7.2 Isomorphic graphs & adjacency matrix

Notebook:	Discrete Mathematics [CM1020]
Created:	2019-10-07 2:31 PM
Updated:	2019-12-20 2:56 PM
Author:	SUKHJIT MANN

Cornell Notes	**Topic:** 7.2 Isomorphic graphs & adjacency matrix	Course: BSc Computer Science Class: Discrete Mathematics-Lecture Date: December 20, 2019

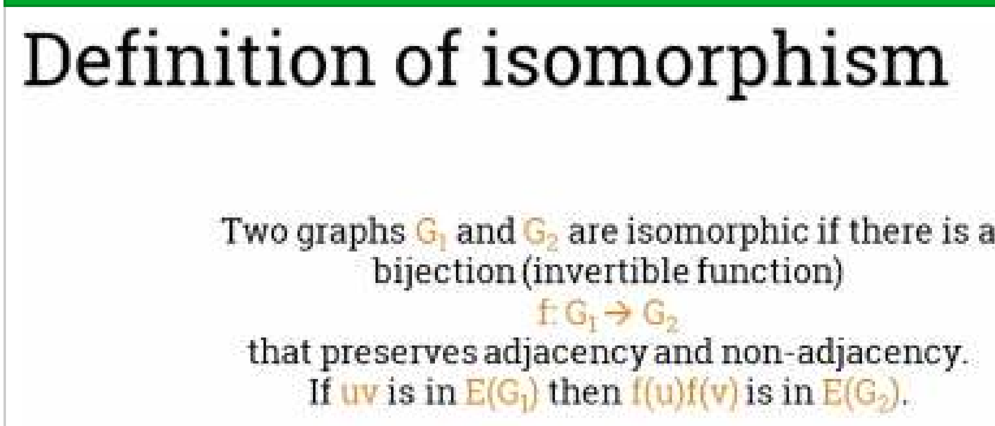
Essential Question:

What are isomorphic graphs and adjacency matrices?

Questions/Cues:

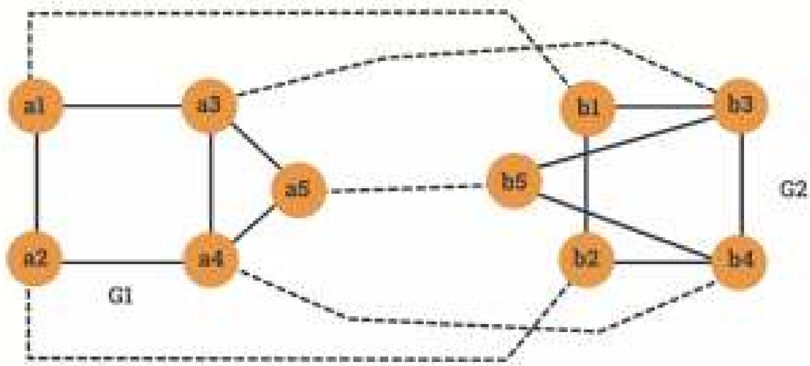
- What is Isomorphism?
- What are some properties of isomorphic graphs?
- What is a Bipartite Graph?
- What is Matching in terms of a Bipartite Graph?
- What is Maximum Matching?
- What is Hopcroft-Karp Algorithm?
- What is an Adjacency List?
- What is the Adjacency Matrix of a graph and what are some observations that can made from the matrix about the graph and vice versa?
- What is a weighted graph?
- What is Dijkstra's Algorithm?

Notes



- This mean u and v are adjacent in G1 if and only if f(u) & f(v) are adjacent in graph G2

Isomorphic graphs



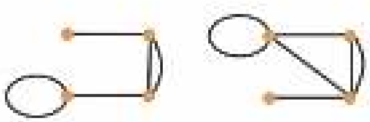
$f: G_1 \rightarrow G_2$

a_i	a_1	a_2	a_3	a_4	a_5
$f(a_i)=b_i$	b_1	b_2	b_3	b_4	b_5

Properties of isomorphic graphs

Two graphs with different degree sequences can't be isomorphic.

