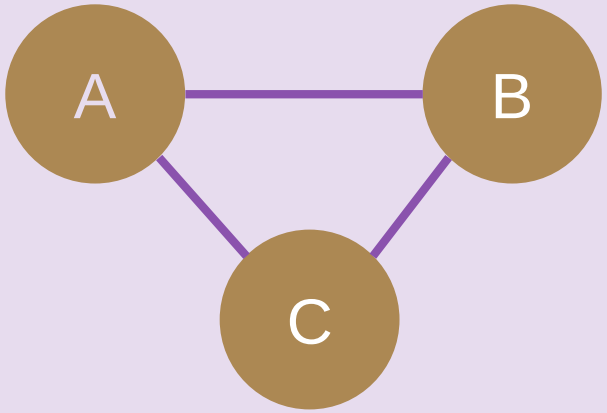
GRAPH THEORY

Hadrian Ang, Kyle See, March 2017

What are graphs?

A graph G is a pair $G = (V, E)$ where V is a non-empty set of vertices and E is a set of edges e such that $e = \{a, b\}$ where a and b are vertices.



Why graphs?

Graphs are usually used to represent different elements that are somehow related to each other.

What are vertices?

Vertices, sometimes called **nodes**, are objects that form graphs. They are usually used to represent certain elements to be related with another.

Examples

Cities in a country
People in a social network

What are edges?

Edges are two element subsets of V (at least in the undirected case, but more on this later). They usually represent connections in a system.

Examples

Roads between cities
Friendships between people

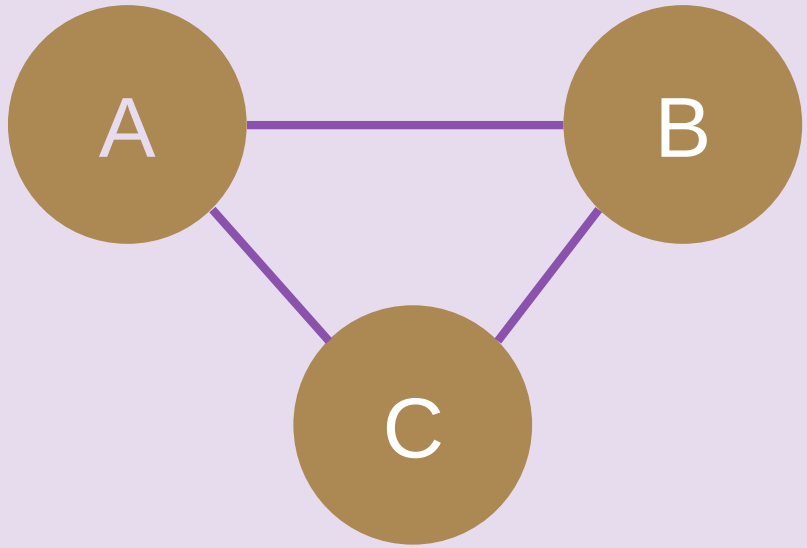
Just some terms…

Two vertices are said to be **adjacent** if they are joined by an edge. In Fig. A, the vertices A and B are adjacent.

An edge is said to be **incident** to the vertices it joins. In Fig. A, the edge $\{A, B\}$ is incident to vertex A and B**.**

The number of edges incident to a vertex is called the **degree** of that vertex. It is sometimes denoted as $\deg(v)$ for some vertex v. All vertices in Fig. A have a degree of 2.

Figure A



Graph Directedness

Graphs can be **directed**, sometimes called **digraphs**, or undirected.

A

In an **undirected** graph, edges go both ways. An edge from A to B is also an edge from B to A. Undirected edges are usually drawn as straight lines between vertices. Edges are subsets of the set V.

Example: $\{A, B\}$

B

In a **directed** graph, edges do not go both ways. In Figure B, there is an edge from A to B, but no edge from B to A. Edges are usually drawn with arrows to show directedness. Instead of being subsets, edges are ordered pairs with both elements from the set V.

