Advanced Data Structures

Lecture 1: Introduction. Structures for graphs

Mircea Marin

September 30 2014

Organizatorial items

- Lecturer and TA: Mircea Marin
  - email: `mmarin@info.uvt.ro`
- Course objectives:
  1. Become familiar with some of the advanced data structures (ADS) and algorithms which underpin much of today's computer programming
  2. Recognize the data structure and algorithms that are best suited to model and solve your problem
  3. Be able to perform a qualitative analysis of your algorithms – time complexity analysis
- Course webpage:
  http://web.info.uvt.ro/˜mmarin/lectures/ADS
- Handouts: will be posted on the webpage of the lecture
- Grading: 40% final exam (written), 60% labwork (mini-projects)
- Attendance: required

Organizatorial items

- Lab work: implementation in C++ or Java of applications, using data structures and algorithms presented in this lecture
- Requirements:
  - ▷ Be up-to-date with the presented material
  - ▷ Prepare the programming assignments for the stated deadlines
- Recommended textbooks:
  - Cormen, Leiserson, Rivest. *Introduction to Algorithms*. MIT Press.
  - Adam Drozdek. *Data structures and algorithms in* C++*.* Third edition, Thomson Course Technology, 2005.
  - Aho, Hopcroft, Ullmann. *Data structures and algorithms*, Addison-Wesley, 1985.

Introductory remarks

- The problem-to-program process.
- Role of abstract data types in this process.
- "big-oh" and "big-omega" notation.

Design and Analysis of Algorithms

From problems to programs

1. Problem specification: make it precise.
   - Choose a good formal model for your problem.

Problem	**Possible model**
Numerical problem	linear equations; differential eqs.
Symbol and text processing	character strings; formal grammars.
Search problem	Ordered lists; binary trees.

2. Find a solution using the selected model.
   - Algorithm = precise description of your solution.

Design and Analysis of Algorithms

From problems to programs

1. Problem specification: make it precise.
   - Choose a good formal model for your problem.

Problem	**Possible model**
Numerical problem	linear equations; differential eqs.
Symbol and text processing	character strings; formal grammars.
Search problem	Ordered lists; binary trees.

2. Find a solution using the selected model.
   - Algorithm = precise description of your solution.
   - Algorithm = finite sequence of clear instructions, which can be performed with finite effort in a finite length of time.

Design and Analysis of Algorithms
From problems to programs

1. Problem specification: make it precise.
   - Choose a good formal model for your problem.

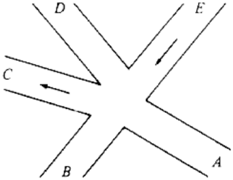
Problem	**Possible model**
Numerical problem	linear equations; differential eqs.
Symbol and text processing	character strings; formal grammars.
Search problem	Ordered lists; binary trees.

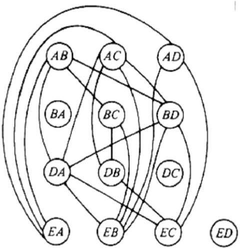
2. Find a solution using the selected model.
   - Algorithm = precise description of your solution.
   - Algorithm = finite sequence of clear instructions, which can be performed with finite effort in a finite length of time.

$$\boxed{\text{Informal algorithm (pseudo-code)}} \xrightarrow{\text{stepwise refinement}} \boxed{\text{Formal algorithm (programming language)}}$$

From problems to programs

Example: traffic light for road intersection





- Problem specification:
  - intersection of 5 roads: A, B, C, D, E; Roads C and E are oneway; There are 13 possible turns: AB, EC, etc.
  - Design a light intersection system to avoid car collisions.
- Chosen model: a graph with
  - ▷ nodes = possible turns: $\{AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED\}$
  - ▷ edges are between turns that can be performed simultaneously.

Example (continued)

- **Graph coloring**: assignment of a color to each node so that no two vertices connected by an edge have the same color.
- REMARK: Our problem is the same as coloring the graph of incompatible turns using as few colors as possible.

