CSCI 2011: Graphs

Chris Kauffman

Last Updated:
Tue Jul 31 15:44:29 CDT 2018

Logistics

Reading: Rosen

- Now: 10.1 - 10.5, 9.6
- ~~Next: 11.1 - 11.5~~
- Trees later in 4041

Goals

- Finish up graphs
- Course Evals

Assignments

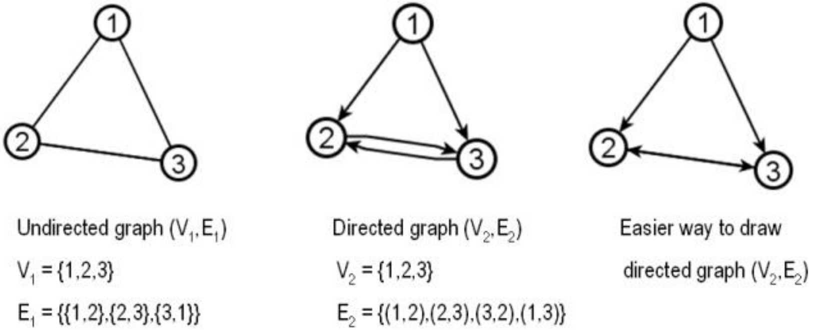
- A08: Post later today
- Due today

Schedule

- Wed: Review for Final Exam
- Thu: **Final Exam** 12:20-2:20

Graph Basics

Formally a graph $G = (V, E)$, pair of vertex set and edge set



Vertices or Nodes

- The "dots" in graphs
- Usually represent places, things
- Set V of ints, names, etc.

Edges or Connectios

- The lines between edges
- Usually represent relationships between them
- Set E of pairs of vertices

Types of Graphs

Directed vs Undirected

- Directed graphs: edges direction so $e = (u, v)$ means $u \to v$ and $(u, v) \not\equiv (v, u)$
- Undirected graphs: no direction to edges so $(u, v) \equiv (v, u)$

Labeled vs Unlabelled

- Edges and vertices can be given **labels** which indicate more information about relationships: edge Labels "Likes", "Hates"
- Won't discuss much except for numeric labels on edges

Weighted vs Unweighted

- An **edge-weighted graph** or just **weighted graph** has numbers associated with each edge so edge $e = (v_1, v_2, w_i)$
- Very common for transportation, communication networks: edge weights are distances / times / costs
- Also possible to have vertex weights but won't discuss much

Undirected Graph Terminology

Adjacency and Neighbors in Unirected Graphs

- Vertices connected by an edge are called **adjacent**
- The set of vertices adjacent to vertex v is called its **neighborhood** and sometimes denoted $N(v)$

Degree in Undirected Graphs

- Degree of a vertex $\deg(v)$ is the number edges connected to it
- In undirected graphs, edge $e = (v, u)$ counts for both v and u
- Edge (v, v) counts twice for the degree of a vertex
- Leads to the following **handshaking property**:
  For an undirected graph $G = (V, E)$ with $|E| = m$ (number of edges is m) the following holds.

$$2m = \sum_{v \in V} \deg(v)$$

Data Structures for Graphs

Dense Graph / Matrix

- Usually use contiguous blocks of memory
- **Adjacency Matrix**
- 2D array of `edge[][]`
- Each entry `edge[i][j]`
  - `true/false` for unweighted graph
  - Numeric value for weighted graph
- Symmetric matrix if graph is undirected

Sparse Graph / Matrix

- Many formats
- Most common for graphs: **Adjacency Lists**
- Array of lists of adjacent vertices

```
// unweighted: neighbors only
neighbors[i] = {4,  9,  17};
// weighted: another array
weights[i]   = {0.5,0.1,1.4}
```

- Majority of graphs in practice are sparse

Pictures

Draw some pictures of how these look

Exercise: Trade-offs on Graph Data Structures

Discuss trade-offs between these two structures for graphs

- Matrix (2D Array)
- Adjacency List (Array of Lists of neighbors)

Ground your answers by determining the big-O Complexity of the following for both data structures

1. Determine if there is an edge between vertices `i` and `j`
2. Print all neighbors of a specific vertex
3. Print all edges in the entire graph
4. In a directed graph, determine all vertices that point to vertex
5. The space complexity for $|V|$ vertices, $|E|$ edges

