CSI - 3105 Design & Analysis of Algorithms
Course 5

Jean-Lou De Carufel

Fall 2019

Example

Exercise #10 (Chapter 1)

Write an algorithm that finds the m smallest numbers in a list of n numbers (where $1 \leq m \leq n$) and analyze its running time in the worst case.

Example

Exercise #10 (Chapter 1)

Write an algorithm that finds the m smallest numbers in a list of n numbers (where $1 \leq m \leq n$) and analyze its running time in the worst case.

What do you think of the following solution?

Sort the list of numbers using Merge Sort. Then scan the list and return the first m numbers.

Sorting using Merge Sort takes $O(n \log(n))$ time and scanning the list takes $O(n)$ time. So in total, this algorithm takes $O(n \log(n)) + O(n) = O(n \log(n))$ time.

Example

A Faster Solution

What do you think of the following solution?

Let L be the list of numbers. Find the m-th smallest element of L using the Select Algorithm with $Select(L, m)$. Let this m-th smallest element be x. Scan the list and return all numbers that are $\leq x$.

A call to the Select Algorithm takes $O(n)$ time and scanning the list takes $O(n)$ time. So in total, this algorithm takes $O(n) + O(n) = O(n)$ time.

Example

A Faster Solution

What do you think of the following solution?

Let L be the list of numbers. Find the m-th smallest element of L using the Select Algorithm with $Select(L, m)$. Let this m-th smallest element be x. Scan the list and return all numbers that are $\leq x$.

A call to the Select Algorithm takes $O(n)$ time and scanning the list takes $O(n)$ time. So in total, this algorithm takes $O(n) + O(n) = O(n)$ time.

Nothing in the question says that the numbers are all different. So we cannot assume that they are!

Example

A Faster Solution

What do you think of the following solution?

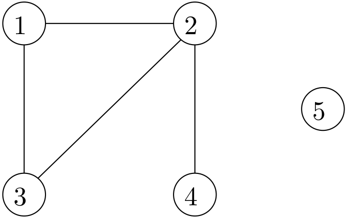
Let L be the list of numbers. Find the m-th smallest element of L using the Select Algorithm with $Select(L, m)$. Let this m-th smallest element be x. Initialize a counter cnt to 0. Scan the list and return all numbers that are $< x$. Every time you return a number, increment cnt by 1. When you are done scanning the list, return the number x exactly $m - cnt$ times.

A call to the Select Algorithm takes $O(n)$ time and scanning the list takes $O(n)$ time. Returning the number x exactly $m - cnt$ times takes $O(n)$ time since $m - cnt \leq n$. So in total, this algorithm takes $O(n) + O(n) + O(n) = O(n)$ time.

Chapter 3: Graph Algorithms

A *graph* G is made of a set V of *vertices* (or *nodes*) together with a set E of edges. We write $G = (V, E)$.

A graph is *undirected* if each edge in E is a pair $\{u, v\}$, where $u, v \in V$ and $u \neq v$.



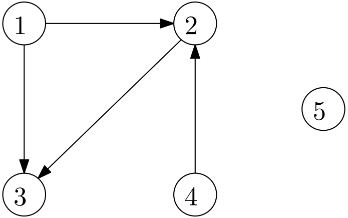
$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}\}$$

Chapter 3: Graph Algorithms

A *graph* G is made of a set V of *vertices* (or *nodes*) together with a set E of edges. We write $G = (V, E)$.

A graph is *directed* is each edge in E is an **ordered** pair (u, v), where $u, v \in V$ and $u \neq v$.



$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1, 2), (1, 3), (2, 3), (4, 2)\}$$

Examples:

- A road map
- Facebook. Vertices are users. There is an edge $\{A, B\}$ if and only if A and B are "friends".
- WWW. Vertices are web pages. There is a **directed** edge (A, B) if and only if A has a link to B.

Examples:

- Scheduling exams!

$$V = \text{set of all courses taught this term}$$

There is an edge $\{u, v\}$ if and only if there is at least one student taking both courses u and v.