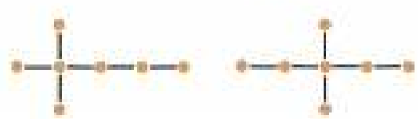
The following graphs are not isomorphic



Properties of isomorphic graphs

Two graphs with the same degree sequence aren't necessarily isomorphic.

The following graphs have the same degree sequence but are not isomorphic

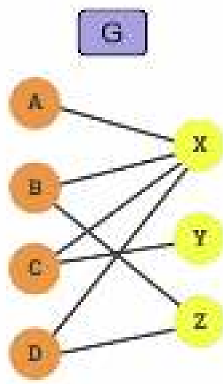


Bipartite graphs

A graph G(V, E) is called a bi-partite graph:

If the set of vertices V can be partitioned in two no-empty disjoint sets V_1 and V_2 in such a way that each edge e in G has one endpoint in V_1 and another endpoint in V_2.

Example



$V_1 = \{A,B,C,D\}$

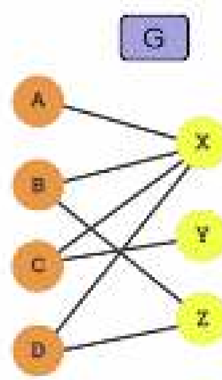
$V_2 = \{X,Y, Z\}$

The graph is 2-colourable

No odd-length cycles

Matching

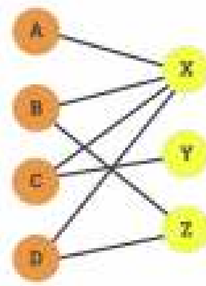
A matching is a set of pairwise non-adjacent edges, none of which are loops; that is, no two edges share a common endpoint.



A vertex is matched (or saturated) if it is an endpoint of one of the edges in the matching. Otherwise the vertex is unmatched.

Maximum matching

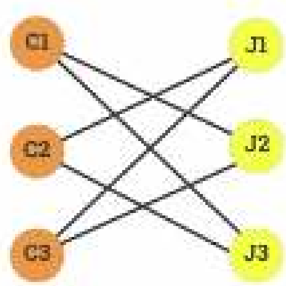
A maximum matching is a matching of maximum size such that if any edge is added, it is no longer a matching.



  - In other words, it's the largest possible number of edges that can still form a matching.
- Hopcroft-Karp Algorithm = Algorithm for solving the maximum matching problem in a bipartite graph

Key Concepts in Hopcroft-Karp Algorithm

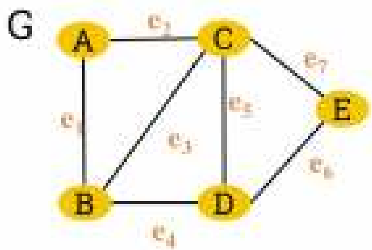
- Augmenting path = augmenting path starts on a free node and alternates between unmatched & matched edges ending on a free node and augments the cardinality of the current matching.
- Breadth-first Search = traverses the graph level by level.
- Depth-first Search = traverses a graph all the way to a leaf before starting a new path



1) Initialize M={ }
2) While there exists an Augmenting Path p
   1) Use BFS to build layers that terminate at free vertices
   2) Start at the free vertices in C, use DFS
3) Return M

Adjacency list

The adjacency list of a graph G is a list of all the vertices in G and their corresponding individual adjacent vertices.