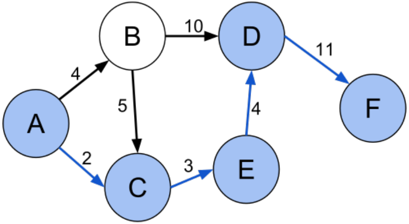
Give your answer in terms of

- $|V|$ number of vertices
- $|E|$ number of edges
- $|N(v)|$ size of neighbor set of vertex v

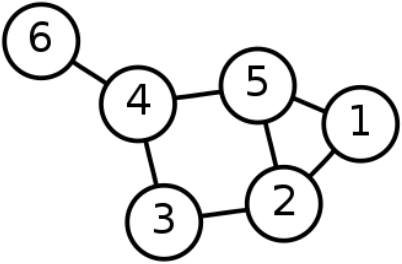
Answers: Trade-offs on Graph Data Structures

1. Determine if there is an edge between vertices `i` and `j`
   - Matrix: $O(1)$: single matrix element access
   - AdjList: $O(|N(v)|)$: `for` loop over list of neighbors
     - Maybe $O(\log(|N(v)|))$ if list is sorted
2. Print all neighbors of a specific vertex
   - Matrix: $O(|V|)$: `for` loop across a row
   - AdjList: $O(N(v))$: `for` loop over list of neighbors
3. Print all edges in the entire graph
   - Matrix: $O(|V|^2)$: doubly nested `for` loop over vertices
   - AdjList $O(|V| + |E|)$: nested loop but only visit edges
4. In a directed graph, determine all vertices that point to vertex
   - Matrix $O(|V|)$: `for` loop over column in matrix
   - AdjList $O(|V| + |E|)$: nested loop over all vertices
5. The space complexity for $|V|$ vertices, $|E|$ edges
   - Matrix $O(|V|^2)$: $|V| \times |V|$ matrix
   - AdjList $O(|V| + |E|)$: $|V|$ array for vertices, combined list length of $|E|$

Paths and Cycles



Directed graph with Path Shown

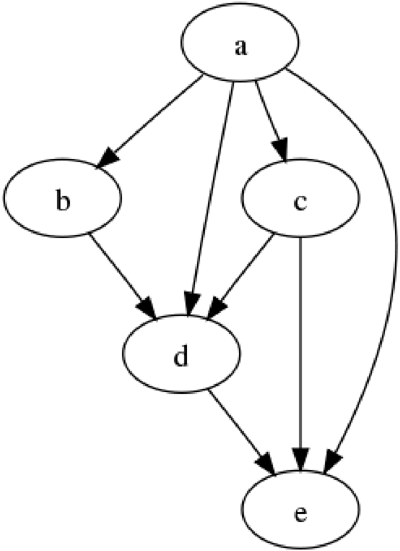


Undirected Graph

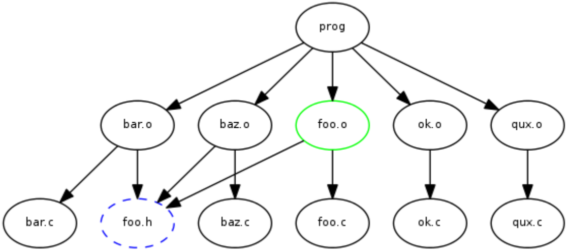
- A **path** is a sequence of edges $e_1, e_2, e_3, \ldots, e_n$ connecting two vertices v and u
- Has length n for the number of edges in it
- A cycle is a path that starts and ends on the same vertex
- Are there any cycles in the graphs above?
- Do trees have cycles?

DAG: Directed Acyclic Graphs

- Directed so edges have direction
- Acyclic as in no cycles
- Often used to indicate **dependencies** between vertices
  - a depends on b, c, d, e
  - b depends on d
  - c depends on d, e
  - e has no dependencies
- Comes up very frequently in computing in a variety of contexts



Examples of DAGs in Computing



- Building programs involves compiling dependent pieces the merging them
- A **build system** like `Makefiles` specifies dependencies and how to resolve them

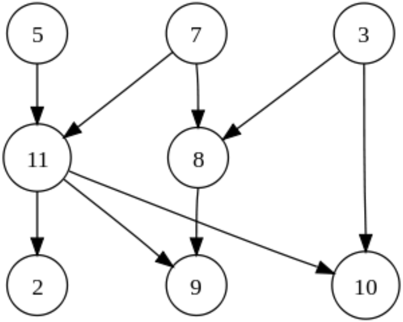
```
prog : bar.o baz.o foo.o ok.o qux.o
  gcc -o prog $^

bar.o : bar.c foo.h
  gcc -c bar.c

ok.o : ok.c
  gcc -c ok.c
...
```

