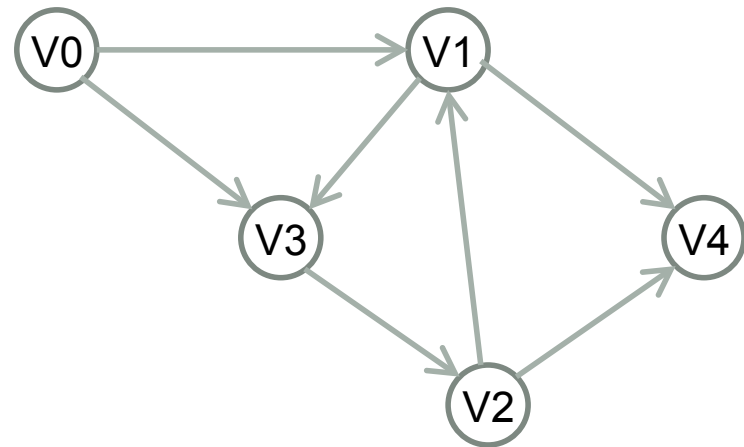
It has a mi-ni-mum spanning tree, oh sooo pretty, you'll see..

If you staaaaaaart earrrrrrlyyyyyyyy..... today

Generic approach to graph search

Generic Goals:

- Find everything that can be explored
- Don't explore anything twice

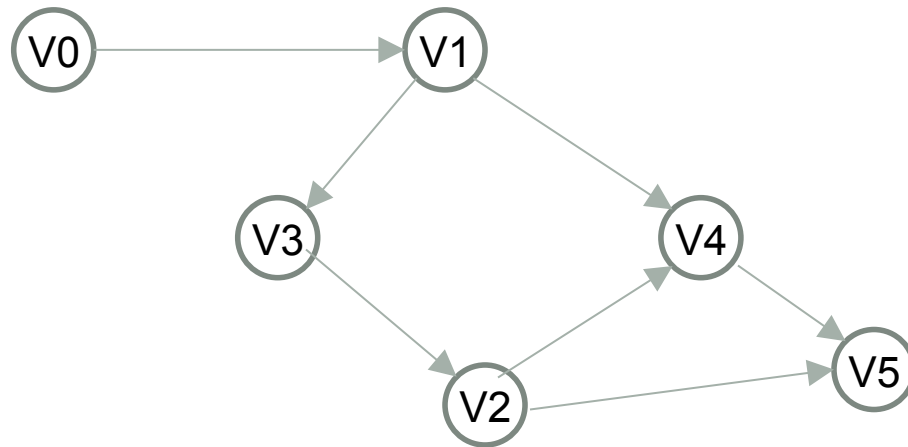


We will look at different Graph search algorithms

- Depth First Search (very useful for PA4)
- Breadth First Search
- Best-First Search (for weighted graphs) (very useful for PA3)