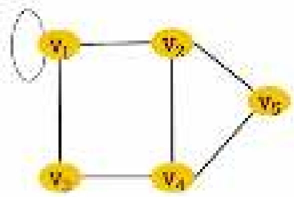


a = b, c
b = a, c, d
c = a, b, d, e
d = b, c, e
e = c, d

Example

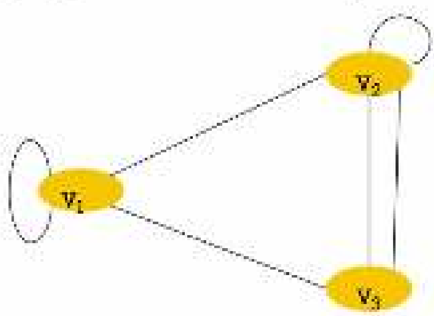
Given an undirected graph G defined by its corresponding adjacency list. Draw the graph G.

$v_1 = v_1, v_2, v_3$
$v_2 = v_1, v_4, v_5$
$v_3 = v_1, v_4$
$v_4 = v_2, v_3, v_5$
$v_5 = v_2, v_4$



Adjacency matrix

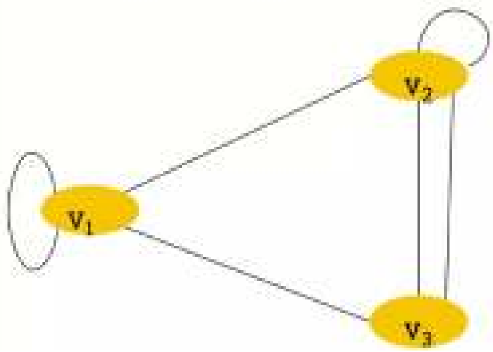
A graph can also be represented by its adjacency matrix.



$$M(G) = \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 2 \\ v_3 & 1 & 2 & 0 \end{array}$$

- Where values highlighted in red correspond to an edge between V1 & V2. The values in the leading diagonal highlighted in orange correspond to loops present in the graph

Observation



$$M(G) = \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 1 & 1 & 1 \\ v_2 & 1 & 1 & 2 \\ v_3 & 1 & 2 & 0 \end{array}$$

- Values in blue correspond to the edge between V1 & V2. Values in green correspond to the edge between V1 & V3. Whereas the values in orange correspond to the parallel edges between V2 & V3.

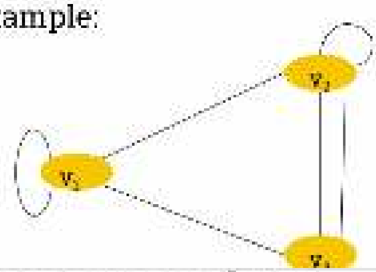
$$M(G) = \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 2 & 1 & 1 \\ v_2 & 1 & 2 & 2 \\ v_3 & 1 & 2 & 0 \end{array}$$

- We can in the previous adjacency matrix, besides the loops, every other edge is represented twice, so for consistency we can represent the loops in the matrix twice by multiplying the values in the leading diagonal by two.

Observation

The adjacency matrix of an undirected graph is symmetric.

Example:



$$M(G) = \begin{array}{c|ccc} & v_1 & v_2 & v_3 \\ \hline v_1 & 2 & 1 & 1 \\ v_2 & 1 & 2 & 2 \\ v_3 & 1 & 2 & 0 \end{array}$$

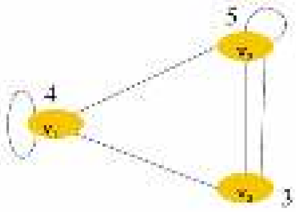
Observation

The number of edges in an **undirected** graph is equal to half the sum of all the elements (m_{ij}) of it's corresponding adjacency matrix.

- In other words, the sum of all the elements of the adjacency matrix of an undirected graph is equal to the sum of its corresponding degree sequence.

Observation

The number of edges in an **undirected** graph is equal to half the sum of all the elements (m_{ij}) of it's corresponding adjacency matrix.

