graphs

- graph $G = (V, E)$ consists of two finite sets :
  - V : vertices
  - E : edges

$E = \{(x, y) \mid x, y \in V\}$

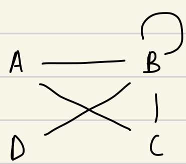
$$0 \leq |E| < \frac{|V|(|V|-1)}{2} \rightarrow {}^{n}C_{2} = \frac{n(n-1)}{2}$$

degree of vertex : no of edges incident to it

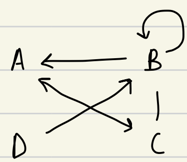
tree is a special graph w/ no cycle

- types

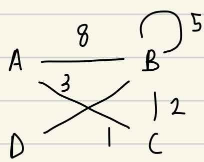
undirected



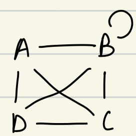
directed



weighted



complete



- terms

for undirected: if $e = (x, y)$ e is incident to x & y
x is adjacent to y and viceversa

- if E is unoreded, G is undirected. else G is directed

- if $e = (x, y)$ is an edge in a directed graph, then y can be reached from x through 1 edge and target y is adjacent to source x

- path : sequence of distinct vertices, each adjacent to to predecessor (except for the first one)

  $|V| = |E| + 1$

- cycle : path containing at least 3 vertices such that the last one is adjacent to the first one

  $|V| = |E|$

- connected graph : undirected : path from any vertice to any other vertice

strongly connected graph : directed : path from any vertex to any other vertex

cyclic graph : 1/+ cycles are present
else : acyclic

graph representation

adjacency matrix adjacency list

Adjacency Matrix

- 2-D array of size $|V| \times |V|$
- $(u,v) \in E \Rightarrow AdjM[U][V] = 1$ else, 0
- if undirected, matrix is symmetric

```
typedef struct _graph{
        int Vsize;
        int esize;
        int **AdjM;
  }Graph;
```

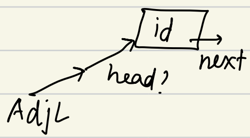
- easy to check if nodes are adjacent
- downside: if sparsely connected. most entries are 0
- time complexity is low (for search if adjacent)
  space complexity is high $\rightarrow O(|V|^2)$
- if graph is weighted, store weight in matrix

$$AdjM[u][v] = \begin{cases} W(u,v) & \text{if } (u,v) \in E \\ c & \text{otherwise} \end{cases}$$

c can be defined as 0 (weight by capacity) or ∞ (weight by cost)

Adjacency List

- like a closed address hash table
- 2D array of listnodes ? → pointer to a pointer to a node
- linked list is used to rep connections to other vertices for each address

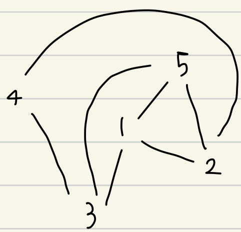


- array size is $|V|$
  number of nodes in linked list is $2|E|$

  every edge is repped twice

1	→ 2 → 3 → 5
2	→ 5 → 4
3	→ 1 → 5
4	→ 3 → 2
5	→ 1 → 3 → 2

array reps vertices



- access time for AdjL[u][v] is linear

  searching the whole list

- space complexity $= |V| + 2|E|$
  $= O(|V| + |E|)$

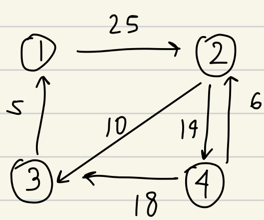
  upper bound of rate of growth

worst case : $O(|V|^2)$
if graph is complete
and $|E| = \frac{|V|(|V|-1)}{2}$

- if weighted, store weight as another data field in node

-
```c
typedef struct _listnode {
    int id ; // or weight
    struct _listnode *next;
} ListNode ;

typedef struct _graph {
    int vSize,
    int eSize ;
    ListNode **AdjL ;
} Graph ;
```

- 

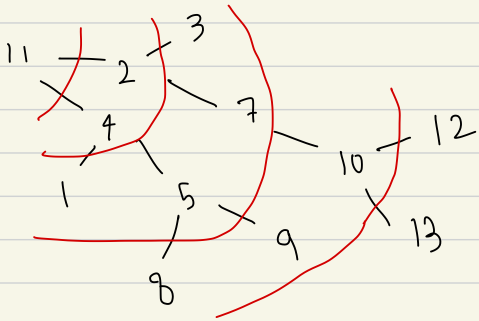
1	→ (2,25)
2	→ (4,14) → (3,10)
3	→ (1,5)
4	→ (2,6) → (3,18)

traversal of graph

- visit every node systematically.
  perhaps perform an operation

breadth first search

- ≈ level order traversal of trees



- used a queue

  [11] → [4 2] → [3 7 4] → [1 5 3 7] → and so on

- if its cyclic / vertice can be accesses by 2 paths
  have an explored list
  check if child has already been accessed.