Example: (A, B)

Figure A

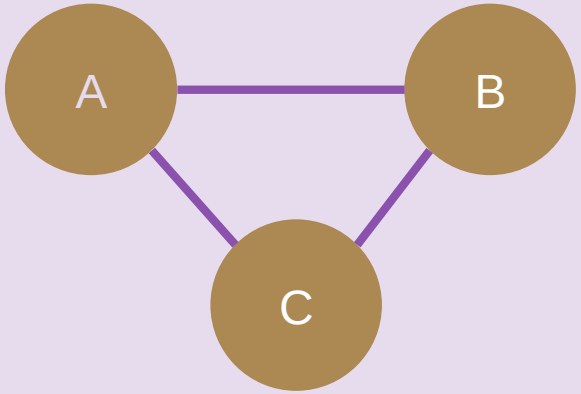
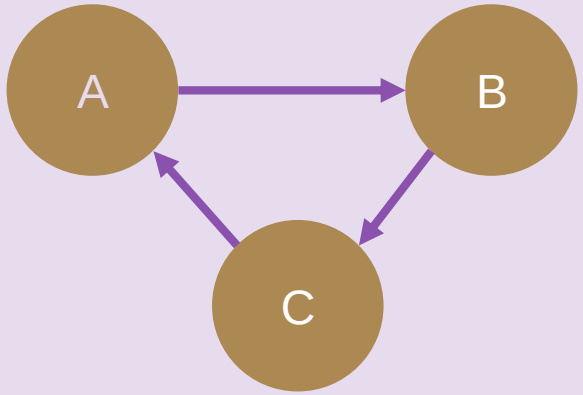


Figure B



Graph Weightedness

Graphs can be **weighted**, or **unweighted**.

A

In an unweighted graph, edges do not have any specific numeric value relative to other edges in the graph.

Example: Following on Instagram, a specific follower is not intrinsically more valuable than another one.

B

In a weighted graph, edges each have some associated value called the **weight** of the edge. Weights are usually drawn near the edge they are associated with.

Example: Roads (some roads are longer than others, thus have more weight).

Figure A

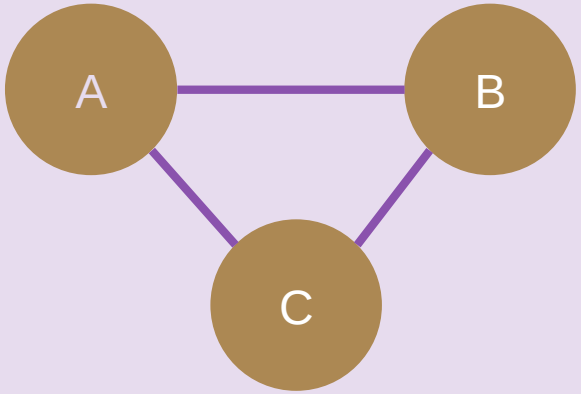
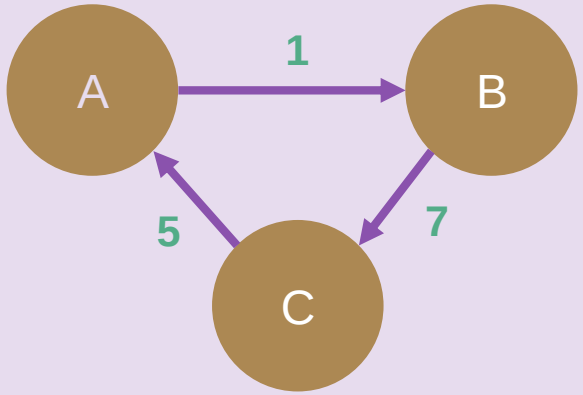


Figure B



Complex Graphs

A graph is called a complex graph when it has loops or multi-edges.

A

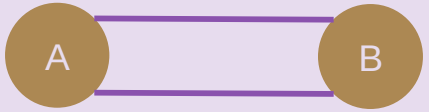
Loop

A **loop** is an edge that has the same source and destination vertex. In other words, it is an edge from a vertex to itself.

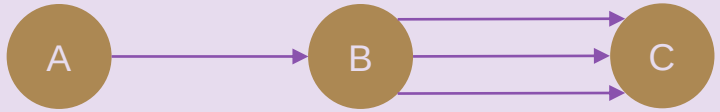
B

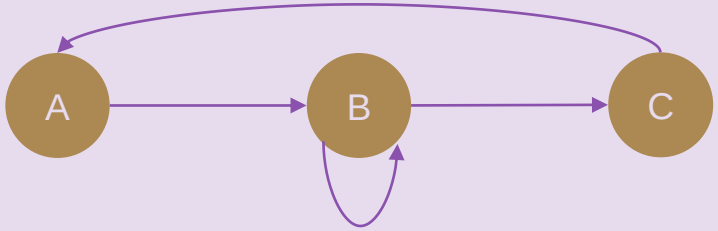
Multi-edge

Multi-edges or **multiple edges** are two or more edges between the same two vertices. Note that in a directed graph, edges from A to B and B to A are not multi-edges (they have to be going in the same direction to be considered so).