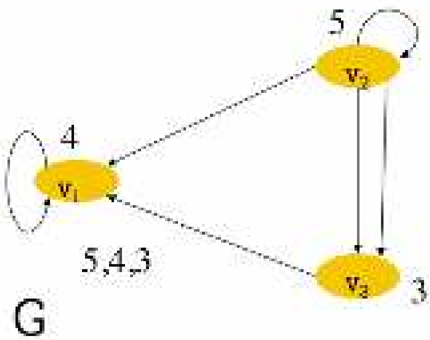


$$M(G) = . \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 0 \end{bmatrix} \end{array}$$

$\sum m_{ij}$= 1+1+1+1+1+2+2+2+2= 5+4+3=12

Number of edges in G = ($\sum m_{ij}$)/2 = 12/2 = 6

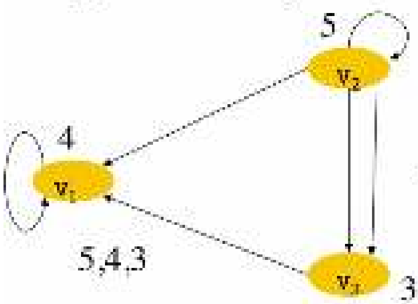
Adjacency matrix of a digraph



$$M(G) = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

Sum(m_{ij})= 1+1+1+1+2 = 6 (number of edges)

Adjacency matrix squared (M^2)

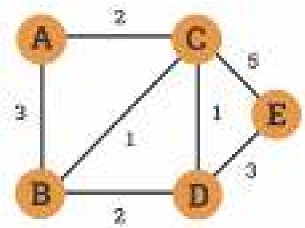


$$M^2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix}$$

$$M^2 = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 2 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

Weighted graphs

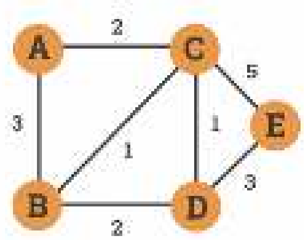
A **weighted graph** is a graph in which each edge is assigned a numerical weight.



Dijkstra's algorithm

An **algorithm** was designed by Edsger W. Dijkstra in 1956, in order to find the **shortest path between** nodes in a weighted graph.

Example



Vertex	Shortest distance from A	Previous vertex
A		
B		
C		
D		
E		

Initialisation

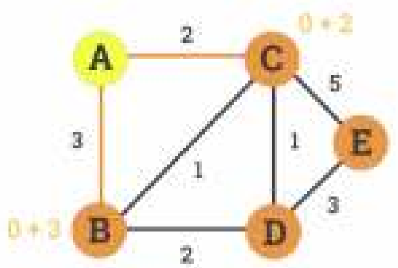
Unvisited ← [ ]
for each vertex v in *G*
    shortest_distance[*A*] ← 0
    shortest_distance[*v*] ← *Infinity*
    previous_vertex[*v*] ← *Undefined*
    add v to *Unvisited*

Unvisited = [ ]

Vertex	Shortest distance from A	Previous vertex
A	0	
B	Inf	Undefined
C	Inf	Undefined
D	Inf	Undefined
E	Inf	Undefined

Unvisited = [A,B,C,D,E]

Example: first iteration



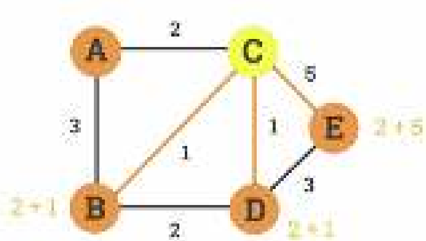
Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	Inf	Undefined
E	Inf	Undefined

Iteration 1

```
while Unvisited is not empty
        u ← vertex in Unvisited with min shortest_distance[u]
        remove u from Unvisited
        for each neighbour v of u
                alt ← shortest_distance[u] + length(u,v)
                if alt < shortest_distance[v]
                        shortest_distance[v] ← alt
                        previous_vertex[v] ← u
```