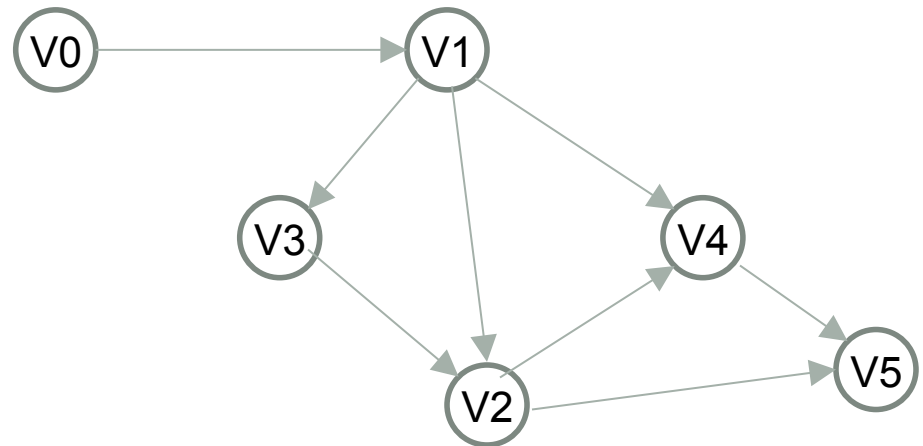
Depth First Search for Graph Traversal

- Search as far down a single path as possible before backtracking



Depth First Search for Graph Traversal

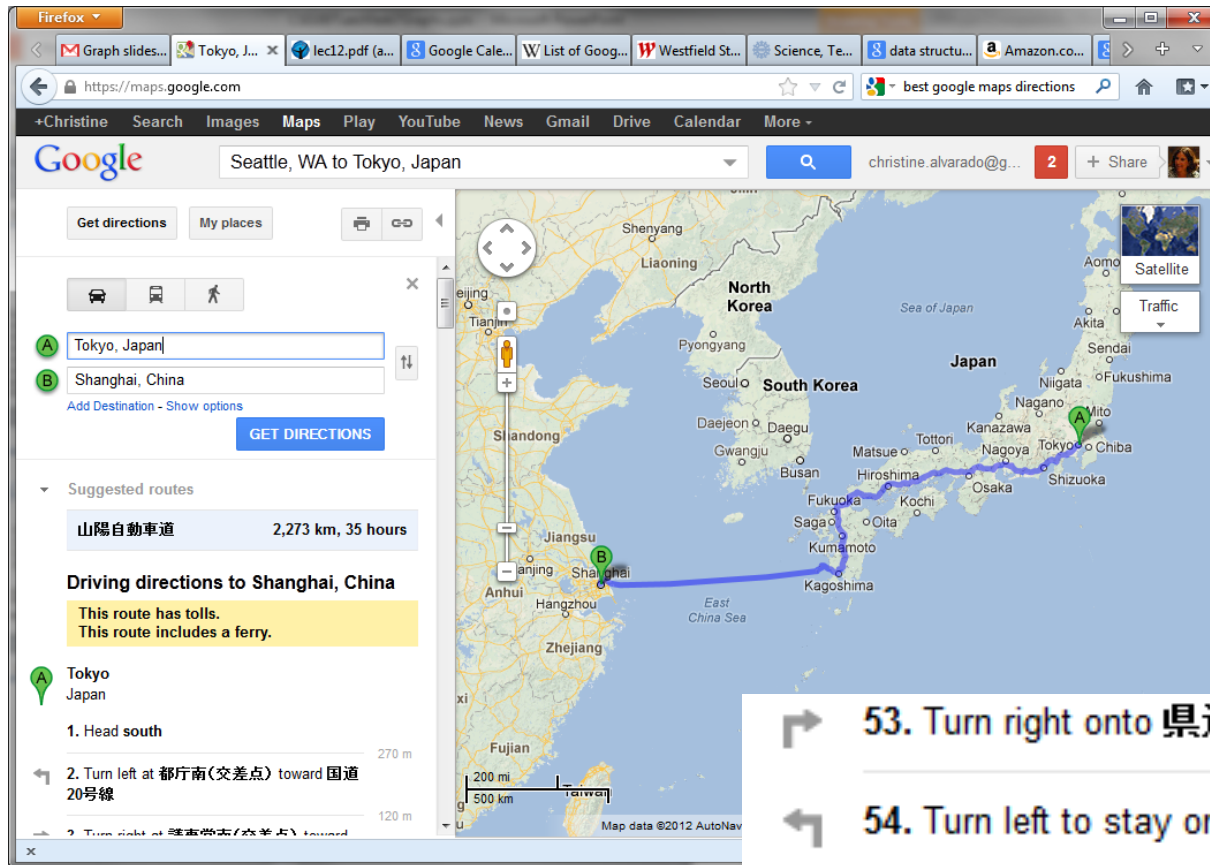
- Search as far down a single path as possible before backtracking



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

Shortest Path Algorithms



Finding the shortest route from one city to another is a natural application of graph algorithms!

