Graph Traversals: Breadth-First and Depth-First Search

Eric Eaton
Bryn Mawr College
Computer Science Department
What is a Graph?

Graphs are collections of vertices joined by edges

“Graph” ≡ “Network”
Example Networks

School Friendship Network
(from Moody 2001)

Airline Network
(Source: Northwest Airlines)

Terrorist Network
(by Valdis Krebs, Orgnet.com)

Protein-Protein Interactions
(by Peter Uetz)
Other Applications

• Intersections and streets within a city
• Computer networks
• Electronic circuits
• Food webs
• Gene regulatory networks
• Steps to solve a puzzle
• many more...
Outline

• **Introduction**

• Graph Basics

• Graph Search Problem
  – Breadth-First Search
  – Depth-First Search

• Complexity Analysis
Outline

• Introduction
• **Graph Basics**
• Graph Search Problem
  – Breadth-First Search
  – Depth-First Search
• Complexity Analysis
Basic Graph Definitions

- A graph $G = (V, E)$ consists of a finite set of vertices $V$ and a finite set of edges $E$.

- Each edge is a pair $(u, v)$ where $u, v \in V$.
  - $V$ and $E$ are sets, so each vertex $u \in V$ is unique, and each edge $e \in E$ is unique.
  - $v$ is adjacent to $u$.

- We will focus on two types:
  - Undirected graphs
  - Directed graphs
Undirected Graph

All edges are two-way

\[ V = \{ 1, 2, 3, 4, 5 \} \]

Edges are unordered pairs:

\[ E = \{ \{1,2\}, \{2,3\}, \{3,4\}, \{2,4\}, \{4,5\}, \{5,1\} \} \]
Directed Graph

All edges are “one-way” as indicated by the arrows

- $V = \{1, 2, 3, 4, 5\}$
- Edges are ordered pairs:
  $E = \{(1,2), (2,4), (3,2), (4,3), (4,5), (5,1), (5,4)\}$
Degree

Undirected Graphs

degree(\(u\)): the number of edges \(\{u, v\}\) for all \(v \in V\)
Degree

Undirected Graphs
\( \text{degree}(u) \): the number of edges \( \{u, v\} \) for all \( v \in V \)

Directed Graphs
\( \text{in-degree}(u) \): the number of edges \( (v, u) \) for all \( v \in V \)
\( \text{out-degree}(u) \): the number of edges \( (u, v) \) for all \( v \in V \)
Paths in Graphs

- A **path** in a graph is a sequence of vertices $w_1, w_2, \ldots, w_n$ s.t. $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$

- The path’s **length** is the number of edges on the path
  - The length of the path from a vertex to itself is 0

- In a **simple path**, all vertices are distinct
  - The first and last vertices may be the same

![Path length = 4](image-url)
 Paths in Graphs

- How many simple paths are there from 1 to 4 and what are their lengths?
Outline

• Introduction

• **Graph Basics**

• Graph Search Problem
  – Breadth-First Search
  – Depth-First Search

• Complexity Analysis
Outline

• Introduction
• Graph Basics

• **Graph Search Problem**
  – Breadth-First Search
  – Depth-First Search

• Complexity Analysis
Graph Search Problem

- **Goal**: Find a simple path from a starting vertex to a goal vertex.

- What applications can be framed as instances of this problem?
Intuition

- From starting vertex, keep expanding vertices until we find the goal
Intuition

- From starting vertex, keep expanding vertices until we find the goal
Intuition

- From starting vertex, keep expanding vertices until we find the goal
Intuition

- From starting vertex, keep expanding vertices until we find the goal
Intuition

- From starting vertex, keep expanding vertices until we find the goal

- Breadth-First: expand shallowest unexpanded vertex
- Depth-First: expand deepest unexpanded vertex
Queuing Function

- Used to maintain a ranked list of nodes that are candidates for expansion
  - Called the “fringe”

- Substituting different queuing functions yields different searches
Protection Against Cycles

- We need to guard against cycles
  - Mark each vertex as “closed” when we encounter it
  - Do not consider closed vertices again
Bookkeeping Structures

- **Node:**
  - vertex ID
  - predecessor node
  - path length

- **Problem:**
  - graph
  - starting vertex
  - goalTest(Vertex v) – tests if vertex is a goal state
General Graph Search