Below are some examples of complex graphs





A graph without multi-edges or loops is called a simple graph.

Walks and Paths

A **walk** is a sequence of vertices and edges

$$v_0, v_1, \dots, v_k$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\}$$

Such that $\{v_i, v_{i+1}\}$ is an edge in G for all i, $0 \leq i < k$. The walk starts at v_0 and ends at v_k. The **length** of the walk is k, its source is v_0 and its destination is v_k.

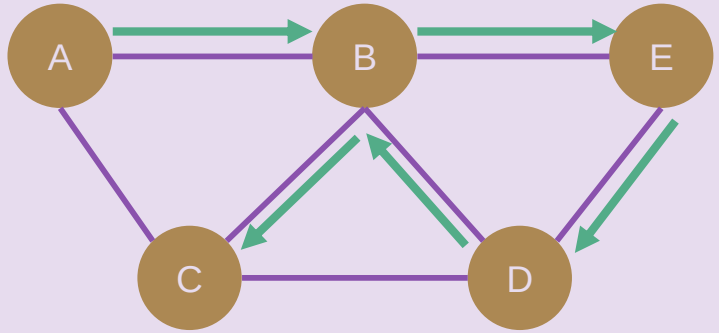
A **path**, similar to a walk, is also a sequence of vertices and edges.

$$v_0, v_1, \dots, v_k, v_i \neq v_j \ \forall i, j$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-1}, v_k\}$$

A path cannot cross the same vertex twice, however.

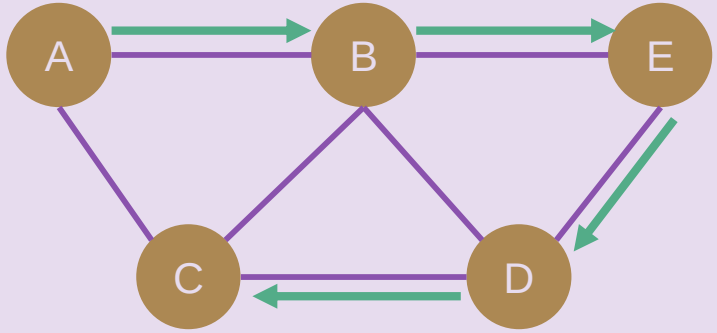
All paths are walks, but not all walks are paths.

Sample Walk



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

Sample Path



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

Closed Walks and Cycles

A **Closed Walk**, just like a walk, is a sequence of vertices and edges, but it starts and ends with the same vertex. In other words, the source of the walk is the same as the destination.

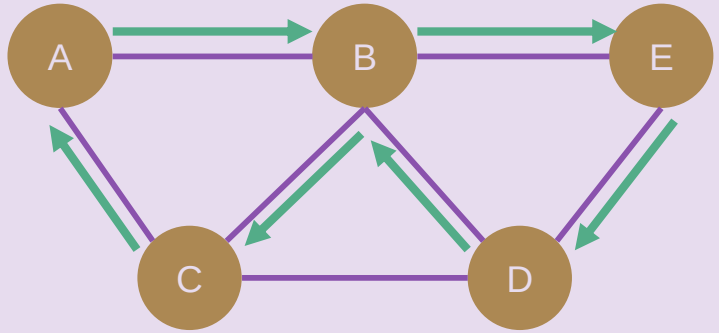
$$v_0, v_1, \ldots, v_k$$
$$\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{k-1}, v_k\}$$

Such that $\{v_i, v_{i+1}\}$ is an edge in G for all i, $0 \leq i < k$ and $v_0 = v_k$.

A **Cycle** is to a closed walk as a path is to a walk. It is also a sequence of vertices and edges starting and ending on the same vertex, but all vertices are unique except for the start / end vertex.