(Of course there are many other examples)

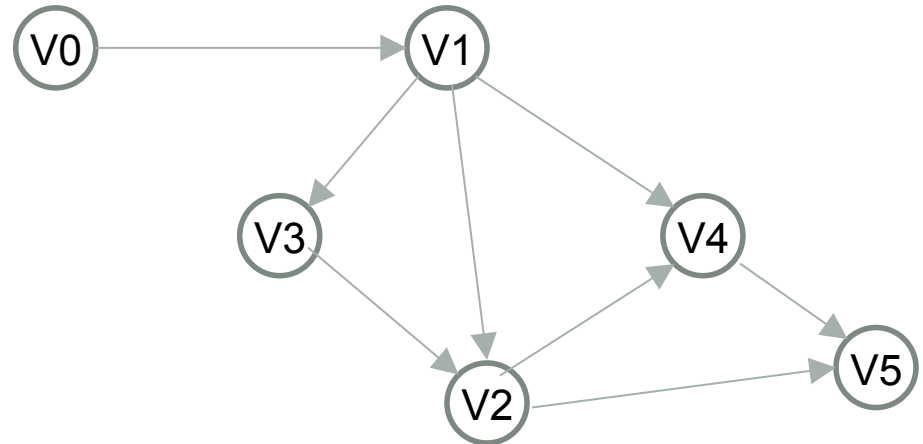
- 53. Turn right onto 県道350号線 1.9 km
- 54. Turn left to stay on 県道350号線 3.4 km
- 55. Jet ski across the Pacific Ocean 782 km
- 56. Continue straight onto 塘后支路 400 m
- 57. Continue onto 水产路

Unweighted Shortest Path

- Input: an unweighted directed graph $G = (V, E)$ and a source vertex s in V
- Output: for each vertex v in V , a representation of the shortest path in G that starts in s and ends at v and consists of the minimum number of edges compared to any other path from s to v

Depth First Search for Graph Traversal

- Search as far down a single path as possible before backtracking

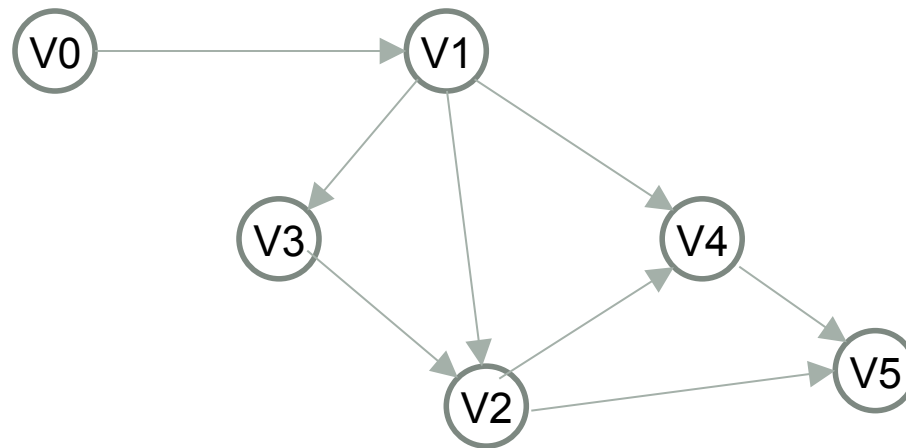


Does DFS always find the shortest path between nodes?