// problem describes the graph, start vertex, and goal test
// queueingFn is a comparator function that ranks two states
// graphSearch returns either a goal node or failure

graphSearch(problem, queuingFn) {
  open = {}, closed = {}               //empty lists
  queuingFn(open, new Node(problem.startvertex)) //init

  loop {
    if empty(open) then return FAILURE   //no nodes remain
    c = removeFront(open)               //get current node
    if problem.goalTest(c.vertex)       //goal test
      return c
    if c.vertex is not in closed {       //avoid duplicates
      add c.vertex to closed
      for each Vertex w adjacent to c.vertex   //expand node
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
Application: Route Finding

start

A

B

C

D

E

F

H

G

J

I

goal
Breadth-First Search

Expands the “shallowest” vertex
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
```
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

\[
\text{graphSearch}(\text{problem, queuingFn}) \{
\text{open} = \{\}, \text{closed} = \{
\text{queuingFn}(\text{open, new Node(\text{problem.startvertex}))}
\text{loop} \{
\text{if empty(open) then return FAILURE}
\text{c = removeFront(open)}
\text{if problem.goalTest(c.vertex) then return c}
\text{if c.vertex is not in closed} \{
\text{add c.vertex to closed}
\text{for each w adjacent to c.vertex}
\text{if w is not in closed}
\text{queuingFn(open, new Node(w, c))};
\}\}
\}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }
    }
}
**Application: Route Finding (BFS)**

```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
            if w is not in closed
                queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
            if w is not in closed
                queuingFn(open, new Node(w, c));
        }
    }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
  }}

open list
(C,1,A)  
(D,1,A)

closed list
A
B

node c
(B,1,A)
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex)
      return c
    if c.vertex is not in closed
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
  }
}
Application: Route Finding (BFS)

open list
(C,1,A)
(D,1,A)
(C,2,B)
(F,2,B)

closed list
A
B

node c
(B,1,A)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c))
    }
  }

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
  }}

open list
(D,1,A)
(C,2,B)
(F,2,B)

closed list

A
B
C

node c
(C,1,A)
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }
    }
}
```
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }
    }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
Application: Route Finding (BFS)

**open list**

| (F,2,C) | (E,2,C) | (E,2,D) |

**closed list**

| A | B | C | D | F |

**node c**

(F,2,B)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
Application: Route Finding (BFS)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
Application: Route Finding (BFS)

```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
            if w is not in closed
                queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (BFS)

open list
(E,2,C)
(E,2,D)
(G,3,F)
(H,3,F)

closed list
A
B
C
D
F

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }}}}
Application: Route Finding (BFS)

open list
(E,2,D)
(G,3,F)
(H,3,F)

closed list
A
B
C
D
F

node c
(E,2,C)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
    }}}}
**Application: Route Finding (BFS)**

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```
Application: Route Finding (BFS)

open list
(G,3,F)
(H,3,F)

closed list
A
B
C
D
E
F

node c
(E,2,D)

graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }}}}
Application: Route Finding (BFS)

```java
public class graphSearch {
    // Problem graph
    public static class Node {
        public final Problem problem; // the problem to solve
        public final Node parent; // parent node
        public final Vertex vertex; // vertex the node represents
        // Other fields...
    }

    // Problem interface
    public interface Problem {
        public boolean startVertex() throws Exception;
        public boolean goalTest(Vertex v) throws Exception;
        public boolean isGoalCompatible(Vertex v) throws Exception;
        // Other methods...
    }

    // Queuing function
    public interface QueuingFn {
        public void add(Node node) throws Exception;
        public void removeFront(Node[] open) throws Exception;
        // Other methods...
    }

    // Start vertex
    public static Node startVertex() throws Exception {
        // Create a node for the start vertex
        Node startNode = new Node(); // Initialize node
        startNode.vertex = startVertex(); // Make it the start vertex
        startNode.parent = null; // No parent for the start node
        add(startNode); // Add to open list
        removeFront(open); // Remove front
        return startNode; // Return the node
    }

    // Goal function
    public static Node goalFunction(Node node) throws Exception {
        // Check if the node is a goal
        if (node.isGoalCompatible()) {
            return node; // Return the node
        }
        // If not, check if the goal is adjacent to the current node
        for (Node w : node.vertex.adjacentNodes()) {
            if (isGoalCompatible(w)) {
                // Add to open list
                add(w);
                removeFront(open);
                // Recursively search
                search(w, node);
            }
        }
        return null; // If no goal found
    }

    // Search function
    public static void search(Node node) throws Exception {
        // Add node to closed list
        add(node);
        // Iterate over neighbors
        for (Node neighbor : node.vertex.adjacentNodes()) {
            // If neighbor is in closed, continue
            if (isGoalCompatible(neighbor)) continue;
            // Add neighbor to open list
            add(neighbor);
            // Recursively search
            search(neighbor, node);
        }
    }

    // Utility methods
    public static void add(Node node) throws Exception {
        // Add node to open list
    }
    public static void removeFront(Node[] open) throws Exception {
        // Remove front from open list
    }
}
```