Advantages of the chosen model:

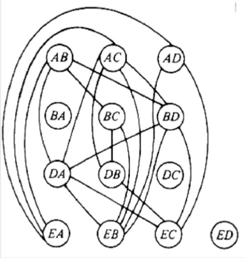
- Graph coloring is a well-studied problem.
  - **Bad news:** the problem is NP complete: to find the best solution, we must essentially try all possibilities :-(
    $\Rightarrow$ finding optimal solution may be very inefficient.
  - **Alternative:** be happy with a reasonably good solution, using a heuristic algorithm. E.g., use the following "greedy" algorithm.

Example (continued)

The "greedy" algorithm - informal description

1. Select some uncolored node and color it with a new color.
2. Scan the list of uncolored nodes. For each node, determine whether it has an edge to any vertex already colored with the new color. If there is no such edge, color the present vertex with the new color.

Example (Optimal coloring)



color	turns	extras
blue	*AB*, *AC*, *AD*, *BA*, *DC*, *ED*	-
red	*BC*, *BD*, *EA*	*BA*, *DC*, *ED*
green	*DA*, *DB*	*AD*, *BA*, *DC*, *ED*
yellow	*EB*, *EC*	*BA*, *DC*, *EA*, *ED*

Remarks about the greedy algorithm

- does not always use the minimum possible number of colors.
  - We can use the theory of algorithms to evaluate the goodness of the solution produced.
  - *k*-clique = set of *k* nodes, every pair of which is connected by an edge.
  - REMARK 1: We need k colors to color a k-clique.
  - REMARK 2: The illustrated graph of incompatible turns contains a 4-clique AC, DA, BD, $EB \Rightarrow \geq 4$ colors are needed.

Pseudo-Language and Stepwise Refinements

Illustrated algorithm: greedy

- First refinement

```
        procedure greedy ( var G: GRAPH; var newclr: SET );
          { greedy assigns to newclr a set of vertices of G that may be
              given the same color }
          begin
(1)           newclr := Ø; †
(2)           for each uncolored vertex v of G do
(3)               if v is not adjacent to any vertex in newclr then begin
(4)                   mark v colored;
(5)                   add v to newclr
                  end
          end; { greedy }
```

Pseudo-Language and Stepwise Refinements

Illustrated algorithm: greedy

- Second refinement

```
          procedure greedy ( var G: GRAPH; var newclr: SET );
            begin
(1)           newclr := Ø;
(2)           for each uncolored vertex v of G do begin
(3.1)            found := false;
(3.2)            for each vertex w in newclr do
(3.3)               if there is an edge between v and w in G then
(3.4)                  found := true;
(3.5)            if found = false then begin
                  { v is adjacent to no vertex in newclr }
(4)                 mark v colored;
(5)                 add v to newclr
                 end
              end
            end; { greedy }
```

Representations of graphs
Adjacency matrix, etc.

Adjacency matrix

	AB	AC	AD	BA	BC	BD	DA	DB	DC	EA	EB	EC	ED
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										1	1	1	
BA													
BC	1							1			1		
BD	1	1					1				1	1	
DA	1	1				1					1	1	
DB		1			1							1	
DC													
EA	1	1	1										
EB		1	1		1	1	1						
EC			1			1	1	1					
ED													

Abstract Data Types (ADT)

ADT = mathematical model with a collection of operations defined on that model.

- ADT = generalization of primitive data types (integer, float, char, boolean, etc.)
- In OOP (e.g., C++), they can be implemented as classes.
  - Benefits: encapsulation, generalization (by class extension), overloading, ...

ADTs for the greedy algorithm

- For colors: List
  Desirable operations:
  - init(colorList) - initialize the list of colors to be empty.
  - first(colorList) - return the first element of the list, and null if the list is empty.
  - next(colorList) - get the next element of the list, and null if there is no next member
  - insert(c, colorList) - insert a new color c into the colorList
- For colored graphs: Graph
  Desirable operations:
  - get the first uncolored node,
  - test whether there is an edge between two nodes,
  - mark a node colored,
  - get the next uncolored node,
  - ...

Running time of a program

Factors on which it depends:

- Size of the input.
- Quality of compiler-generated code.
- Nature and speed of computer instructions.
- Time complexity of the underlying algorithm.

The running time of a program should be defined as a function $T(n)$ of the size n of input.

The Big-Oh, Big-Omega, and Big-Theta notation

Assumptions:

- $T(n)$: running time function, depending on input size $n \in \mathbb{N}$.