Let C be the minimum number of colors needed such that
  - each vertex gets one color.
  - For each edge $\{u, v\}$, u and v have different colors.

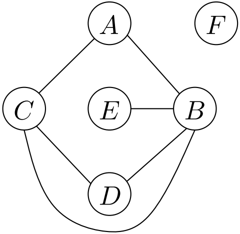
Then we can make an exam schedule with C time slots $1, 2, ..., C$: all vertices (i.e., courses) with color i have their exam in time slot i. In this way, there are no conflicts!

But computing C is very difficult...

In a graph $G = (V, E)$, two vertices $u, v \in V$ are *adjacent* if there is an edge between u and v.

A vertex $u \in V$ is *incident* to an edge $e \in E$ if one of the two vertices of e is u.

The *degree* of a vertex $u \in V$ is equal to the number of edges incident to u.



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

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

$\deg(B) = 4$

$\deg(E) = 1$

$\deg(F) = 0$

When the graph is oriented, the *outdegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the starting point of e. The *indegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the endpoint of e.

When the graph is oriented, the *outdegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the starting point of e. The *indegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the endpoint of e.

Theorem (Handshaking Lemma)

Let $G = (V, E)$ be a graph. then

$$\sum_{u \in V} \deg(u) = 2|E|.$$

PROOF: $\square$

When the graph is oriented, the *outdegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the starting point of e. The *indegree* of a vertex $u \in V$ is equal to the number of edges $e \in E$ such that u is the endpoint of e.

Theorem (Handshaking Lemma)

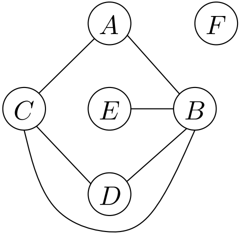
Let $G = (V, E)$ be a graph. then

$$\sum_{u \in V} \deg(u) = 2|E|.$$

PROOF: Each edge is counted twice! $\square$

How to store a graph?

$$G = (V, E) \qquad V = \{v_1, v_2, \ldots, v_n\}$$

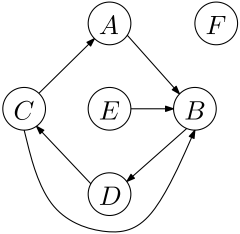


$$\begin{array}{c|c} & \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix} & \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency matrix

How to store a graph?

$$G = (V, E) \qquad V = \{v_1, v_2, \ldots, v_n\}$$

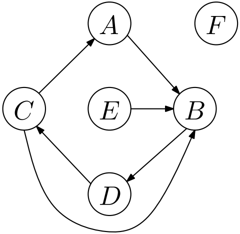


$$\begin{array}{c c} & \begin{matrix} A & B & C & D & E & F \end{matrix} \\ \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Adjacency matrix

How to store a graph?

$$G = (V, E) \qquad V = \{v_1, v_2, \ldots, v_n\}$$



$$\begin{array}{c|c} & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

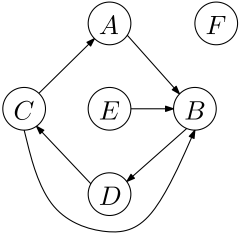
Adjacency matrix

Advantage:

- In $O(1)$ time, we can test if there is an edge between two given vertices.

How to store a graph?

$$G = (V, E) \qquad\qquad V = \{v_1, v_2, \ldots, v_n\}$$



$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 0 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 0 & 0 \\ C & 1 & 1 & 0 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & 0 & 0 & 0 \\ E & 0 & 1 & 0 & 0 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Adjacency matrix

Advantage:

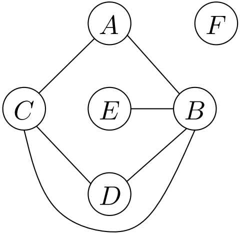
- In $O(1)$ time, we can test if there is an edge between two given vertices.

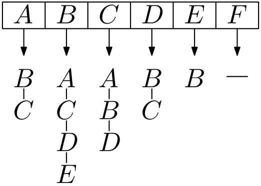
Disadvantage:

- Uses $\Theta(n^2)$ space for any graph.
- Finding all neighbours of a given vertex takes $O(n)$ time.