- can check if directed graph is strongly connected
  ↳ if search results in printing of all vertices

algorithm :

```
function BFS (Graph G, Vertex v)
  create a Queue Q
  enqueue v into Q
  mark v as visited
  while Q is not empty do
     dequeue a vertex, w
     for each unvisited vertex u adjacent to w
        enqueue u in Q
        mark u as visited
     end for
  end while
end fn
```

- equally correct to visit adjacent nodes in any order
- if shortest path from s to vertex v is defined as the path w/ minimum number of edges, then BFS finds the shortest paths from s to all variables reachable from s.
- tree built by BFS is the breadth first spanning tree
  ↓
  can infer distance

- time complexity of BFS
  - → each edge is processed once in the while loop → $O(|E|)$
  - → each vertex is queued and dequeued once → $O(|V|)$
  - → worst case :

ways of representing
- $\Theta(|V| + |E|)$ if graph is repped by adjacency list
- $\Theta(|V|^2)$ if graph is repped by adjacency matrix

each vertex takes $\Theta(|V|)$ to scan for its neighbours

Depth - First Search

- ≈ pre-order traversal of the trees
- DFS explores every vertice as deeply as possible before backing up.
- function DFS (Graph G, Vertex v)

```
function DFS (Graph G, Vertex v)
  create a stack S          ← can also use recursive
  push v into Stack
  mark v as visited
  while S is not empty do
    peek stack, denote vertex as w
    if no unvisited vertices are adjacent to w then pop
    a vertex from S
   else
      push an unvisited vertex u adjacent to w
      mark u as visited
   end if
  end while
end fn
```

- if vertex has several neighbours, it would be equally correct to go through them in any order
- if directed graph is strongly connected, the tree T by DFS is a set of $|V|-1$ edges, connecting all vertices

problems to know · tower of hanoi

- applications :
  → checking connectivity
  → finding connected components
  → solving puzzles
  → solve maze (not shortest past though)

· Time complexicy is affected by graph representation
  ↳ each node once
  every edge is traversed twice (forward + backtracking)

· ∴ adjacency list time complexity is $O(|V| + |E|)$

eliminates certain permutations
saves time
not the best algo tho.
smarter than seeing all.

more on DFS - W10, 20th march

backtracking : you make a series of decisions among various choices, where :

trying various sequences until I find one that works.

→ don't have enough info to know what to choose
→ each decision leads to a new set of choices
→ some sequence of choices may be a soln to your problem

colour, queen, sudoku, hamiltonian

1. colouring problem

colour, check if adjacent have same $\xrightarrow{Y}$ backtrack colour again

N ↓

continue

I/P format : 2D adjacency matrix representation
graph[V][V]

number of colours : m

O/P format : array colours[V] should have numbers 1 to m naive soln : check all possible combos : $m^{|V|}$

recursive soln :

f^n takes current index.(ci)
if ci == |v|
print result
else assign a colour to the vertex
↓
to interate through colours. ← (for) every assigned colour: check if config is safe, recursively call the function with the next index.

if any recursive f^n returns true,
break loop and return true.

time complexity :

$$W(|V|) = 1 + mW(|V| - 1)$$
$$W(0) = 1$$
$$\begin{aligned} W(|V|) &= 1 + m(1 + mW(|V| - 2)) \\ &= 1 + m + m^2 W(|V| - 2) \\ &= 1 + m + m^2 + m^3 W(|V| - 3) \\ &= 1 + m + \dots m^{|V|-1} + m^{|V|} W(0) \\ &= 1 + m + \dots + m^{|V|} \end{aligned}$$
$$W(|V|) = \frac{m^{|V|} - 1}{m - 1}$$

time complexity becomes exponential

Queen Problem

backtracking algorithm

1. place a queen on the top left corner of the chess board

2. place queen on second, move her untill she cant be hit by the first queen.

3. place queen on 3rd col ....

4. if there is no place for the i^{th} queen, program back-tracks to move the $(i-1)^{th}$ queen.

5. if $(i-1)^{th}$ queen is at the end, prog removes queen, backtracks to $(i-2)$ col and so on

—x—

```
fn NQueens(Board[N][N], column)
  if column >= N return true          → soln found
  else
    for i ← 1, N do                   helper fn.
      if Board[i][column] is safe to place then
        Place a queen in the square
        if NQueens(Board[N][N], column+1) then return true
        end if
prev. → Delete the queen
      endif
    end for
  end if
return false
end fn
```

back tracking :

Backtracking (n)
    base caase : return true → city = dest

    for 1 to n
        do something / move forward
        if (Backtracking (n-1)) return true
        reverse whatever you have done earlier
    return false