▷ "Big-Oh" notation: $T(n)$ is $O(f(n))$ if there are $c \in \mathbb{R}$, $c > 0$, and $n_0 \in \mathbb{N}$ such that $T(n) \leq c\, f(n)$ for all $n \geq n_0$.

▷ "Big-Omega" notation: $T(n)$ is $\Omega(f(n))$ if there are $c \in \mathbb{R}$, $c > 0$ such that $T(n) \geq c\, f(n)$ for infinitely many values of n.

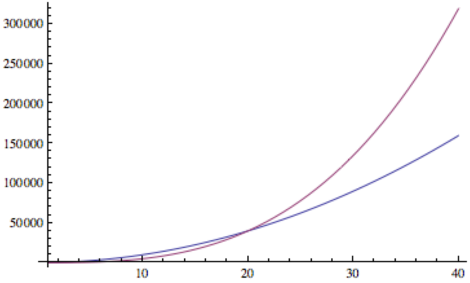
▷ "Big -Theta" notation: $T(n)$ is $\Theta(f(n))$ if there are $c_1, c_2 \in \mathbb{R}$, $c_1, c_2 > 0$, and $n_0 \in \mathbb{N}$ such that $c_1\, f(n) \leq T(n) \leq c_2\, f(n)$ for all $n \geq n_0$.

Desirable running times

- Polynomial running time: $T(n)$ is $O(n^k)$ for some $k > 0$.
  - Algorithms with polynomial running time are called tractable.
- **Question:** Algorithm A_1 has running time $5\,n^3$ and A_2 has running time $100\,n^2$. Which alg. has better running time?

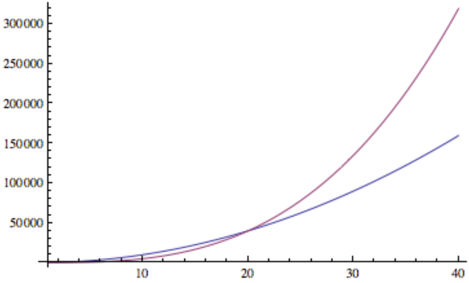
Desirable running times

- Polynomial running time: $T(n)$ is $O(n^k)$ for some $k > 0$.
  - Algorithms with polynomial running time are called tractable.
- **Question:** Algorithm A_1 has running time $5\,n^3$ and A_2 has running time $100\,n^2$. Which alg. has better running time?



Desirable running times

- Polynomial running time: $T(n)$ is $O(n^k)$ for some $k > 0$.
  - Algorithms with polynomial running time are called tractable.
- **Question:** Algorithm A_1 has running time $5\,n^3$ and A_2 has running time $100\,n^2$. Which alg. has better running time?



Answer: A_1 if $n < 20$; A_2 otherwise.

Graphs
Types of graphs

Graph: data structure (V, E) consisting of

- a finite set of nodes $V = \{v_1, \ldots, v_n\}$,
- a finite set of edges $E = \{e_1, \ldots, e_m\}$ between nodes.

1: Graphs are directed or undirected:

Directed: Every edge $e \in E$ has a *source* (or *initial*) node $u \in V$ and a *destination* (or *end*) node $v \in V$: $endpoints(e) = (u, v)$

Undirected: Every edge $e \in E$ has two endpoints $u, v \in V$: $endpoints(e) = \{u, v\}$

2: Graphs are simple or multiple:

Simple: at most one edge between two endpoints: $endpoints(e) = endpoints(e')$ implies $e = e'$.

Multiple: Can have more edges between two endpoints.

Terminology

- The edges of a directed graph are called arcs.
- A directed graph is also known as a digraph.
- A graph is weighted if it also has an additional function $w : E \to \mathbb{R}$ which associates a weight $w(e)$ to every $e \in E$.
- v is adjacent to u if there exists $e \in E$ such that
  - $endpoints(e) = (u, v)$ in directed graphs.
  - $endpoints(e) = \{u, v\}$ in undirected graphs.
- A loop is an edge e whose endpoints coincide:
  - $endpoints(e) = (u, u)$ if the graph is directed.
  - $endpoints(e) = \{u\}$ if the graph is undirected.

