Graphs
Unweighted Graphs

Bjarki Ágúst Guðmundsson
Tómas Ken Magnússon

School of Computer Science
Reykjavík University

Árangursrík forritun og lausn verkefna
Today we’re going to cover

- Graph basics
- Graph representation (recap)
- Depth-first search
- Connected components
- DFS tree
- Bridges
- Strongly connected components
- Topological sort
- Breadth-first search
- Shortest paths in unweighted graphs
What is a graph?
What is a graph?

- Vertices
  - Road intersections
  - Computers
  - Floors in a house
  - Objects
What is a graph?

- **Vertices**
  - Road intersections
  - Computers
  - Floors in a house
  - Objects

- **Edges**
  - Roads
  - Ethernet cables
  - Stairs or elevators
  - Relation between objects
Types of edges

- Unweighted
Types of edges

- Unweighted or Weighted
Types of edges

- Unweighted or Weighted
- Undirected
Types of edges

- Unweighted or Weighted
- Undirected or Directed
Types of edges

- Unweighted or Weighted
- Undirected or Directed
Multigraphs
Multigraphs

- Multiple edges
Multigraphs

- Multiple edges
- Self-loops
Adjacency list

0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2

vector<int> adj[4];
adj[0].push_back(1);
adj[0].push_back(2);
adj[1].push_back(0);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
adj[3].push_back(2);
Adjacency list (directed)

0: 1
1: 2
2: 0, 1, 3
3:

vector<int> adj[4];
adj[0].push_back(1);
adj[1].push_back(2);
adj[2].push_back(0);
adj[2].push_back(1);
adj[2].push_back(3);
Vertex properties (undirected graph)

Degree of a vertex
- Number of adjacent edges
- Number of adjacent vertices

Handshaking lemma
\[ \sum_{v \in V} \deg(v) = 2 \|E\]
Vertex properties (undirected graph)

- Degree of a vertex
  - Number of adjacent edges
  - Number of adjacent vertices

\[
\sum_{v \in V} \deg(v) = 2 \sum_{j \in V} j = 2 + 2 + 3 + 1 = 8
\]
Vertex properties (undirected graph)

- **Degree of a vertex**
  - Number of adjacent edges
  - Number of adjacent vertices

- **Handshaking lemma**

\[ \sum_{v \in V} \deg(v) = 2|V| \]
Vertex properties (undirected graph)

- **Degree of a vertex**
  - Number of adjacent edges
  - Number of adjacent vertices

- **Handshaking lemma**

\[ \sum_{v \in V} \deg(v) = 2|V| \]

\[
2 + 2 + 3 + 1 = 2 \times 4
\]
Vertex properties (undirected graph)

0: 1, 2
1: 0, 2
2: 0, 1, 3
3: 2

adj[0].size() // 2
adj[1].size() // 2
adj[2].size() // 3
adj[3].size() // 1
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges
Vertex properties (directed graph)

- **Outdegree of a vertex**
  - Number of outgoing edges

- **Indegree of a vertex**
  - Number of incoming edges
Vertex properties (directed graph)

- Outdegree of a vertex
  - Number of outgoing edges
- Indegree of a vertex
  - Number of incoming edges
Vertex properties (directed graph)

- **Outdegree of a vertex**
  - Number of outgoing edges

- **Indegree of a vertex**
  - Number of incoming edges
Adjacency list (directed)

0: 1
1: 2
2: 0, 1, 3
3:

adj[0].size() // 1
adj[1].size() // 1
adj[2].size() // 3
adj[3].size() // 0
Paths