Exercise: Topological Sort

- A DAG indicates dependencies and creates a **partially ordered set** of the vertices
- In a build system, need to determine which thing to do first (e.g. which code to compile first)
- Usually achieved via **topological sort**: produce a listing of vertices compatible with the partial ordering in the DAG
  - Bottom "children" last
  - Top-most nodes first



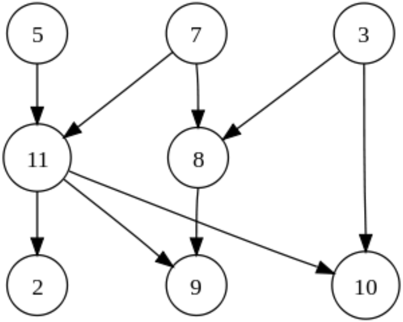
- Topological order may not be unique: two possibilities

```
order 1: 5, 7, 3, 11, 8, 2, 9, 10
order 2: 3, 5, 7, 8, 11, 2, 9, 10
```

- Find another valid ordering

Answers: Topological Sort

- 5, 7, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

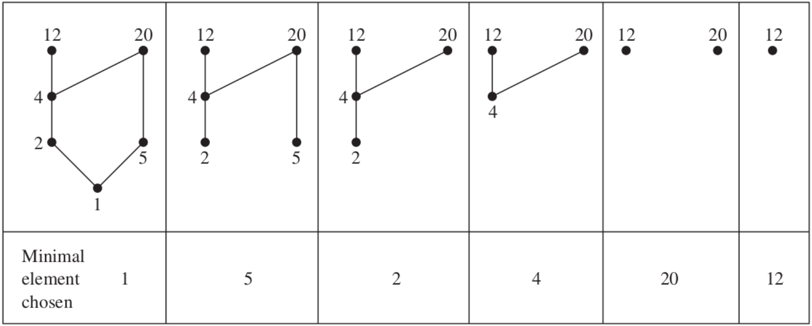


- Topological order may not be unique: two possibilities

```
order 1: 5, 7, 3, 11, 8, 2, 9, 10
order 2: 3, 5, 7, 8, 11, 2, 9, 10
```

- Find another valid ordering

Textbook Algorithm for Topological Sort

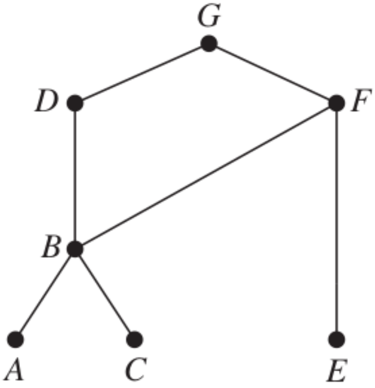



```
vertex[] topo_sort(graph G){
  int n = number_of_vertices(G);
  vertex order[] = new vertex[n];
  for(i=n-1; i>=0; i++){
    vmin = find a "minimal" vertex in G;
    order[i] = vmin;
    G = remove vmin from G;
  }
  return vertex_order;
}
```

- From Section 9.6 on partially ordered sets
- Works fine here for DAGs

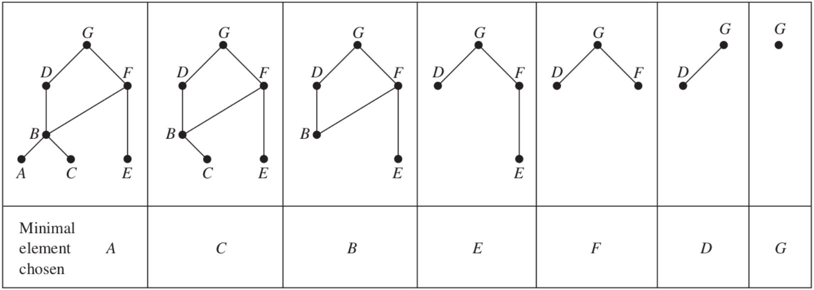
Exercise: Textbook Algorithm for Topological Sort

```
vertex[] topo_sort(graph G){
  int n = number_of_vertices(G);
  vertex order[] = new vertex[n];
  for(i=n-1; i>=0; i++){
    vmin = find a "minimal" vertex in G;
    order[i] = vmin;
    G = remove vmin from G;
  }
  return order;
}
```



- Demonstrate Algorithm on the following DAG (with direction implied by height)
- Discuss it's computational complexity: what assumptions are needed?