How to store a graph?

$G = (V, E)$ $\qquad$ $V = \{v_1, v_2, ..., v_n\}$

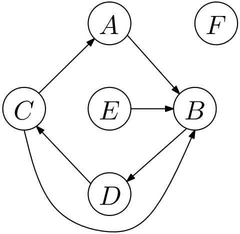


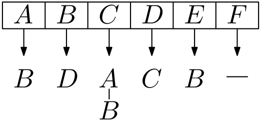


Adjacency list

How to store a graph?

$G = (V, E)$ $\qquad$ $V = \{v_1, v_2, \dots, v_n\}$

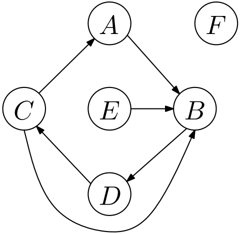


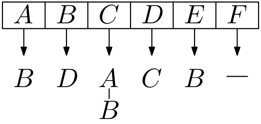


Adjacency list

How to store a graph?

$G = (V, E)$ $\qquad$ $V = \{v_1, v_2, \ldots, v_n\}$





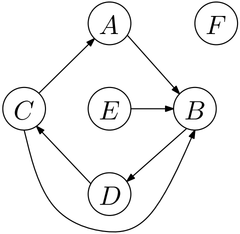
Adjacency list

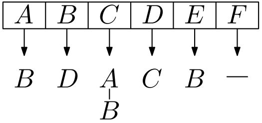
Advantage:

- Uses $\Theta(|V| + |E|)$ space.
- Finding all neighbours of a vertex $u \in V$ takes $O(1 + \deg(u))$ time.

How to store a graph?

$G = (V, E)$ $\qquad$ $V = \{v_1, v_2, \ldots, v_n\}$





Adjacency list

Advantage:

- Uses $\Theta(|V| + |E|)$ space.
- Finding all neighbours of a vertex $u \in V$ takes $O(1 + \deg(u))$ time.

Disadvantage:

- Testing if $\{u, v\}$ (or (u, v)) is an edge takes $O(1 + \deg(u))$ time.

Section 3.1: Exploring an Undirected Graph

Let $G = (V, E)$ be an undirected graph.

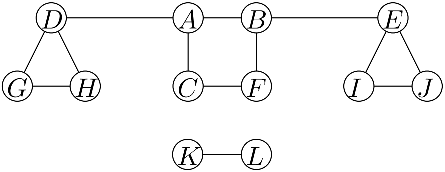
Task: Find all vertices that can be reached from a given vertex $v \in V$.

Algorithm *explore*(v)

```
visited(v) = TRUE
previsit(v)                          // See later
for each edge {u, v} ∈ E do
    if visited(u) = FALSE then
        call explore(u)
    end if
end for
postvisit(v)                         // See later
```

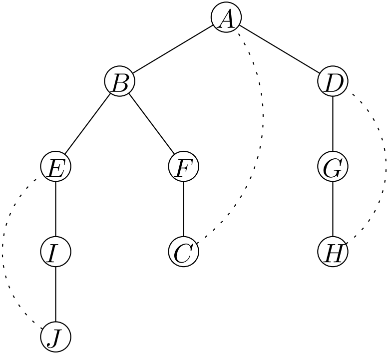
Algorithm *explore*(v)

```
visited(v) = TRUE
previsit(v)                                   // See later
for each edge {u, v} ∈ E do
    if visited(u) = FALSE then
        call explore(u)
    end if
end for
postvisit(v)                                  // See later
```



Run *explore*(A). In the for-loop, use alphabetical order (i.e., adjacency lists are sorted alphabetically). Each time an edge $\{u, v\}$ is traversed (because $visited(u) = \text{FALSE}$): u is discovered for the first time.

- Draw $\{u, v\}$ as a solid edge.
- All other edges: dotted.



The solid edges form a *tree* (connected, no cycle). These edges are called *tree edges*. The dotted edges are called *back edges*.

Why is algorithm *explore*(v) correct?