- Path / Walk / Trail:

  \[ e_1 e_2 \ldots e_k \]

  such that

  \[ e_i \in E \]

  \[ e_i = e_j \Rightarrow i = j \]

  \[ \text{to}(e_i) = \text{from}(e_{i+1}) \]
Paths

- Path / Walk / Trail:

\[ e_1 e_2 \ldots e_k \]

such that

\[ e_i \in E \]

\[ e_i = e_j \Rightarrow i = j \]

\[ \text{to}(e_i) = \text{from}(e_{i+1}) \]
Paths

► Path / Walk / Trail:

\[ e_1 e_2 \ldots e_k \]

such that

\[ e_i \in E \]

\[ e_i = e_j \Rightarrow i = j \]

\[ \text{to}(e_i) = \text{from}(e_{i+1}) \]
Paths

- Path / Walk / Trail:

  \[ e_1 e_2 \ldots e_k \]

  such that

  \[ e_i \in E \]

  \[ e_i = e_j \Rightarrow i = j \]

  \[ \text{to}(e_i) = \text{from}(e_{i+1}) \]
Cycles

- Cycle / Circuit / Tour:

\[ e_1 e_2 \ldots e_k \]

such that

\[ e_i \in E \]

\[ e_i = e_j \Rightarrow i = j \]

\[ \text{to}(e_i) = \text{from}(e_{i+1}) \]

\[ \text{from}(e_1) = \text{to}(e_k) \]
Cycles

- Cycle / Circuit / Tour:

\[ e_1 e_2 \ldots e_k \]

such that

\[ e_i \in E \]

\[ e_i = e_j \Rightarrow i = j \]

\[ \text{to}(e_i) = \text{from}(e_{i+1}) \]

\[ \text{from}(e_1) = \text{to}(e_k) \]
Cycles

- **Cycle / Circuit / Tour:**

  \[ e_1 e_2 \ldots e_k \]

  such that

  \[ e_i \in E \]

  \[ e_i = e_j \Rightarrow i = j \]

  \[ \text{to}(e_i) = \text{from}(e_{i+1}) \]

  \[ \text{from}(e_1) = \text{to}(e_k) \]
Cycles

- Cycle / Circuit / Tour:

\[ e_1 e_2 \ldots e_k \]

such that

\[ e_i \in E \]

\[ e_i = e_j \Rightarrow i = j \]

\[ \text{to}(e_i) = \text{from}(e_{i+1}) \]

\[ \text{from}(e_1) = \text{to}(e_k) \]
Depth-first search

- Given a graph (either directed or undirected) and two vertices $u$ and $v$, does there exist a path from $u$ to $v$?
- Depth-first search is an algorithm for finding such a path, if one exists

- It traverses the graph in depth-first order, starting from the initial vertex $u$
- We don’t actually have to specify a $v$, since we can just let it visit all reachable vertices from $u$ (and still same time complexity)

- But what is the time complexity?
- Each vertex is visited once, and each edge is traversed once
- $O(n + m)$
Depth-first search

Stack: | 0 1 2 3 4 5 6 7 8 9 10

marked | 0 0 0 0 0 0 0 0 0 0 0
Depth-first search

Stack: 0 |

marked | 1 0 0 0 0 0 0 0 0 0 0 0 0 0
Depth-first search

Stack: 0 |  
marked | 1 0 0 0 0 0 0 0 0 0 0 0
Depth-first search

Stack: 0 | 2 1

marked

| 1 1 1 0 0 0 0 0 0 0 0 0 |
Depth-first search

Stack: 2 | 1

marked: 1 1 1 0 0 0 0 0 0 0 0
Depth-first search

Stack: 2 | 1

marked | 1 1 1 0 0 0 0 0 0 0 0 0
Depth-first search

Stack: 2 | 3 1

marked | 1 1 1 1 0 0 0 0 0 0 0
Depth-first search

Stack: \( 3 \mid 1 \)

marked: 1 1 1 1 0 0 0 0 0 0 0
Depth-first search

Stack: 1 |

marked | 1 1 1 1 1 0 0 0 0 0 0 0 0
Depth-first search

Stack: 1 |

marked | 1 1 1 1 0 0 0 0 0 0 0 0
Depth-first search

Stack: 1 | 4

marked | 1 1 1 1 1 0 0 0 0 0 0
Depth-first search

Stack: 4 | 0 1 2 3 4 5 6 7 8 9 10
marked: 1 1 1 1 1 0 0 0 0 0 0
Depth-first search

Stack: 4 |

marked | 1 1 1 1 1 1 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
Depth-first search

Stack: 4 | 5

marked | 1 1 1 1 1 1 1 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
Depth-first search