Classification of graphs

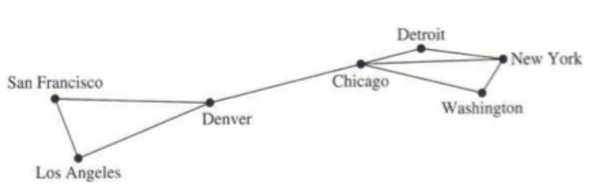
Type	Edges	Multiple edges?	Loops?
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Directed graph	Directed	No	Yes
Directed multigraph	Directed	Yes	Yes

Examples
Simple graph

(V, E) where

- E can be considered to be a set of unordered pairs of elements of V (the edges).

Example (Computer network)



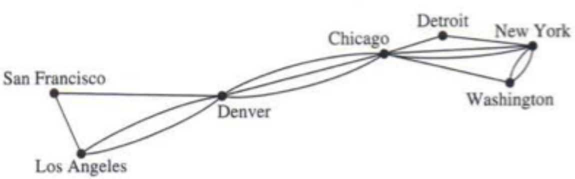
$V = \{\text{San Francisco, Denver, Chicago, Detroit, } \ldots\}$
$E = \{\{\text{Denver,Chicago}\}, \{\text{Detroit, New York}\}, \ldots\}$

Examples
Multigraph

(V, E) together with a function
$$endpoints : E \to \{\{u, v\} \mid u, v \in V, u \neq v\}.$$

- The edges $e_1, e_2 \in E$ are multiple (or parallel) if $f(e_1) = f(e_2)$.
- INTUITION: $f(e) = \{u, v\}$ tells us that e is an edge between nodes u and v. NOTE: $\{u, v\} = \{v, u\}$
- MAIN DIFFERENCE FROM SIMPLE GRAPH: there can be more edges between two nodes.

Example (computer network with multiple lines)



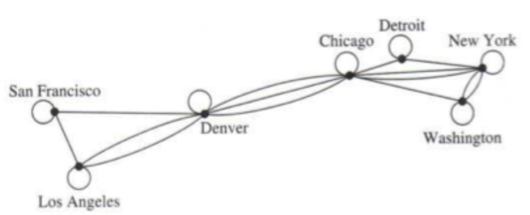
Examples
Pseudograph

(V, E), together with a function

$$endpoints : E \to \{\{u, v\} \mid u, v \in V\}$$

- MAIN DIFFERENCE FROM MULTIGRAPH: we can have $endpoints(e) = \{u, v\}$ with $u = v$, that is, an edge from a node to itself.

Example (Network of phone lines, where there are lines from a phone to itself)



Examples
Directed graph (digraph)

(V, E) where

- E can be considered to be a set of ordered pairs of nodes.

Example (Communication network with one-way telephone lines)

$V = \{\text{San Francisco, Denver, Chicago, Detroit, } \ldots\}$
$E = \{(\text{Denver,Chicago}), (\text{Chicago,Denver}), (\text{Detroit, New York}), \ldots\}$

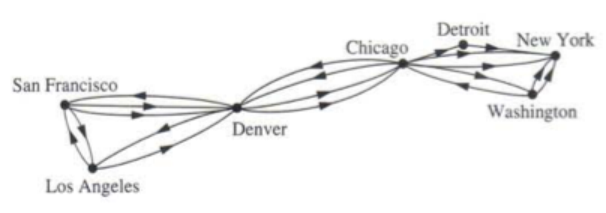
Examples
Directed multigraph

(V, E) together with a function

$$endpoints : E \to \{(u, v) \mid u, v \in V\}.$$

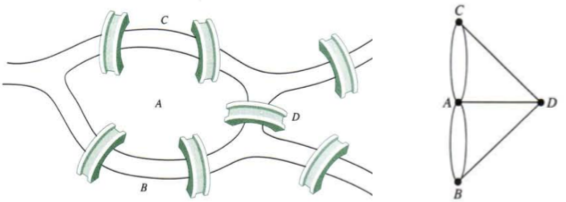
- The edges $e_1, e_2 \in E$ are multiple (or parallel) if $f(e_1) = f(e_2)$.