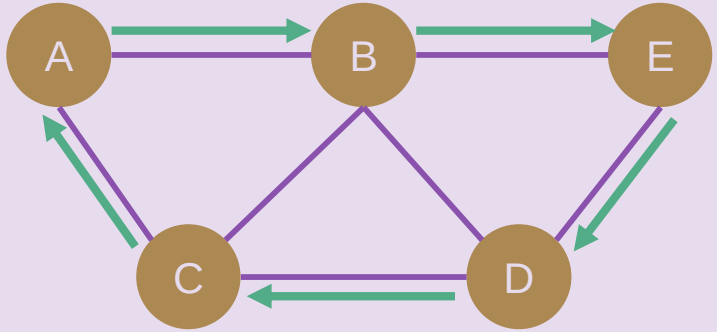
All cycles are closed walks, but not all closed walks are cycles.

Sample Closed Walk



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

Sample Cycle



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

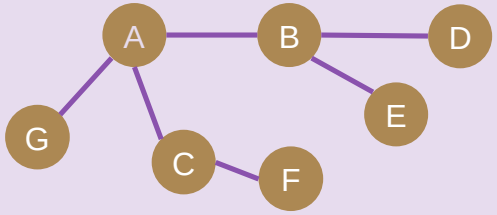
Special Graphs

Many special types of graphs have names to more uniquely define them. The following are a few of these graphs.

A

Tree

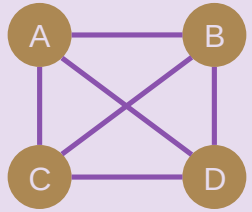
An undirected graph where there is exactly one path from any vertex to any other vertex



B

Complete Graph

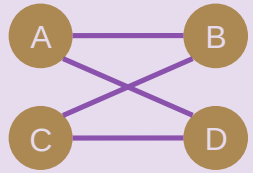
A graph where each vertex has an edge to every other vertex in the graph.



C

Bipartite Graph

A graph where the vertices can be divided into two sets such that no vertex in one set has an edge to another vertex in the same set. Called complete bipartite if all vertices in one set are connected to all vertices in the other set.



D

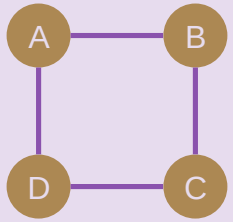
Path/Linked List

A tree that simply consists of one connected line of vertices.

E

Cycle

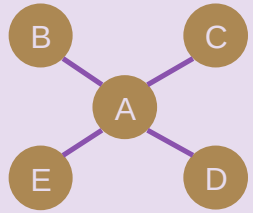
A graph that consists of one entire loop or cycle.



F

Star

A tree with one “center” vertex connected to all the other vertices, but all other vertices only connected to the center vertex.



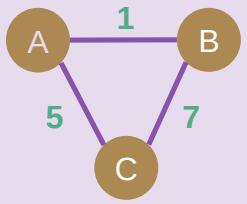
Computer Representation

There are three main ways of representing graphs using a programming language. Carefully selecting which representation to use is important when solving a problem.

A

Adjacency Matrix

Stores the graph in a matrix usually represented as a 2-D array such that `matrix[i][j]` contains the weight of the edge (i, j).



	A	B	C
A	0	1	5
B	1	0	7
C	5	7	0

B

Adjacency List

Each vertex is associated with a list (can be represented in C++ as a vector) populated by vertices each is adjacent to. One possible implementation of this is to use objects.

```
class Vertex{
  public:
    int index;
    vector<Vertex*> adj;
    Vertex(int ind){
      index = ind;
    }
};

int main(){
  Vertex* a = new Vertex(1);
  Vertex* b = new Vertex(2);
  //If there is an undirected
  //edge between a and b
  a->adj.push_back(b);
}
```

C

Edge List

Perhaps least common among the three approaches, a list is created to store all the edges in the graph. Edges can be stored as objects.

```
class Edge{
  public:
    int source, dest;
    Edge(int s, int d){
      source = s;
      dest = d;
    }
};

int main(){
  vector<Edge*> edges;
  //Assume 1 and 2 are indices
  //of adjacent vertices stored
  //in an array

  Edge* edge = new Edge(1,2);
  edges.push_back(edge);
}
```

Adjacency Matrix

Pros

Can easily retrieve the weight of the edge between two connected vertices or check if two vertices are connected (just check `matrix[A][B]`).

Cons

Looping through all neighbors of one vertex is expensive because you have to go through all of the empty cells.
Takes a lot of memory, a lot of it wasted on empty cells (especially for graphs with many vertices, but few edges).
Because of this, adjacency matrices cannot be used for problems with relatively large limits.
Each cell can only contain one value, so adjacency matrices do not support complex graphs (unless you use a 2-dimensional matrix of vectors).