Stack: 5

marked: 1 1 1 1 1 1 1 0 0 0 0 0
Depth-first search

Stack: 5 |

marked | 0 1 2 3 4 5 6 7 8 9 10
        | 1 1 1 1 1 1 1 0 0 0 0 0
Depth-first search

Stack: 5 | 8 6 7

marked 1 1 1 1 1 1 1 1 1 1 0 0
Depth-first search

Stack: 8 6 7

marked: 1 1 1 1 1 1 1 1 1 1 1 0 0
Depth-first search

Stack: 8 | 6 7

marked: 1 1 1 1 1 1 1 1 1 1 0 0
Depth-first search

Stack: 8 | 9 6 7

marked: 1 1 1 1 1 1 1 1 1 1 0
Depth-first search

Stack: 9 | 6 7

marked | 1 1 1 1 1 1 1 1 1 1 1 0
Depth-first search

Stack: 6 |  
marked | 0 1 1 1 1 1 1 1 1 1 1 0
Depth-first search

Stack: 7 | 8
marked: 1 1 1 1 1 1 1 1 1 1 1 1 0

Nodes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Graph showing the order of traversal and marked nodes.
Depth-first search

Stack: 7 |

marked 0 1 2 3 4 5 6 7 8 9 10

1 1 1 1 1 1 1 1 1 1 0
Depth-first search

Stack: 7 | 10

marked: 1 1 1 1 1 1 1 1 1 1 1
Depth-first search

Stack: 10 |

marked

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Depth-first search

Stack:  

marked  

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
Depth-first search

vector<int> adj[1000];
vector<bool> visited(1000, false);

void dfs(int u) {
    if (visited[u]) {
        return;
    }
    visited[u] = true;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        dfs(v);
    }
}
An undirected graph can be partitioned into connected components.

A connected component is a maximal subset of the vertices such that each pair of vertices is reachable from each other.

We’ve already seen this in a couple of problems, but we’ve been using Union-Find to keep track of the components.
Connected components
Connected components

1. Connected component with nodes 7, 1, 4
2. Connected component with nodes 5, 3, 6
3. Isolated node 2
Connected components

- Also possible to find these components using depth-first search
- Pick some vertex we don’t know anything about, and do a depth-first search out from it
- All vertices reachable from that starting vertex are in the same component
- Repeat this process until you have all the components
- Time complexity is $O(n + m)$
Connected components

```cpp
vector<int> adj[1000];
vector<int> component(1000, -1);

void find_component(int cur_comp, int u) {
    if (component[u] != -1) {
        return;
    }
    component[u] = cur_comp;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        find_component(cur_comp, v);
    }
}

int components = 0;
for (int u = 0; u < n; u++) {
    if (component[u] == -1) {
        find_component(components, u);
        components++;
    }
}
```
When we do a depth-first search from a certain vertex, the path that we take forms a tree.

When we go from a vertex to another vertex that we haven’t visited before, the edge that we take is called a forward edge.

When we go from a vertex to another vertex that we’ve already visited before, the edge that we take is called a backward edge.

To be more specific: the forward edges form a tree.

see example
Depth-first search tree

- This tree of forward edges, along with the backward edges, can be analyzed to get a lot of information about the original graph.

- For example: a backward edge represents a cycle in the original graph.

- If there are no backward edges, then there are no cycles in the original graph (i.e. the graph is acyclic).
Analyzing the DFS tree

- Let’s take a closer look at the depth-first search tree
- First, let’s number each of the vertices in the order that we visit them in the depth-first search
- For each vertex, we want to know the smallest number of a vertex that we visited when exploring the subtree rooted at the current vertex
- Why? We’ll see in a bit...
- *see example*
const int n = 1000;
vector<int> adj[n];
vector<int> low(n), num(n, -1);
int curNum = 0;

void analyze(int u, int p) {
    low[u] = num[u] = curNum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (v == p) continue;
        if (num[v] == -1) {
            analyze(v, u);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], num[v]);
        }
    }
}

for (int u = 0; u < n; u++) {
    if (num[u] == -1) {
        analyze(u, -1);
    }
}
Analyzing the DFS tree