Answers: Textbook Algorithm for Topological Sort




```
vertex[] topo_sort(graph G){
  int n = number_of_vertics(G);
  vertex order[] = new vertex[n];
  for(i=n-1; i>=0; i++){
    vmin = find a "minimal" vertex in G;
    order[i] = vmin;
    G = remove vmin from G;
  }
  return order;
}
```

- Computational complexity is HARD to determine as the pseudocode is vague
- Determining a "minimal" vertex is non-trivial
- Removing a vertex is also non-trivial

Topological Sort is usually a Depth First Search

Problems with Textbook Algorithm

- Computational complexity is HARD to determine as the pseudocode is vague
  - Could end up search all remaining vertices/edges
- Determining a "minimal" vertex is non-trivial
  - Search for vertices with no children
- Removing a vertex is also non-trivial
  - Must eliminate all edges that point to it

Graph Searches are Useful

- Depth-first search is very useful for topological sort
- Depth-first and Breadth-first search are useful for lots of things
- Will adapt depth-first search for to determine a valid topological ordering

Depth First Graph Search

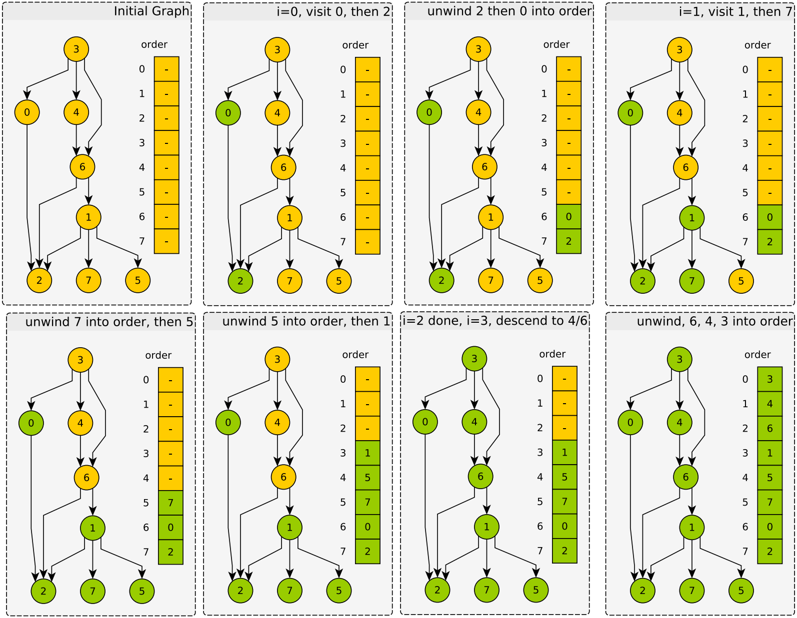
- Visit every node in a graph
- Go deep down paths until
  - Minimal vertex: no neighbors
  - Previously visited vertex
- Global vars track answers

```
int  order[];   // holds topo order
int  last;      // index into order
bool visited[]; // marks verts as found

void depth_first_toposort(graph G){
  int n_vert = num_vertices(G);
  order = new int[n_vert];     // init globals
  visited = new vool[n_vert];
  last = n_vert - 1;
  for(int i=0; i<n_vert; i++){ // iter over verts
    helper(G, i);
  }                            // order now filled
}                              // visited all true

void helper(graph G, int vert){// recursive
  if(visited[vert] == true){   // base case:
    return;                    // already visited
  }
  visited[vert] = true;        // now visited
  int neighbors[] = neighbors(G,vert);
  int n_neigh = length(neighbors);
  for(int j=0; j<n_neigh; j++){
    helper(G, neighbors[j]);   // visit neighbors
  }
  order[last] = v;             // push into order
  last = last-1;
}
```