Example (communication network with multiple one-way telephone lines)



Graphs: Applications

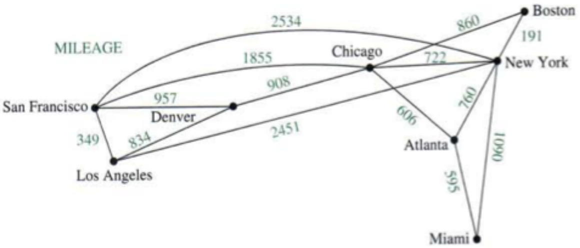
1. Can we walk down all streets (or bridges) of a city without going down a street (or bridge) twice?
   Classical problem: The Seven bridges of Königsberg



   Approach: Euler circuits
2. Can a circuit be implemented on a planar board? (connections should not intersect)
   Approach: planar graphs
3. (For chemists):Distinguish molecules with same formula but different structure

Graphs: Applications (continued)

4. Check network connectivity: are 2 computers connected by a communication link?
5. Find shortest paths between 2 cities in a transportation network



   Approach: weighted graphs

6. Schedule exams

Representations of graphs

1. adjacency list
2. list of edges
3. adjacency matrix
4. incidence matrix

   . . .

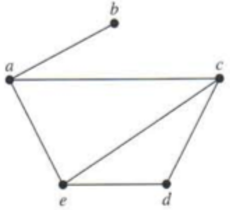
1. Adjacency lists

For every $v \in V$:

ADJACENCY-LIST(v):= list of all vertices adjacent to v.

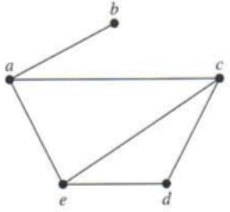
Can be used to represent graphs with no multiple edges. The represented graphs can be either simple or directed.

1. Adjacency lists
Examples

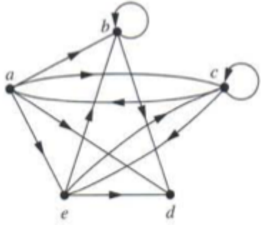
1. 

Vertex	Adjacency list
a	$[b, c, e]$
b	$[a]$
c	$[a, d, e]$
d	$[c, e]$
e	$[a, c, d]$

1. Adjacency lists
Examples

1. 

Vertex	Adjacency list
a	$[b, c, e]$
b	$[a]$
c	$[a, d, e]$
d	$[c, e]$
e	$[a, c, d]$

2. 

Vertex	Adjacency list
a	$[b, c, d, e]$
b	$[b, d]$
c	$[a, c, e]$
d	$[\,]$
e	$[b, c, d]$

2. List of edges

Suitable for the representation of graphs with few edges (the list is short)

Example (Undirected graph from previous slide)

$\textsc{Edge-List}(G) = [\{a, b\}, \{a, c\}, \{a, e\}, \{c, d\}, \{c, e\}, \{d, e\}]$

Example (Directed graph from previous slide)

$\textsc{Edge-List}(G) = \{(a, b), (a, c), (a, d), (a, e),$
$(b, b), (b, d), (c, a), (c, c), (c, e), \ldots\}$

3. Adjacency matrix (A_G)

If n = number of nodes of graph G then

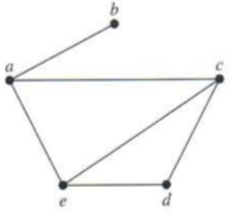
$$A_G = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \quad \text{is an } n \times n \text{ matrix}$$

where the value of $a_{i,j}$ depends on what kind of graph G is:

- For undirected graphs: $a_{i,j} =$ number of edges between node i and node j.
- For directed graphs: $a_{i,j} =$ number of edges from node i to node j.