- Time complexity of this is just $O(n + m)$, since this is basically just one depth-first search
- Now, as promised, let’s see some applications of this
We have an undirected graph

Without loss of generality, assume it is connected (i.e. one big connected component)

Find an edge, so that if you remove that edge the graph is no longer connected

Naive algorithm: Try removing edges, one at a time, and finding the connected components of the resulting graph

That’s pretty inefficient, $O(m(n + m))$
Let’s take a look at the values that we computed in the DFS tree

We see that a forward edge \((u, v)\) is a bridge if and only if \(\text{low}[v] > \text{num}[u]\)

Simple to extend our analyze function to return all bridges

Again, this is just \(O(n + m)\)
const int n = 1000;
vector<int> adj[n];
vector<int> low(n), num(n, -1);
int curnum = 0;

vector<pair<int, int>> bridges;

void find_bridges(int u, int p) {
    low[u] = num[u] = curnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (v == p) continue;
        if (num[v] == -1) {
            find_bridges(v, u);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], num[v]);
        }
        if (low[v] > num[u]) {
            bridges.push_back(make_pair(u, v));
        }
    }
}

for (int u = 0; u < n; u++) {
    if (num[u] == -1) {
        find_bridges(u, -1);
    }
}
Strongly connected components

- We know how to find connected components in undirected graphs
- But what about directed graphs?

- Such components behave a bit differently in directed graphs, especially since if $v$ is reachable from $u$, it doesn’t mean that $u$ is reachable from $v$

- The definition remains the same, though
- A strongly connected component is a maximal subset of the vertices such that each pair of vertices is reachable from each other
The connected components algorithm won’t work here.

Instead we can use the depth-first search tree of the graph to find these components.

see example
Strongly connected components

```cpp
vector<int> adj[100];
vector<int> low(100), num(100, -1);
vector<bool> incomp(100, false);
int curnum = 0;

stack<int> comp;

void scc(int u) {
    // scc code...
}

for (int i = 0; i < n; i++) {
    if (num[i] == -1) {
        scc(i);
    }
}
```
void scc(int u) {
    comp.push(u);
    incompt[u] = "true";

    low[u] = num[u] = currnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (num[v] == -1) {
            scc(v);
            low[u] = min(low[u], low[v]);
        } else if (incompt[v]) {
            low[u] = min(low[u], num[v]);
        }
    }

    if (num[u] == low[u]) {
        printf("comp: ");
        while (true) {
            int cur = comp.top();
            comp.pop();
            incompt[cur] = false;
            printf("%d, ", cur);
            if (cur == u) {
                break;
            }
        }
        printf("\n");
    }
}

Strongly connected components
Strongly connected components

- Time complexity?
- Basically just the DFS analyze function (which was $O(n + m)$), with one additional loop to construct the component
- But each vertex is only in one component...
- Time complexity still just $O(n + m)$
Example problem: Come and Go

▶ http://uva.onlinejudge.org/external/118/11838.html
Topological sort

- We have $n$ tasks
- Each task $i$ has a list of tasks that must be finished before we can start task $i$
- Find an order in which we can process the tasks
- Can be represented as a directed graph
  - Each task is a vertex in the graph
  - If task $j$ should be finished before task $i$, then we add a directed edge from vertex $i$ to vertex $j$

- Notice that this can’t be solved if the graph contains a cycle
- A modified depth-first search can be used to find an ordering in $O(n + m)$ time, or determine that one does not exist
vector<int> adj[1000];
vector<bool> visited(1000, false);
vector<int> order;

void topsort(int u) {
    if (visited[u]) {
        return;
    }

    visited[u] = true;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        topsort(v);
    }

    order.push_back(u);
}

for (int u = 0; u < n; u++) {
    topsort(u);
}
Example problem: Ordering Tasks

▶ http://uva.onlinejudge.org/external/103/10305.html
Breadth-first search

- There’s another search algorithm called Breadth-first search.
- Only difference is the order in which it visits the vertices.
- It goes in order of increasing distance from the source vertex.
Breadth-first search

Queue:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Breadth-first search

Queue: 0

marked | 0 1 2 3 4 5 6 7 8 9 10
-------|---------------------
        | 0 1 0 0 0 0 0 0 0 0 0
Breadth-first search