DFS Topo Search



Exercise: Depth First Graph Search

- Computational Complexity?
- What data structure could be used for order and `last` rather than a plain array?

```
int  order[];   // holds topo order
int  last;      // index into order
bool visited[]; // marks verts as found

void depth_first_toposort(graph G){
  int n_vert = num_vertices(G);
  order = new int[n_vert];     // init globals
  visited = new vool[n_vert];
  last = n_vert - 1;
  for(int i=0; i<n_vert; i++){ // iter over verts
    helper(G, i);
  }                            // order now filled
}                              // visited all true

void helper(graph G, int vert){// recursive
  if(visited[vert] == true){   // base case:
    return;                    // already visited
  }
  visited[vert] = true;        // now visited
  int neighbors[] = neighbors(G,vert);
  int n_neigh = length(neighbors);
  for(int j=0; j<n_neigh; j++){
    helper(G, neighbors[j]);   // visit neighbors
  }
  order[last] = v;             // push into order
  last = last-1;
}
```

Answers: Depth First Graph Search

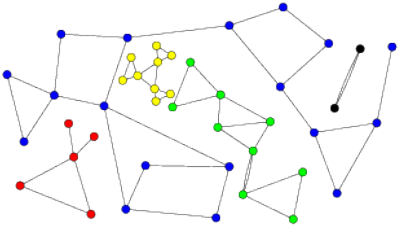
- Computational Complexity?
  - $O(|V| + |E|)$
  - $|V|$: number of vertices
  - $|E|$: number of edges
- What data structure could be used for `order` and `last` rather than a plain array?
  - A **stack** as reformulated to the right
  - Bottom of stack are minimal vertices

```
int_stack order[]; // holds topo order
bool visited[];    // marks verts as found

void depth_first_toposort(graph G){
  int n_vert = num_vertices(G);
  order = new int_stack();     // init globals
  visited = new vool[n_vert];
  for(int i=0; i<n_vert; i++){ // iter over verts
    helper(G, i);
  }                            // order now filled
}                              // visited all true

void helper(graph G, int vert){// recursive
  if(visited[vert] == true){   // base case:
    return;                    // already visited
  }
  visited[vert] = true;        // now visited
  int neighbors[] = neighbors(G,vert);
  int n_neigh = length(neighbors);
  for(int j=0; j<n_neigh; j++){
    helper(G, neighbors[j]);   // visit neighbors
  }
  stack_push(order, v);        // push into order
}
```

Depth First Search for Connected Components



- **Connected components** in graphs are sets of vertices that are reachable from one another
- Variants of depth first search can also be used to determine connected components in graphs

Breadth First Search in Graphs

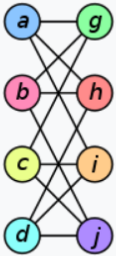
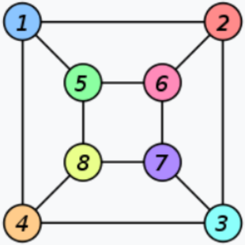
- BFS uses a **queue** and need not be recursive
  - Depth-First Search uses recursive call stack
- Can also calculate connected components
- Can compute **distance from an origin vertex**
  - single source shortest path
- Easy to modify this algorithm to use edge weights for distance

```
queue verts[];  // next verts/distances
bool visited[]; // marks verts as found
int dists[];    // dist from origin

void breadth_first_shortest_path(graph G,
                                 int origin)
{
  verts = new queue();
  enqueue(verts, (origin, 0));
  while(not empty(verts)){
    (v, dist) = dequeue(verts);
    if(visited[v] == true){
      continue to next iteration;
    }
    visited[v] = true;
    dists[v] = dist;
    int neighbors[] = neighbors(G,v);
    int n_neigh = length(neighbors);
    for(int j=0; j<n_neigh; j++){
      enqueue(neighbors[j], dist+1);
    }
  }
  // dists[] now contains distances from origin
}
```

Isomorphism: Graph "Equality"

- Notion of equality of two graphs: derive a mapping function from one to the other
- Mapping function must preserve all properties of the graph
  - Vertex degrees (number of neighbors)
  - Paths and degrees
  - Sub Graphs
- We'll study a few by hand tricks determine "not isomorphic"

Graph G	Graph H	An isomorphism between G and H
		$f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$

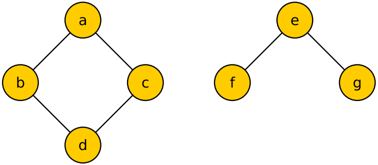
Things to Look for in Isomorphism

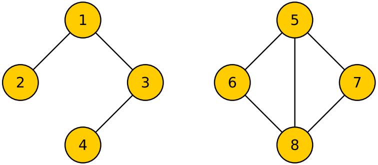
Not Isomorphic If...

