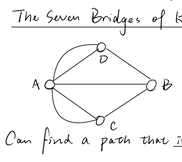
Introduction to Graph

The Seven Bridges of Königsberg



Can find a path that include every edge exactly once?

No. A graph has that path (Euler Path) iff it contains exactly 0 or 2 vertices with an odd degree (i.e. # edges linked with this vertex)

How to make it possible?

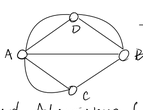
Solution: Build one more bridge to remove 2 odd-degree vertices

How to find a possible path?

Algorithm:

```
u ← any odd-degree vertex
if no such vertex:
    u ← any vertex.
while u has an edge not taken yet
    take that edge (u,v)
    u ← v
```

This algorithm may stuck!

e.g.  if A → D → A → C → A → B → C

then stuck at C!

Modified Algorithm (Hierholzer's algorithm)

```
u ← any odd-degree vertex
Find-path(u)
while there are still edges not taken
    u ← any previously seen vertex that
        is endpoint of untaken edge.
    p ← Find-path(u)
    insert p into existing path at u
```

```
Find-path(u)
    while u has an edge not taken yet.
        take that edge (u,v)
        u ← v
```

Traveling Salesman Problem

"How to visit all places and then return to starting point, travelling the shortest possible distance?"

⇒ reformulated as graph problem:

given a graph in which edges has weights. how to find cycle with minimum total weight that includes all vertices.

Still don't have algorithm for this

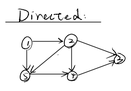
Undirected and Directed Path

Graph: $G = (V, E)$

V: vertices ⇒ Set of nodes.

E: edges ⇒ Set of edges between pair of nodes

Undirected: 

Directed: 

Simple Graph (Most Case):

There is at most 1 edges between the same pair of nodes in undirected graph

There is at most 2 edges between the same pair of nodes in undirected graph

Undirected Graph:

$n = |V|$ $m = |E|$

deg(v): # edges touching v.

$\Rightarrow \sum_{v \in V} deg(v) = 2|E|$

Directed Graph

$\sum_{v \in V} deg^{out}(v) = \sum_{v \in V} deg^{in}(v) = |E|$

e.x. What is maximum number of edges in a directed graph G with |V| nodes?

⇒ $|V| \cdot (|V|-1) = |V|^2 - |V|$

What is maximum number of edges in a undirected graph G with |V| nodes?

⇒ $\frac{|V|(|V|-1)}{2}$

e.x. Show that if all nodes in a graph have degree 3. then the # of nodes: |V| must be even

$\sum_{v \in V} deg(v) = 2|E| = 3 \cdot |V|$

⇒ 3·|V| is even since $|E| \in \mathbb{Z}$ ⇒ |V| is even.

e.x. Assume a graph where all nodes have an odd degree, Show that |V| must be even

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

let node i's degree $= 2k_i + 1$ where k_i is an integer ≥ 0.

$\Rightarrow 2|E| = 2(k_1 + \cdots + k_n) + |V|$, where $n = |V|$.

$|V| = 2 \cdot (|E| - \sum_{i=1}^{n} k_i) \Rightarrow$ |V| is even

e.x. Show that the # of guests who shake hands an odd number of time is even.

Graph $G = (V, E)$.

let guests shaking hands an odd number of time be V_{odd}

let guests shaking hands an even number of time be V_{even}.

$|V_{odd}| + |V_{even}| = |V|$.

$\sum_{v \in V_{odd}} deg(v) + \sum_{v \in V_{even}} deg(v) = 2|E|$

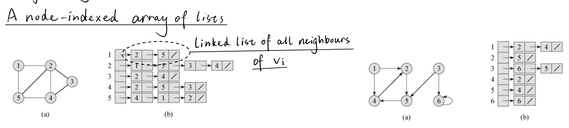
$\Rightarrow \sum_{v \in V_{odd}} deg(v) = 2|E| - \sum_{i=1}^{|V_{even}|} k_i$

$2\sum_{i=1}^{|V_{odd}|} k_i + |V_{odd}| = 2|E| - \sum_{i=1}^{|V_{even}|} k_i \Rightarrow |V_{odd}|$ is even.

Graph representation

Adjacency list:

A node-indexed array of lists

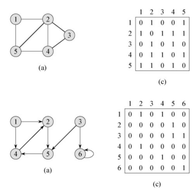


Given node u. retrieving all neighbour in O(deg(u)) time.

Given u,v checking if (u,v) is an edge in O(deg(u)) time.