Queue: 0

marked 1 0 0 0 0 0 0 0 0 0 0 0
Breadth-first search

Queue: 0 1 2

marked: 1 1 1 0 0 0 0 0 0 0 0 0
Breadth-first search

Queue: 1 2

marked | 1 1 1 0 0 0 0 0 0 0 0 0
Breadth-first search

Queue: 1 2

marked | 1 1 1 0 0 0 0 0 0 0 0

0 1 2 3 4 5 6 7 8 9 10
Breadth-first search

Queue: 1 2 4

marked 1 1 1 0 1 0 0 0 0 0 0
Breadth-first search

Queue: 2 4

marked | 1 1 1 0 1 0 0 0 0 0 0 0 0
Breadth-first search

Queue: 2 4

marked: 1 1 1 0 1 0 0 0 0 0 0
Breadth-first search

Queue: 2 4 3

marked: 1 1 1 1 1 0 0 0 0 0 0
Breadth-first search

Queue: 4 3

marked | 1 1 1 1 1 1 0 0 0 0 0 0
Breadth-first search

Queue: 4 3

marked

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Breadth-first search

Queue: 4 3 5

marked: 1 1 1 1 1 1 1 0 0 0 0 0
Breadth-first search

Queue: 3 5

marked: 1 1 1 1 1 1 1 0 0 0 0 0
Breadth-first search

Queue: 5

marked: 1 1 1 1 1 1 1 0 0 0 0 0
Breadth-first search

Queue: 5

marked: 1 1 1 1 1 1 1 0 0 0 0 0
Breadth-first search

Queue: 5 6 7 8

marked | 1 1 1 1 1 1 1 1 1 0 0

0 1 2 3 4 5 6 7 8 9 10
Breadth-first search

Queue: 6 7 8

marked: 1 1 1 1 1 1 1 1 1 1 0 0
Breadth-first search

Queue: 7 8

marked: 1 1 1 1 1 1 1 1 1 0 0
Breadth-first search

Queue: 7 8

marked | 0 1 2 3 4 5 6 7 8 9 10
---|---
    | 1 1 1 1 1 1 1 1 1 1 0 0
Breadth-first search

Queue: 7 8 10

marked 1 1 1 1 1 1 1 1 1 1 0 1
Breadth-first search

Queue: 8 10

marked | 1 1 1 1 1 1 1 1 1 1 0 1
Breadth-first search

Queue: 8 10

marked: 1 1 1 1 1 1 1 1 1 0 1
Breadth-first search

Queue: 8 10 9

marked | 1 1 1 1 1 1 1 1 1 1 1 1
Breadth-first search

Queue: 10 9

marked | 1 1 1 1 1 1 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10
Breadth-first search

Queue: 9

marked | 0 1 2 3 4 5 6 7 8 9 10
-------|-----------------------
        | 1 1 1 1 1 1 1 1 1 1 1
Breadth-first search

Queue:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Breadth-first search

```cpp
vector<int> adj[1000];
vector<bool> visited(1000, false);

queue<int> Q;
Q.push(start);
visited[start] = true;

while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (!visited[v]) {
            Q.push(v);
            visited[v] = true;
        }
    }
}
```
Shortest path in unweighted graphs

- We have an unweighted graph, and want to find the shortest path from $A$ to $B$
- That is, we want to find a path from $A$ to $B$ with the minimum number of edges
- Breadth-first search goes through the vertices in increasing order of distance from the start vertex
- Just do a single breadth-first search from $A$, until we find $B$
- Or let the search continue through the whole graph, and then we have the shortest paths from $A$ to all other vertices
- Shortest path from $A$ to all other vertices: $O(n + m)$
Shortest path in unweighted graphs

vector<int> adj[1000];
vector<bool> dist(1000, -1);

queue<int> Q;
Q.push(A);
dist[A] = 0;

while (!Q.empty()) {
    int u = Q.front(); Q.pop();

    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (dist[v] == -1) {
            Q.push(v);
            dist[v] = 1 + dist[u];
        }
    }
}

printf("%d\n", dist[B]);