Notes

A ***sentinel*** value must be used to “fill in” the empty spaces left by edges that do not exist in the graph. Typical values used include 0, -1, and INT_MAX. The choice of sentinel values depends on the problem and how you implement your solution. For example, some problems may require you to have negative weight edges, so it may be wiser to use INT_MAX.

Adjacency Matrix

Sample Implementation

```
int adj[N][N];

int main(){
    //start of test case
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            adj[i][j] = sentinel;
        }
    }
    //if a and b are connected
    adj[a][b] = weight;
    adj[b][a] = weight; //if undirected

    //check all neighbors of a
    for(int i=0; i<n; i++){
        if(adj[a][i] == sentinel) continue;
        //use adj[a][i]
    }
}
```

Adjacency List

Pros

Using objects can help in attaching specific information to each vertex.
Less space used than adjacency matrix.
It is easy to loop through all the neighbors of a vertex.

Cons

To check if two vertices are connected, one will have to loop through the whole adjacency list of one of the vertices.

Notes

Don’t forget to clear each vector when beginning a new test case if you declare them globally.

Adjacency List

Sample no objects implementation

```
vector<int> adj[N];

int main(){
    //start of test case

    for(int i=0; i<n; i++){
        //clear adj[i]
    }

    //if a and b are connected
    adj[a].push_back(b);
    adj[b].push_back(a); //if undirected

    //check all neighbors of a
    for(int i=0; i<adj[a].size(); i++){
        //a is adjacent to adj[a][i]
    }
}
```

```
/*if weighted, use pair<int, int> or create a
second vector<int> array and use the same indices
to correspond to the same edge, for example:

vector<int> adj[N], adjw[N];
adj[a].push_back(b);
adjw[a].push_back(weight);

use adj[a][i] and adjw[a][i] */
```

Edge List

Pros

Can easily iterate over all the edges in the graph (required for some algorithms).
Least space used since edges are not repeated for the two vertices they are incident to (unlike in adjacency lists)

Cons

Makes it difficult to get only the edges incident to a specific vertex (there are ways to get around this, but it requires more space and effort, more on this in the sample implementation).
Like adjacency lists, it is expensive to determine if two vertices are connected.

Notes

Don’t forget to clear the list of edges at the beginning of each test case.
Creating helper functions may make using edge lists easier for many problems.

Edge List

Sample no objects implementation

```
int n, e, last[N], prev[E], head[E];

void init(int n){
    e = 0;
    //set last[i] to -1 for I from 0 to n
}

void addEdge(int u, int v){
    head[e] = v; prev[e] = last[u];
    last[u] = e++;
}
```

```
int main(){
    //start of test case
    init(nodes);

    //if a has an edge to b
    addEdge(a, b);
    addEdge(b, a); //if undirected

    //check all neighbors of a
    for(int e=last[a]; e >= 0; e = prev[e]){
        //a is adjacent to head[e]
    }
}

//for weighted graphs, add an extra array similar
to adjacency list
```

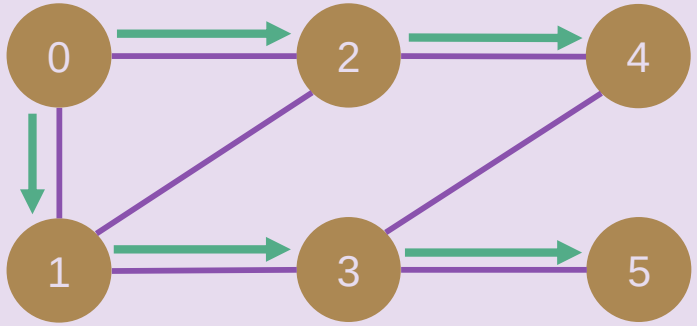
Graph Searching

There are two main ways of searching through or traversing a graph: breadth-first or depth-first.

A

Breadth-First Search (BFS)

Start at some vertex we call the root of the search. From this root, visit all adjacent vertices first before moving to the next level. This can be done with the use of the queue data structure.