Space $(\Theta(|V| + |E|))$: $S = \underbrace{\Theta(|V|)}_{\text{all vertices}} + \underbrace{\Theta(|E|)}_{\text{all edges}}$

Adjacency Matrix:



|V| x |V| matrix

retrieving all neighbor in Θ(|V|) time

Given u,v checking if (u,v) is an edge in Θ(|V|) time.

Space $\Theta(|V|^2)$

Adjacency list is more commonly used. Since most graphs are Sparse

(* Sparse graph: $|E| = O(|V|)$ (or $|E| \ll |V|^2$) ⇒ not fully-connected!)

Usually. Assume no self loop and duplicated edges

$0 \leq |E| \leq |V|(|V|-1)/2$. for undirected graph

$0 \leq |E| \leq |V|(|V|-1)$ for directed graph

These two representations can convert to each other in $\Theta(|V|^2)$ time!

Paths and Connectivity

* Paths is a sequence $v_1, v_2, \ldots, v_k$ such that (v_i, v_{i+1}) is an edge

the length of such path is k-1

* def: a path is simple if all nodes are distinct

* An undirected graph is connected if for all pairs of nodes u,v. there is a path between them

⇒ if connected, then $|E| \geq |V|-1$ (edge case: O—O—O—O. $|E| = |V|-1$)

* An cycle is a path $v_1, v_2, \ldots, v_n$ where $v_1 = v_n$

⇒ An cycle is simple if first k-1 nodes are distinct

Trees:

An undirected graph is a tree if connected & does not contain cycles

An undirected graph is a forest if does not contain cycles (collection of trees)

Theorem:

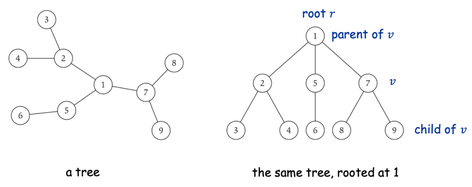
Any two of the following imply the third:

(1) G is connected

(2) G contains no cycles.

(3) $|E| = |V| - 1$

Rooted Tree:



a tree

the same tree, rooted at 1

Basic Graph Algorithms

Origin input: Adjacency lists / matrices —(derive: BFS / DFS)→ Structure

Breadth-first Search:

Exploring outward from s in all possible directions, adding nodes "one layer" at a time.

$L_0 = \{s\}$.

L_1: all neighbour of L_0

L_2: all neighbour of L_1 which do not belongs to L_0 / L_1.

⋮

L_i: all neighbour of L_{i-1} which do not belongs to $L_0 / L_1 / \cdots / L_{i-1}$

Distance from u to v: length of shortest path between them

Theorem: For each i, L_i contains all nodes at distance exactly i from s

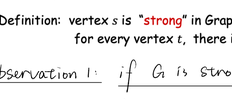
There is a path between t and s iff t appears in some layers

Algorithm:

```
BFS(G,s):
for each vertex u ∈ V-{s}: // initialization
    u.color ← white  // stage
    u.d ← ∞          // depth
    u.p ← nil        // parent.
s.color ← gray
s.d ← 0
initialize an empty queue Q:
enqueue(Q,s)
while Q ≠ ∅ do
    u ← dequeue(Q)
    for each v ∈ Adj[u]: (Adjacency List/matrix)
        if v.color = white then:
            v.color ← gray  // partially processed
            v.d ← u.d+1
            v.p ← u.
            Enqueue(Q,v)
    u.color ← black // fully processed
```

Running time: $\sum_u (1 + deg(u)) = O(|V| + |E|)$ (remark: $\Theta(E)$ if connected)

Connected Components:



Modified Algorithm for finding Connected Component:

```
BFS(G):
for each vertex u ∈ V do:
    u.color ← white
    u.d ← ∞
    u.p ← nil
for each vertex u ∈ V do:
    if u.color ← white then
        BFS_Visit(G,u)
```

```
BFS_Visit(G,s): // Assume s color is white
s.color ← gray
s.d ← 0
initialize an empty queue Q:
enqueue(Q,s)
while Q ≠ ∅ do
    u ← dequeue(Q)
    for each v ∈ Adj[u]: (Adjacency List/matrix)
        if v.color = white then:
            v.color ← gray  // partially processed
            v.d ← u.d+1
            v.p ← u.
            Enqueue(Q,v)
    u.color ← black // fully processed
```

S-t Connectivity and Shortest path in directed graphs

S-t Connectivity (often called reachability for directed graph)

Undirected: s can reach t ⟺ t can reach s.