Unvisited = [B,C,D,E]

Example: second iteration



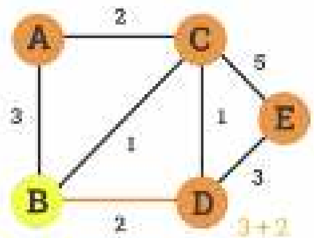
Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	3	C
E	7	C

Iteration 2

```
while Unvisited is not empty
        u ← vertex in Unvisited with min shortest_distance[u]
        remove u from Unvisited
        for each neighbour v of u
                alt ← shortest_distance[u] + length(u,v)
                if alt < shortest_distance[v]
                        shortest_distance[v] ← alt
                        previous_vertex[v] ← u
```

Unvisited = [B,D,E]

Example: third iteration



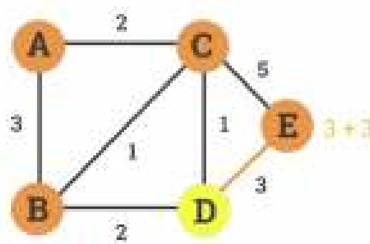
Iteration 3

while *Unvisited* is not empty:
u ← vertex in *Unvisited* with min shortest_distance[u]
remove u from *Unvisited*
for each neighbour v of u:
alt ← shortest_distance[u] + length(u, v)
if alt < shortest_distance[v]
shortest_distance[v] ← alt

Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	3	C
E	7	C

Unvisited =[D,E]

Example: fourth iteration



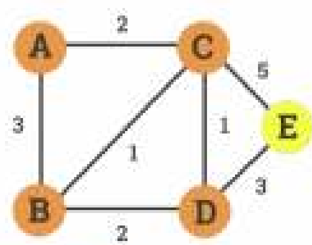
Iteration 4

while *Unvisited* is not empty:
u ← vertex in *Unvisited* with min shortest_distance[u]
remove u from *Unvisited*
for each neighbour v of u:
alt ← shortest_distance[u] + length(u, v)
if alt < shortest_distance[v]
shortest_distance[v] ← alt
previous_vertex[v] ← u

Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	3	C
E	6	D

Unvisited = [E]

Example: fifth iteration



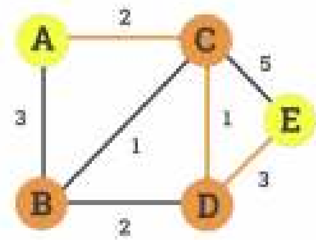
Iteration 5

while *Unvisited* is not empty:
u ← vertex in *Unvisited* with min shortest_distance[u]
remove u from *Unvisited*
for each neighbour v of u:
alt ← shortest_distance[u] + length(u, v)
if alt < shortest_distance[v]
shortest_distance[v] ← alt
previous_vertex[v] ← u

Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	3	C
E	6	D

Unvisited = [ ]

Example: complete table



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	A
C	2	A
D	3	C
E	6	D

Pseudocode

Let *G* be a graph and *s* a source vertex. The following pseudocode calculates the shortest distance and the previous vertex from *s* to every other node in the graph

```
Unvisited = { }
for each vertex v in G
        shortest_distance[v] ← Infinity
        previous_vertex[v] ← Undefined
        add v to Unvisited
        shortest_distance[s] ← 0
while Unvisited is not empty:
        u ← vertex in Unvisited with min shortest_distance[u]
        remove u from Unvisited
        for each neighbour v of u
                alt ← shortest_distance[u] + length(u, v)
                if alt < shortest_distance[v]
                        shortest_distance[v] ← alt
                        previous_vertex[v] ← u
return shortest_distance[], previous_vertex[]
```

Summary

In this week, we learned what isomorphism is, what bipartite & isomorphic graphs are & the adjacency list/matrix of a graph. Also we looked at what is a weighted graph is & Dijkstra's Algorithm for finding the shortest path between nodes in a weighted graph.