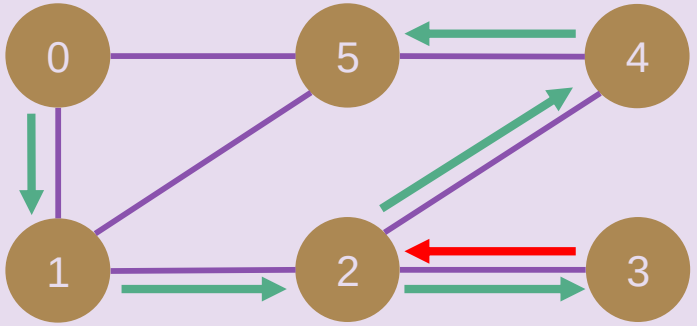
Applications

Using BFS, we can solve the shortest path problem for unweighted graphs, as it is ensured that we go from the source to the destination in the smallest number of vertex-vertex traversals.

B

Depth-First Search (DFS)

Start at some vertex we call the root of the search. From this root, keep going to the next level of vertices until a dead-end is reached. Once a dead-end is keep moving up levels until a new DFS can be started on an unvisited node. This can be done with the use of a stack data structure.



Applications

Has applications in finding strongly connected components in a graph (subgraphs where all vertices are reachable from every other vertex, but more on this in the future).

Graph Searching

Sample implementation (using the object version of adjacency list)

A

Breadth-First Search (BFS)

```
int main(){
    //setup graph

    queue<Vertex*> q;
    q.push(root);
    while(q.size()>0){
        Vertex* v = q.front(); q.pop();
        for(int i=0; i<v->adj.size(); i++){
            q.push(v->adj[i]);
        }
    }
}
```

B

Depth-First Search (DFS)

```
int main(){
    //setup graph

    stack<Vertex*> s;
    s.push(root);
    while(s.size()>0){
        Vertex* v = s.top(); s.pop();
        for(int i=0; i<v->adj.size(); i++){
            s.push(v->adj[i]);
        }
    }
}
```

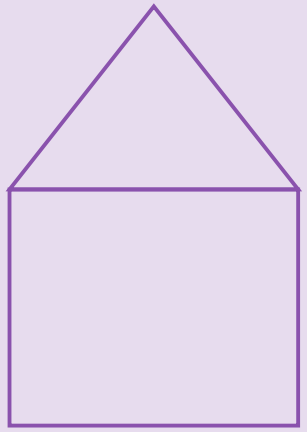
Note

To avoid looping infinitely, we keep track of which vertices have already been visited in the course of the search, typically using booleans. If a vertex has been visited before, we skip it because we would already be repeating a part of the search we did before.

Euler Walk

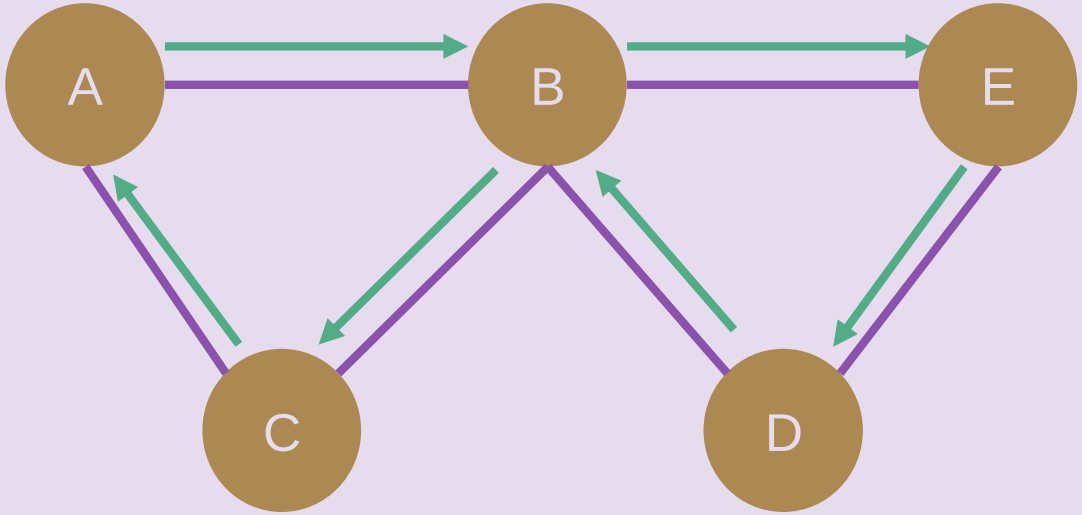
An **Euler Walk** (sometimes called an Euler Path) is a walk that goes through every edge in the graph exactly once.

Example: Drawing the house below without lifting your pen is an example of an Euler Walk.



Euler Tour

An **Euler Tour** (sometimes called an Euler Cycle) is an Euler Walk that starts and ends on the same vertex.