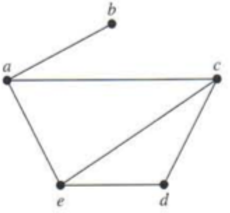
3. Adjacency matrix
Examples

- Example 1:



3. Adjacency matrix
Examples

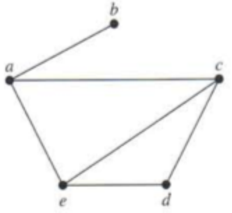
- Example 1: We must fix an enumeration of the vertices



	a	*b*	*c*	*d*	*e*
a					
b					
c					
d					
e					

3. Adjacency matrix
Examples

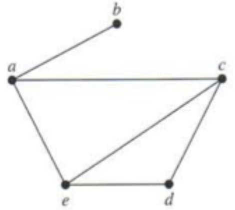
- Example 1: We must fix an enumeration of the vertices



$$\begin{array}{c c} & \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \\ e \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

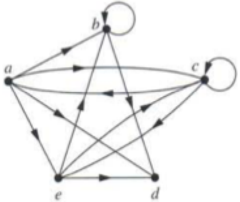
3. Adjacency matrix
Examples

- Example 1: We must fix an enumeration of the vertices



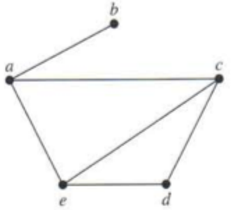
$$\begin{array}{c|ccccc} & a & b & c & d & e \\ \hline a & 0 & 1 & 1 & 0 & 1 \\ b & 1 & 0 & 0 & 0 & 0 \\ c & 1 & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 1 & 0 & 0 \\ e & 1 & 0 & 1 & 0 & 0 \end{array}$$

- Example 2:



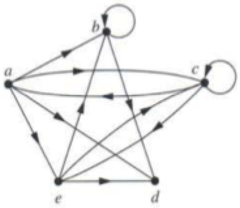
3. Adjacency matrix
Examples

- Example 1: We must fix an enumeration of the vertices



$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

- Example 2: We must fix an enumeration of the vertices



$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array}\begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

4. Incidence matrix (M_G)

Suitable for the representation of graphs G with no parallel edges.

4. Incidence matrix (M_G)

Suitable for the representation of graphs G with no parallel edges.

$$M_G = \begin{pmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,p} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n,1} & m_{n,2} & \cdots & m_{n,p} \end{pmatrix} \quad \text{is an } n \times p \text{ matrix}$$

where

n = number of nodes of graph G
p = number of edges of G
$m_{i,j}$ depends on what kind of graph G is:

4. Incidence matrix (M_G)

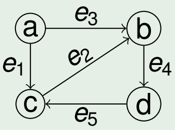
- For undirected graphs:

$$m_{i,j} = \begin{cases} 1 & \text{if } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise} \end{cases}$$

- For directed graphs:

$$m_{i,j} = \begin{cases} -1 & \text{if } v_i \text{ is the start node of } e_j \\ 1 & \text{if } v_i \text{ is the end node of } e_j \\ 0 & \text{otherwise} \end{cases}$$

Example



$M_G =$

	e_1	e_2	e_3	e_4	e_5
a	-1	0	-1	0	0
b	0	1	1	-1	0
c	1	-1	0	0	1
d	0	0	0	1	-1

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

> EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

> EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

> EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

A: A sequence $[e_1, \ldots, e_n]$ of n edges which satisfy the following condition:

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

> EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

A: A sequence $[e_1, \ldots, e_n]$ of n edges which satisfy the following condition:

In undirected graphs: $e_1 = \{x_0, x_1\}, \ldots, e_i = \{x_{i-1}, x_i\}, \ldots,$
$e_n = \{x_{n-1}, x_n\}$, and $x_0 = u$, $x_n = v$.

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

A: A sequence $[e_1, \ldots, e_n]$ of n edges which satisfy the following condition:

In undirected graphs: $e_1 = \{x_0, x_1\}, \ldots, e_i = \{x_{i-1}, x_i\}, \ldots,$
$e_n = \{x_{n-1}, x_n\}$, and $x_0 = u$, $x_n = v$.

In directed graphs: $e_1 = (x_0, x_1), \ldots, e_i = (x_{i-1}, x_i), \ldots,$
$e_n = (x_{n-1}, x_n)$, and $x_0 = u$, $x_n = v$.

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

A: A sequence $[e_1, \ldots, e_n]$ of n edges which satisfy the following condition:

In undirected graphs: $e_1 = \{x_0, x_1\}, \ldots, e_i = \{x_{i-1}, x_i\}, \ldots,$
$e_n = \{x_{n-1}, x_n\}$, and $x_0 = u$, $x_n = v$.