- **open list**: (H, 3, F)
- **closed list**: A, B, C, D, E, F
- **node c**: (G, 3, F)
Application: Route Finding (BFS)

```java
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```
Breadth-First Search

A
---
B
  
G
  
D
  
C
  
F
  
H
  
E
  
J
  
I

start 0
Breadth-First Search

start

A

0

B

1

C

D

E

F

G

H

J

I
Breadth-First Search

start

A -> B
D -> C
E
J
I

0 1 2
Breadth-First Search

start

A -> B -> C -> D -> E
A -> H
A -> F
A -> G
A -> J
A -> I
Breadth-First Search

start

A

B

C

D

E

F

G

H

I

J
Depth-First Search

Expands the “deepest” vertex
Application: Route Finding (DFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
  }
}
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
Application: Route Finding (DFS)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
Application: Route Finding (DFS)

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex if w is not in closed
                queuingFn(open, new Node(w, c));
        }
    }
}
```

Node c (B,1,A)

**open list**
(C,1,A)
(D,1,A)

**closed list**
A
B

Need to add (C,2,B) and (F,2,B) to open

Since DFS expands the deepest node, what must we insure about the open list?

What queuing function should we use?
Application: Route Finding (DFS)

```java
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w, c));
    }
  }
}
```

DFS uses a LIFO Stack!
Application: Route Finding (DFS)

```
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) then return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
Application: Route Finding (DFS)

open list
(C,2,B)
(C,1,A)
(D,1,A)

closed list
A
B
F

node c
(F,2,B)

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
Application: Route Finding (DFS)

open list
(H,3,F)
(G,3,F)
(C,2,B)
(C,1,A)
(D,1,A)

closed list
A
B
F

node c
(F,2,B)

goal

graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
  }}}
Application: Route Finding (DFS)

**Open List:**
- (H,3,F)
- (G,3,F)
- (C,2,B)
- (C,1,A)
- (D,1,A)

**Closed List:**
- A
- B
- F

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) then return c
    if c.vertex is not in closed {
      add c.vertex to closed
      for each w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
    }
  }
}
```
What we’ve found so far...

**Breadth-First Search** *(FIFO Queue)*

- Solves **unweighted shortest path problem**:
  
  Finds the shortest path between vertices if edges are unweighted (or equal cost)

**Depth-First Search** *(LIFO Stack)*

- Finds nearby goals quickly if lucky
- If unlucky, finds nearby goals very slowly
Application: 8-Puzzle

Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles as to produce a desired goal configuration.
Application: 8-Puzzle

• What are the vertices?
• What are the edges?
• Starting vertex?
• Goal vertex / vertices?
Application: 8-Puzzle

• **What are the vertices?** Each vertex corresponds to a particular tile configuration

• **What are the edges?** Consider four operators: Move Blank Square Left, Right, Up or Down
  – This is a more efficient encoding than considering each of 4 moves for each tile

  The edges signify applying an operator to a board configuration

• **Initial state?** A particular board configuration

• **Goal vertex?** A particular board configuration
Outline

• Introduction
• Graph Basics
• Graph Search Problem
  – Breadth-First Search
  – Depth-First Search
• Complexity Analysis
Outline

• Introduction

• Graph Basics

• Graph Search Problem
  – Breadth-First Search
  – Depth-First Search

• Complexity Analysis
What is the Time Complexity of BFS & DFS?

- In the worst case, the goal vertex won’t be found

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}

    queuingFn(open, new Node(problem.startvertex))

    loop {
        if empty(open) then return FAILURE

        c = removeFront(open)

        if problem.goalTest(c.vertex) return c

        if c.vertex is not in closed {
            add c.vertex to closed
            for each Vertex w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```
In the worst case, the goal vertex won’t be found

Each vertex is in the queue at most once, so the outer loop runs at most $|V|$ iterations

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each Vertex w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }
    }
}
```
What is the Time Complexity of BFS & DFS?

- In the worst case, the goal vertex won’t be found