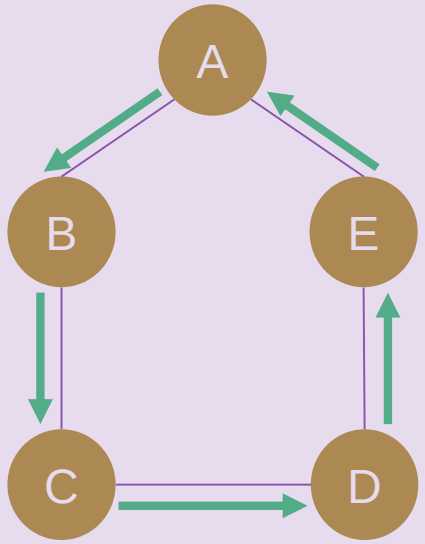
Eulerian Graphs

A

Euler Tour

Theorem: A connected graph (meaning there is a path from each node to every other node) has an Euler Tour if and only if every vertex in the graph has even degree.

Intuition: To traverse every edge exactly once and go back to the start vertex, the number of ways to enter a vertex must be equal to the number of ways to exit.

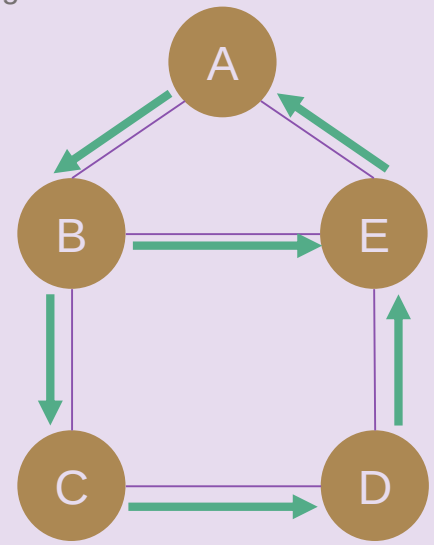


B

Euler Walk

Corollary: A connected graph has an Euler Walk if and only if there are at most two vertices with odd degree

Intuition: Start at one of the vertices with odd degree. If there is an edge that leads to the other vertex of odd degree, ignore it first. Find an Euler Tour starting from this initial vertex then traverse the previously ignored edge.



Note

For those interested in the algorithm to determine an Euler Tour/Walk, look up Hierholzer's Algorithm.

Directed Acyclic Graph

As the name implies, a **Directed Acyclic Graph** or DAG is simply a directed graph without any cycles.

DAGs are important because they allow us to capture the reality of dependencies (tasks that make prerequisites of other tasks).

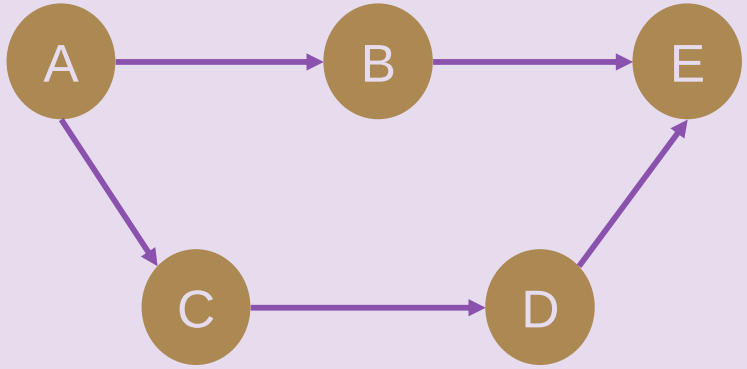
Example:

Having to take some subjects before others.

A certain program needing the output of another program before executing.

Topological Sorting

A topological sorting of a DAG is a list of all the vertices such that each vertex v appears before all other vertices reachable from v. Note that a single DAG may have multiple different topological orderings.

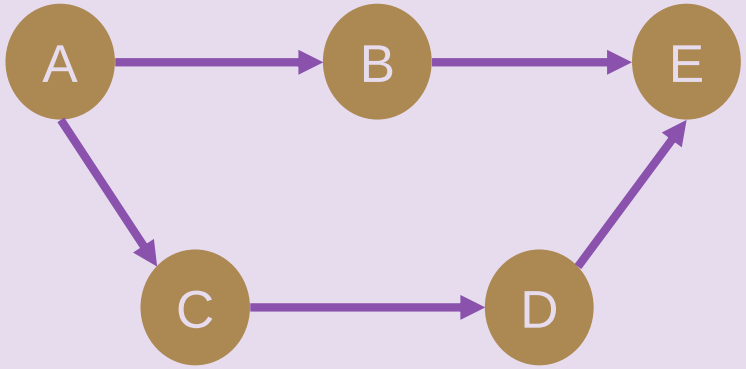


Sample Topological Sorting: A, B, C, D, E
Another possible sorting is A, C, D, B, E

Topological Sorting

Theorem: Every finite DAG has a topological ordering. We can show this by starting a search from the *minimal* elements.