- A. Yes
- B. No

Breadth First Search

- Explore all the nodes reachable from a given node before moving on to the next node to explore

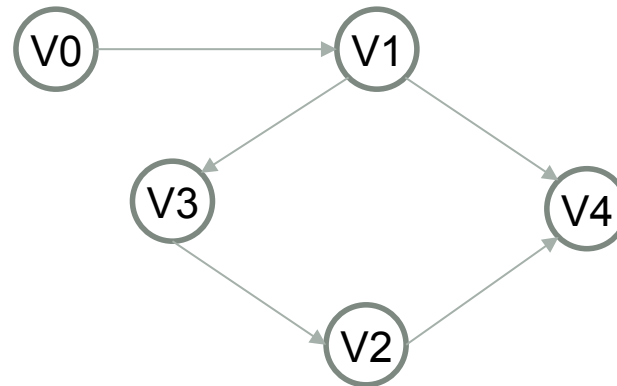


Assuming BFS chooses the lower number node to explore first, in what order does BFS visit the nodes in this graph?

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

BFS Traverse: Idea

- Input: an unweighted directed graph $G = (V, E)$ and a source vertex s in V
- Output: for each vertex v in V , a representation of the shortest path in G that starts in s and ends at v



Start at s . It has distance 0 from itself.

Consider nodes adjacent to s . They have distance 1. Mark them as visited.

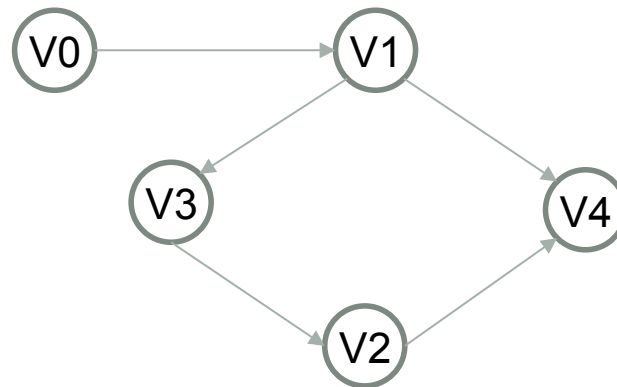
Then consider nodes that have not yet been visited

adjacent to those at distance 1. They have distance 2. Mark them as visited.

Etc. etc. until all nodes are visited.

Breadth First Search

- Explore all the nodes reachable from a given node before moving on to the next node to explore



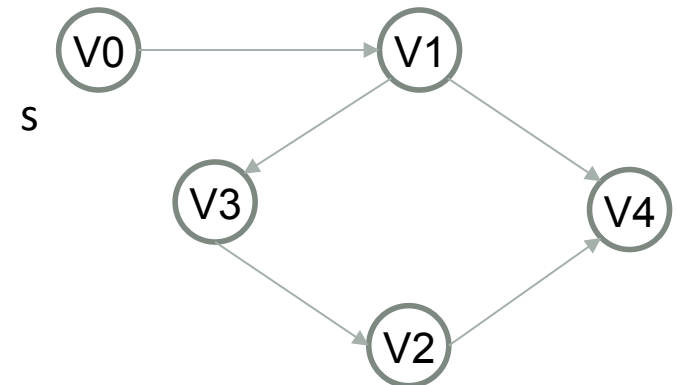
Does BFS always find the shortest path from the source to any node?

- A. Yes for unweighted graphs
- B. Yes for all graphs
- C. No

BFS Traverse: Sketch of Algorithm

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- Enqueue s into a queue
- While the queue is not empty:
 - Dequeue the vertex v from the head of the queue
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as $1 +$ the distance to v
 - Enqueue it in the queue



Queue:

BFS Traverse: Sketch of Algorithm

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- Enqueue s into a queue
- While the queue is not empty:
 - Dequeue the vertex v from the head of the queue
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as $1 + \text{the distance to } v$
 - Enqueue it in the queue

Questions:

- What data do you need to keep track of for each node?
- How can you tell if a vertex has been visited yet?
- This algorithm finds the length of the shortest path from s to all nodes. How can you also find the path itself?