```
graphSearch(problem, queuingFn) {
  open = {}, closed = {}
  queuingFn(open, new Node(problem.startvertex))
  loop {
    if empty(open) then return FAILURE
    c = removeFront(open)
    if problem.goalTest(c.vertex) return c
    if c.vertex is not in closed
      add c.vertex to closed
      for each Vertex w adjacent to c.vertex
        if w is not in closed
          queuingFn(open, new Node(w,c));
  }
}
```

Each vertex is in the queue at most once, so the outer loop runs at most $|V|$ iterations

Performance will depend on the time for getAdjacent()
Graph Representation: Adjacency Matrix

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 0 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]
Graph Representation: Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

What is the performance of getAdjacent(u)?
Graph Representation: Adjacency List

What is the performance of `getAdjacent(u)`?
What is the Time Complexity of BFS & DFS?

■ Using an adjacency matrix:

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {}
    queuingFn(open, new Node(problem.startvertex))
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open)
        if problem.goalTest(c.vertex) return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each Vertex w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w,c));
        }
    }
}
```

$|V| \text{ iterations}$

$O(|V|)$
What is the Time Complexity of BFS & DFS?

- Using an adjacency matrix: $O(|V|^2)$
What is the Time Complexity of BFS & DFS?

- Using an adjacency list:

```java
graphSearch(problem, queuingFn) {
    open = {}, closed = {};
    queuingFn(open, new Node(problem.startvertex));
    loop {
        if empty(open) then return FAILURE
        c = removeFront(open);
        if problem.goalTest(c.vertex) return c
        if c.vertex is not in closed {
            add c.vertex to closed
            for each Vertex w adjacent to c.vertex
                if w is not in closed
                    queuingFn(open, new Node(w, c));
        }
    }
}
```

\(|V| \) iterations

\(O(\text{out-degree}(c.\text{vertex}))\)
What is the Time Complexity of BFS & DFS?

- For an adjacency list, looping over all adjacent vertices of $u$ will be $O(\text{out-degree}(u))$

- Therefore, the traversal performance is

$$O \left( \sum_{i=1}^{\left| V \right|} \text{out-degree}(v_i) \right) = O(|E|)$$

since the inner loop is repeated $O(\left| V \right|)$ times

- However, in a disconnected graph, we must still look at every vertex, so the performance is $O(\left| V \right| + |E|)$

How do these terms compare?
Sparse vs Dense Graphs

- A **sparse graph** is one with “few” edges. That is $|E| = O(|V|)$

- A **dense graph** is one with “many” edges. That is $|E| = O(|V|^2)$
What is the Time Complexity of BFS & DFS?

- For an adjacency list, getAdjacent($u$) will be $O(\text{out-degree}(u))$

- Therefore, the traversal performance is

$$O \left( \sum_{i=1}^{\mid V \mid} \text{out-degree}(v_i) \right) = O(|E|)$$

since getAdjacent is done $O(\mid V \mid)$ times

- However, in a disconnected graph, we must still look at every vertex, so the performance is $O(\mid V \mid + |E|)$

Ranges from $O(\mid V \mid)$ to $O(\mid V \mid^2)$, depending on density
What is the **Space** Complexity of BFS & DFS?

- Really depends on the graph representation

<table>
<thead>
<tr>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1 0 1 0 0 0</td>
</tr>
<tr>
<td>2 0 0 0 1 0</td>
</tr>
<tr>
<td>3 0 1 0 0 0</td>
</tr>
<tr>
<td>4 0 0 1 0 1</td>
</tr>
<tr>
<td>5 1 0 0 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → 2</td>
</tr>
<tr>
<td>2 → 4</td>
</tr>
<tr>
<td>3 → 2</td>
</tr>
<tr>
<td>4 → 3 → 5</td>
</tr>
<tr>
<td>5 → 1 → 4</td>
</tr>
</tbody>
</table>
What is the **Space Complexity** of BFS & DFS?

- Really depends on the graph representation

**Adjacency Matrix**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Space Complexity**

\( O( |V|^2) \)

**Adjacency List**

- 1 → 2
- 2 → 4
- 3 → 2
- 4 → 3 → 5
- 5 → 1 → 4

**Space Complexity**

\( O( |V| + |E| ) \)
Does BFS find Shortest Paths in Weighted Graphs?
Summary

• Breadth-First Search
  – Solves unweighted shortest path problem
  – Uses FIFO queue
  – Traverses the graph in level-order

• Depth-First Search
  – Uses LIFO stack
  – Takes a “deep-dive” into the graph

• Time/Space Complexity: $O( |V| + |E| )$