First, how can we explain that it always terminates?

Algorithm *explore*(v)

```
visited(v) = TRUE
previsit(v)                         // See later
for each edge {u, v} ∈ E do
    if visited(u) = FALSE then
        call explore(u)
    end if
end for
postvisit(v)                        // See later
```

Why is algorithm *explore*(v) correct?

First, how can we explain that it always terminates?

The number of vertices u such that "$\mathit{visited}(u) = \text{FALSE}$" decreases in each recursive call. Since there is a finite number of vertices, the algorithm eventually terminates.

Algorithm *explore*(v)

```
visited(v) = TRUE
previsit(v)                              // See later
for each edge {u, v} ∈ E do
    if visited(u) = FALSE then
        call explore(u)
    end if
end for
postvisit(v)                             // See later
```

How can we explain that it does visit all vertices that are reachable from v?

Algorithm *explore*(v)

1: *visited*$(v) = \text{TRUE}$
2: *previsit*(v) // See later
3: **for** each edge $\{u, v\} \in E$ **do**
4:     **if** *visited*$(u) = \text{FALSE}$ **then**
5:         call *explore*(u)
6:     **end if**
7: **end for**
8: *postvisit*(v) // See later

How can we explain that it does visit all vertices that are reachable from v?

Lemma

Assume that, initially, $\mathit{visited}(u) = \text{FALSE}$. *After* $\mathit{explore}(v)$ *has terminated,*

$$\mathit{visited}(u) = \text{TRUE}$$

$$\Longleftrightarrow$$

there is a path from v *to* u.

Algorithm *explore*(v)

```
visited(v) = TRUE
previsit(v)                                  // See later
for each edge {u, v} ∈ E do
    if visited(u) = FALSE then
        call explore(u)
    end if
end for
postvisit(v)                                 // See later
```

Solid edges form a tree
(connected, no cycles)
These edges are called: <u>tree edges</u>

Dotted edges: <u>back edges</u>

Why is algorithm explore(v) correct?

Why does it terminate: number of vertices u with visited(u) = false decreases in each recursive call.

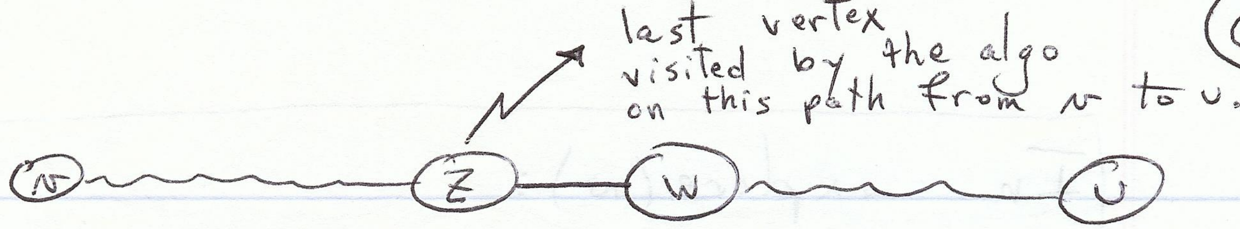
Assume that, initially, visited(u) = false.

<u>Claim:</u> After explore(v) has terminated:

visited(u) = true $\Leftrightarrow$ there is a path from v to u.

<u>proof:</u> [$\Rightarrow$] Follows from the algorithm: the algorithm "walks" from a vertex to a neighboring vertex

[$\Leftarrow$] By contradiction: Assume there is a path from v to u, and assume that, after termination, visited(u) = false. Consider any path from v to u:



So z was visited, but w was not. This is a contradiction. When visiting z, the algorithm notices that visited (w) = false and then visits w.

$\square$

<u>Connected components</u> of $G = (V, E)$:

number the connected components as $1, 2, 3, \ldots$

for each vertex v: ccnumber(v) = number of the connected component that v belongs to.

Algo DFS(G): // depth-first search

```
for all v ∈ V:
        visited(v) = false
cc = 0
for all v ∈ V:
        if visited(v) = false
              cc = cc+1
              explore(v)
```