A vertex v is *minimal* if and only if v is not reachable from any other vertex (has an in-degree of 0).



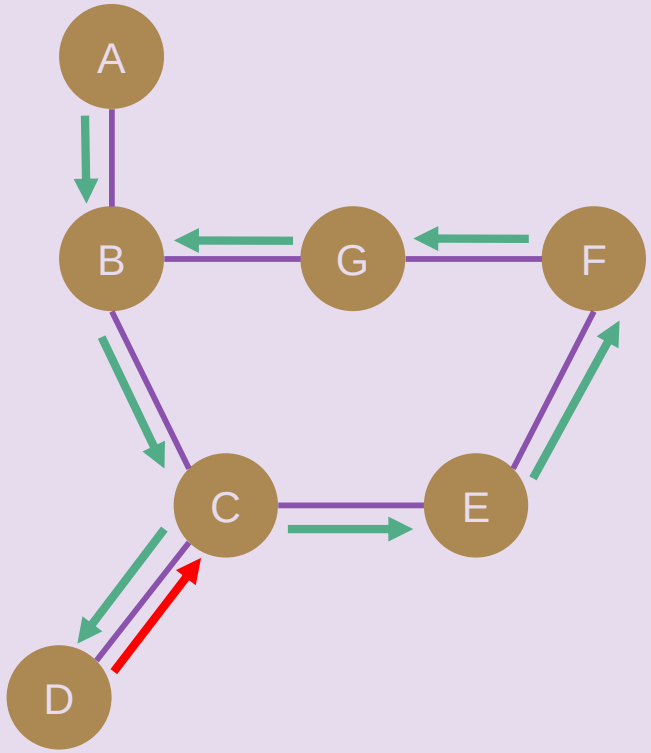
Vertex A is the only minimal vertex in the above graph

Towards an Algorithm

Generating a topological ordering can be done with a BFS-like approach. Starting a search from minimal elements, we can easily generate a topological ordering. Once a vertex has been visited, we can remove it from the graph (subsequently removing all of its edges). Assuming a valid DAG, deleting a vertex will result in new minimal elements. We then place these new minimal elements into our queue and continue our search from there. The order with which we visit vertices and delete them from the graph should be a topological ordering.

Cycle Finding

Finding the cycles within a graph can be done using a DFS-like approach. Start the search from some root vertex. When processing a vertex, we know that a path exists from the root to the current vertex using the vertices we passed through in the previous iterations, excluding those we have already backtracked from. Because of this, if there is an edge from the current vertex to some vertex already in this path, we know that a cycle exists from that vertex back to itself.



Cycle exists: B, C, E, F, G

Sample Implementation (using object version of adjacency list and recursion instead of a stack)

```
void DFS(Vertex* v, Vertex* p){
    v->inPath = true;
    for(int i=0; i<v->adj.size(); i++){
        if(v->adj[i] == p) continue;
        if(v->adj[i]->inPath) //cycle found
        DFS(v->adj[i], v);
    }
    v->inPath = false;
}

int main(){
    //setup graph

    DFS(root, NULL);
}

//Note that this only finds cycles reachable from root.
//You may have to repeat the process for other vertices.
```

Challenges (to be submitted)

- Codeforces 292B – Network Topology
- UVa 280 – Vertex
- UVa 10150 – Doublets
- UVa 459 – Graph Connectivity
- UVa 10203 – Snow Clearing
- UVa 11060 – Beverages
- UVa 10116 – Robot Motion
- UVa 12582 – Wedding of Sultan
- UVa 572 – Oil Deposits

Challenges (at least 2, the rest is optional)

- UVa 1103 – Ancient Messages
- Codeforces 510C – Fox and Names
- Codeforces 115A – Party
- Codeforces 475B – Strongly Connected City
- Codeforces 22C – System Administrator
- Codeforces 402E – Strictly Positive Matrix