In directed graphs: $e_1 = (x_0, x_1), \ldots, e_i = (x_{i-1}, x_i), \ldots,$
$e_n = (x_{n-1}, x_n)$, and $x_0 = u$, $x_n = v$.

- It can also be described by the sequence of nodes it passes through: $[x_0, x_1, \ldots, x_n]$.

Connectivity

Many problems can be modelled with paths formed by traveling along the edges of graphs.

EXAMPLE. Message delivery — Does there exist a delivery path from the sender of a message to the intended receiver?

Q: What is a path of length n from node u to node v?

A: A sequence $[e_1, \ldots, e_n]$ of n edges which satisfy the following condition:

In undirected graphs: $e_1 = \{x_0, x_1\}, \ldots, e_i = \{x_{i-1}, x_i\}, \ldots,$
$e_n = \{x_{n-1}, x_n\}$, and $x_0 = u$, $x_n = v$.

In directed graphs: $e_1 = (x_0, x_1), \ldots, e_i = (x_{i-1}, x_i), \ldots,$
$e_n = (x_{n-1}, x_n)$, and $x_0 = u$, $x_n = v$.

- It can also be described by the sequence of nodes it passes through: $[x_0, x_1, \ldots, x_n]$.

A circuit is a path with $x_0 = x_n$.

Connectivity in digraphs

Assumption: $G = (V, E)$ is digraph with $V = \{1, 2, \ldots, n\}$

Given: two nodes $i, j \in V$

Determine: whether there exists a path from i to j.

Connectivity in digraphs

Assumption: $G = (V, E)$ is digraph with $V = \{1, 2, \ldots, n\}$

Given: two nodes $i, j \in V$

Determine: whether there exists a path from i to j.

Definition (Transitive closure)

The transitive closure of G is the graph $G^* = (V, E')$ such that $(v, v') \in E'$ if and only if there exists a path from v to v' in G.

Let A and A^* be the adjacency matrices of G and G^*.

- There is a path from i to j if and only if $A^*[i][j] = 1$
  $\Rightarrow$ it is desirable to compute A^*.

Connectivity in digraphs

Assumption: $G = (V, E)$ is digraph with $V = \{1, 2, \ldots, n\}$

Given: two nodes $i, j \in V$

Determine: whether there exists a path from i to j.

Definition (Transitive closure)

The transitive closure of G is the graph $G^* = (V, E')$ such that $(v, v') \in E'$ if and only if there exists a path from v to v' in G.

Let A and A^* be the adjacency matrices of G and G^*.

- There is a path from i to j if and only if $A^*[i][j] = 1$
  $\Rightarrow$ it is desirable to compute A^*.

How can we compute A^* if we know A?

Connectivity in digraphs
The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$
    ($\oplus$ is Boolean addition)
  - $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$.
    ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)

Connectivity in digraphs
The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$ ($\oplus$ is Boolean addition)
  - $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$. ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)
- There exists a path of length p from i to j iff the element at position (i, j) of the matrix $A^p = \underbrace{A \odot \ldots \odot A}_{p \text{ times}}$ is 1, where A is the adjacency matrix of G.

Connectivity in digraphs

The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$ ($\oplus$ is Boolean addition)
  - $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$. ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)
- There exists a path of length p from i to j iff the element at position (i, j) of the matrix $A^p = \underbrace{A \odot \ldots \odot A}_{p \text{ times}}$ is 1, where A is the adjacency matrix of G.
- There exists a path of length $\leq p$ from i to j iff the element at position (i, j) of the matrix $Id \oplus A \oplus \ldots \oplus A^{p+1}$ is 1, where A is the adjacency matrix of G, and Id is the identity matrix.

Connectivity in digraphs

The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$
    ($\oplus$ is Boolean addition)
  - $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$.
    ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)
- There exists a path of length p from i to j iff the element at position (i, j) of the matrix $A^p = \underbrace{A \odot \ldots \odot A}_{p \text{ times}}$ is 1, where A is the adjacency matrix of G.
- There exists a path of length $\leq p$ from i to j iff the element at position (i, j) of the matrix $Id \oplus A \oplus \ldots \oplus A^{p+1}$ is 1, where A is the adjacency matrix of G, and Id is the identity matrix.
- If there is a path from i to j in G then there is a path of length $\leq n - 1$ from i to j.