- Different number of vertices and edges
- Number of vertices with given degree is different

Beyond Easy stuff

Look lack of **paths** with same degree sequence

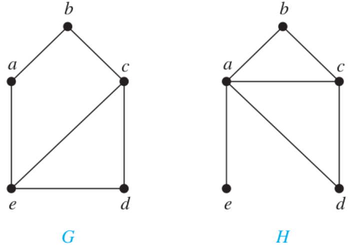




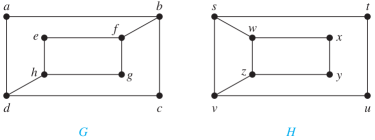
Exercise: Show Not Isomorphic

- Show these graphs are not isomorophic
- Use an argument based on a degrees of vertices or lack of paths with specific degree sequences

Pair 1



Pair 2



Answers: Show Not Isomorphic

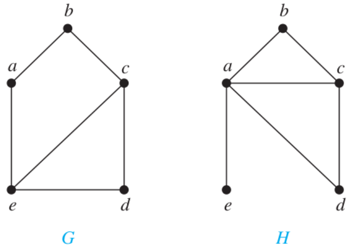
Pair 1

- G has no vertices with degree 1
- H has e of degree 1
- No mapping possible

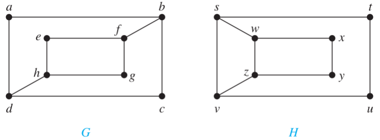
Pair 2

- G, H both have vertices with degree 3
- Consider mapping for d from G
- Need a path with degree sequence $\{(d,3),(a,2),(b,3)\}$ in H
- Choices are v, z, w, s in H but none have such a sequence
- No mapping possible

Pair 1

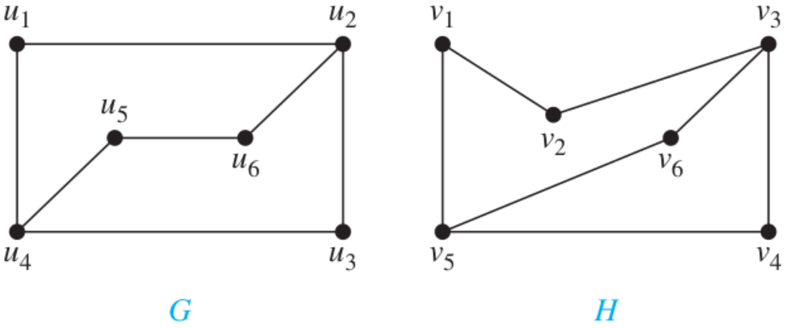


Pair 2

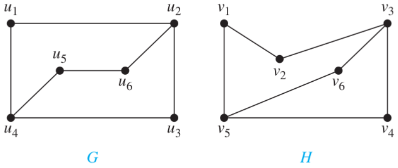


Exercise: Show graphs isomorphic

- Show the two graphs below are isomorphic
- Do so by finding a mapping of vertices u_i to v_j



Answers: Show graphs isomorphic



$$
\begin{aligned}
f(u_1) &= v_6 & \quad f(u_4) &= v_5 \\
f(u_2) &= v_3 & \quad f(u_5) &= v_1 \\
f(u_3) &= v_4 & \quad f(u_6) &= v_2
\end{aligned}
$$

- Several other possibilities
- Notice rearrangement of adjacency matrix makes them equal

$$
\mathbf{A}_G = \begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} u_1 & u_2 & u_3 & u_4 & u_5 & u_6 \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

$$
\mathbf{A}_H = \begin{array}{c} \\ v_6 \\ v_3 \\ v_4 \\ v_5 \\ v_1 \\ v_2 \end{array}
\begin{array}{c}
\begin{array}{cccccc} v_6 & v_3 & v_4 & v_5 & v_1 & v_2 \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
$$

Exercise: Counting Question

- How many possible mappings are there from two graphs with N vertices?
- How does one check for correctness

Algorithms for Graph Isomorpism

- For graphs with N vertices, $N!$ possible mappings between them
- A **brute force** algorithm would simple check all possible mappings
  - Determine a permutation
  - Re-arrange one adjacency matrix according to permutation
  - Check if matrices are equal
- This is the algorithm known to work in all cases
- Graph Isomorphism is an NP-Hard, no know if it is NP-complete
- Many heuristics exist to speed up in some cases but not all