Directed: not necessarily true

S-t shortest path

Undirected: s→t's shortest path = t→s's shortest path

Directed: not necessarily true

BFS in directed graph: Same in undirected graph

Strong Connectivity in Directed Graph

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Definition: vertex s is "strong" in Graph G if, for every vertex t, there is a path from s to t and from t to s.

Observation 1: if G_1 is strong then every vertex in G_1 is strong.

Observation 2: A graph is strong iff every vertex in G is strong.

Algorithm for check Strong Connectivity:

idea: ① Pick any nodes → ② Run BFS in s → ③ Reverse all edges in G and run BFS from s

Return true iff all nodes reached in Both BFS

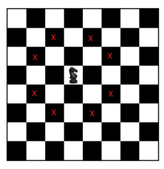
```
Strongly-connected-components(G):
create G^rev which is G with all edges reversed:
while there are nodes left do:
    u ← any node
    run BFS in G starting from u.
    run BFS in G^rev starting from u,
    C ← {nodes reached in both BFS's}.
    output C as a strongly connected component.
    remove C and its edges from G and G^rev
```

Running time: O(|V||E|)

exist a O(|V|+|E|) algorithm

e.x. Find the minimum number of steps taken by a knight to reach an destination (x',y') from an input position (x,y)

From some position (x, y) the knight can move to the following positions provided that they are within the board limits:
$(x+2, y-1)$
$(x+2, y+1)$
$(x-2, y+1)$
$(x-2, y-1)$
$(x+1, y+2)$
$(x+1, y-2)$
$(x-1, y+2)$
$(x-1, y-2)$



Starting from (x,y) and apply BFS ⇒ Continue this process for each neighbor

⇒ the first time that you reach (x',y') corresponds to the minimum number of steps

DFS: Depth-first Search

Algorithm:

```
DFS(G):
for each vertex u ∈ V do:
    u.color ← white.
    u.p ← nil.
for each vertex u ∈ V do:
    if u.color = white then.
        DFS_visit(u)
```

```
DFS_visit(u):
u.color ← gray.
for each v ∈ Adj(u) do:
    if v.color = white:
        v.p ← u.
        DFS_visit(v)
u.color ← black.
```

Running Time: $\Theta(|V| + |E|)$

Application: Cycle Detection

idea: if $|E| = |V| - 1$ ⇒ A tree.

if $|E| > |V| - 1$ ⇒ Must have a cycle!

if $|E| < |V| - 1$ ⇒ cannot be connected

Algorithm:

Run BFS/DFS to find all connected component

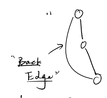
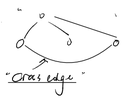
For each component. Count its # edge.

if # edges ≥ # vertices ⇒ Cycle!

Running Time: $\Theta(|V|+|E|)$

After running BFS/DFS on an undirected graph. all edges can be classified:

① Tree edges: traverse. by the BFS/DFS

② Back edges:  ③ Cross edge: 

DFS for cycle detection

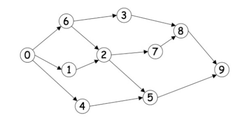
```
CycleDetection(G):
for each vertex u ∈ V do:
    u.color ← white.
    u.p ← nil
for each vertex u ∈ V do:
    if u.color = white. then DFS_visit(u)
return "no cycle"
```

```
DFS_visit(u):
u.color ← gray.
for each v ∈ Adj(u) do:
    if v.color = white:
        v.p ← u.
        DFS_visit(v)
    else if v ≠ u.p then          "Back Edge"
        output "Cycle found."
        while u ≠ v do:           print out cycle.
            print u.
            u ← u.p.
        output v
        return.
u.color ← black
```

DAG: Directed Acyclic Graph

"With no cycle"

- Directed Acyclic Graph (DAG): Directed graph with no cycles.
- A Topological ordering of a graph is a linear ordering of the vertices of a DAG such that if (u, v) is in the graph, u appears before v in the linear ordering



- Topological ordering may not be unique

Topological Sort

Algorithm

Output a vertex u with in-degree zero in current graph.

Remove u and all edges (u,v) from current graph

If graph is not empty goto step 1

```
TopologicalSort(G)
initialize Q to be an empty queue.
for each u in V do:
    if indegree(u) = 0 do:
        Enqueue(Q,u)
While Q is not empty:
    u ← Dequeue(Q)
    Output u.
    for each v ∈ adj[u] do:
        indegree(v) ← indegree(v) - 1
        if indegree(v) = 0 then
            Enqueue(Q,v)
```