Connectivity in digraphs

The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$ ($\oplus$ is Boolean addition)
  - $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$. ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)
- There exists a path of length p from i to j iff the element at position (i, j) of the matrix $A^p = \underbrace{A \odot \ldots \odot A}_{p \text{ times}}$ is 1, where A is the adjacency matrix of G.
- There exists a path of length $\leq p$ from i to j iff the element at position (i, j) of the matrix $Id \oplus A \oplus \ldots \oplus A^{p+1}$ is 1, where A is the adjacency matrix of G, and Id is the identity matrix.
- If there is a path from i to j in G then there is a path of length $\leq n - 1$ from i to j.
- The transitive closure of adjacency matrix A is the matrix A^* where $A^*[i][j] = 1$ iff there exists a path of length ≥ 1 from i to j.

Connectivity in digraphs

The naive computation of the transitive closure

- If A, B are Boolean matrices of size $n \times n$, we can define:
  - ▷ $A \oplus B = C$ if $C[i][j] = A[i][j] \oplus B[i][j] = \max(A[i][j], B[i][j])$
    ($\oplus$ is Boolean addition)
  - ▷ $A \odot B = C$ if $C[i][j] = (A[i][1] \odot B[1][j]) \oplus \ldots \oplus (A[i][n] \odot B[n][j])$.
    ($\odot$ is the Boolean product defined by $b_1 \odot b_2 := \min(b_1, b_2)$.)
- There exists a path of length p from i to j iff the element at position (i, j) of the matrix $A^p = \underbrace{A \odot \ldots \odot A}_{p \text{ times}}$ is 1, where A is the adjacency matrix of G.
- There exists a path of length $\leq p$ from i to j iff the element at position (i, j) of the matrix $Id \oplus A \oplus \ldots \oplus A^{p+1}$ is 1, where A is the adjacency matrix of G, and Id is the identity matrix.
- If there is a path from i to j in G then there is a path of length $\leq n - 1$ from i to j.
- The transitive closure of adjacency matrix A is the matrix A^* where $A^*[i][j] = 1$ iff there exists a path of length ≥ 1 from i to j.

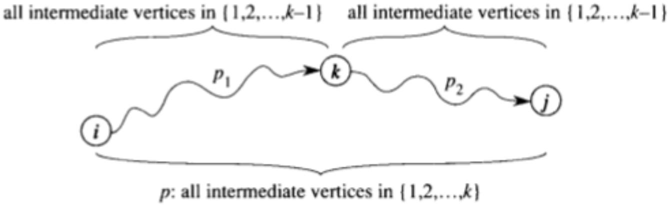
  $\Rightarrow$ $A^* = Id \oplus A \oplus \ldots \oplus A^{n-1}$ time compleity $O(n^4)$

Connectivity in digraphs
Transitive closure: a more efficient computation algorithm

Warshall discovered a much better approach:

- For all $i, j \in V = \{1, 2, \ldots, n\}$ consider all paths from i to j whose intermediate nodes are from $\{1, \ldots, k\}$. Let $C[i][j]^{(k)}$ be 1 if there exists such a path, and 0 otherwise.
- Warshall observed that $C[i][j]^{(k)}$ can be computed by recursion on k:

$$C[i][j]^{(k)} = \begin{cases} A[i][j] & \text{if } k = 0, \\ C[i][j]^{(k-1)} \oplus C[i][k]^{(k-1)} \odot C[k][j]^{(k-1)} & \text{if } k \geq 1. \end{cases}$$



- i, j are connected by a path iff $C[i][j]^{(n)} = 1$.

Transitive closure

Warshall's algorithm

```
procedure Warshall
  input:  int A[n][n] // an n x n Boolean matrix
  output: int C[n][n] // the transitive closure of A
for (int i:=0; i < n; i++)
  for (int j:=0;j<n;j++)
     C[i][j]=A[i][j];

for (int k:=0; k<n; k++)
  for (int i:=0; i<n; i++)
    for (int j:=0; j<n; j++)
       if (C[i][j] == 0)
         C[i][j] = min(C[i][k], C[k][j]);
```

- Note: In this implementation, the nodes are indexed from 0 to $n-1$
- Time complexity $O(n^3)$.