BFS Traverse: Details

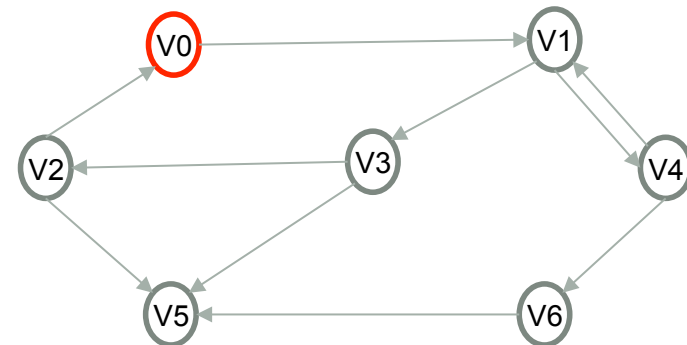
source

V0: dist=	prev=	adj: V1
V1: dist=	prev=	adj: V3, V4
V2: dist=	prev=	adj: V0, V5
V3: dist=	prev=	adj: V2, V5, V6
V4: dist=	prev=	adj: V1, V6
V5: dist=	prev=	adj:
V6: dist=	prev=	adj: V5

The queue (give source vertex dist=0 and prev=-1 and enqueue to start):

HEAD

TAIL



Representing the graph with structs

```
#include <iostream>
#include <limits>
#include <vector>
#include <queue>

using namespace std;

struct Vertex
{
    vector<int> adj; // The adjacency list
    int dist;       // The distance from the source
    int index;      // The index of this vertex
    int prev;       // The index of the vertex previous in the path
};

vector<Vertex*> createGraph()
{ ... }
```

Unweighted Shortest Path: C++ code

```
/** Traverse the graph using BFS */
```

```
void BFSTraverse( vector<Vertex*> theGraph, int from )  
{  
    queue<Vertex*> toExplore;  
    Vertex* start = theGraph[from];  
    // finish the code...
```

```
}
```

```
struct Vertex  
{  
    vector<int> adj;  
    int dist;  
    int index;  
    int prev;  
};
```

Unweighted Shortest Path: C++ code

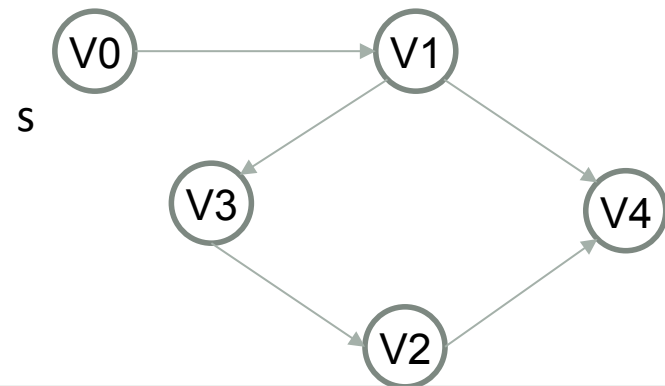
```
/** Traverse the graph using BFS */
void BFSTraverse( vector<Vertex*> theGraph, int from )
{
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    start->dist = 0;
    toExplore.push(start);
    while ( !toExplore.empty() ) {

        Vertex* next = toExplore.front();
        toExplore.pop();
        vector<int>::iterator it = next->adj.begin();
        for ( ; it != next->adj.end(); ++it ) {
            Vertex* neighbor = theGraph[*it];
            if (neighbor->dist == numeric_limits<int>::max()) {
                neighbor->dist = next->dist + 1;
                neighbor->prev = next->index;
                toExplore.push(neighbor);
            }
        }
    }
}
```

What is this algorithm??

```
Mystery(G,v) ( v is the vertex where the search starts )  
  Stack S := {} ; ( start with an empty stack )  
  for each vertex u, set visited[u] := false;  
  push S, v;  
  while (S is not empty) do  
    u := pop S;  
    if (u is not visited) then  
      visited[u] := true;  
      for each unvisited neighbour w of u  
        push S, w;  
    end if  
  end while  
END Mystery()
```

Stack:



A. BFS B. DFS C. Dijkstra's algorithm D. Nothing interesting