DATA STRUCTURE AND ALGORITHM

Tutorial

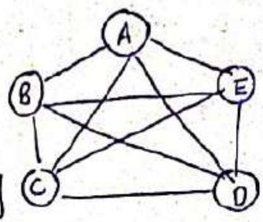
1. a) A complete undirected graph

adjacency matrix

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

adjacency list

0 A → B → C → D → E
1 B → A → C → D → E
2 C → A → B → D → E
3 D → A → B → C → E
4 E → A → B → C → D



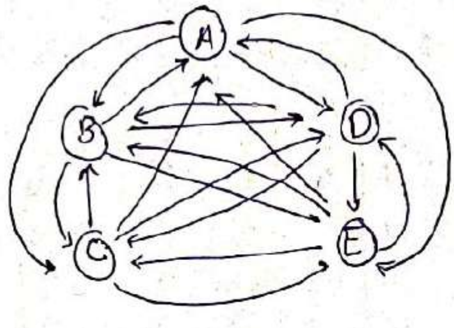
b) A complete directed graph

adjacency matrix

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

adjacency list

A → B → C → D → E
B → A → C → D → E
C → A → B → D → E
D → A → B → C → E
E → A → B → C → D



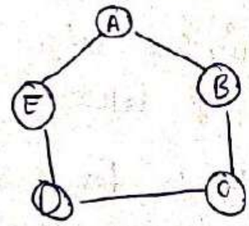
c) An undirected cycle graph

adjacency matrix

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

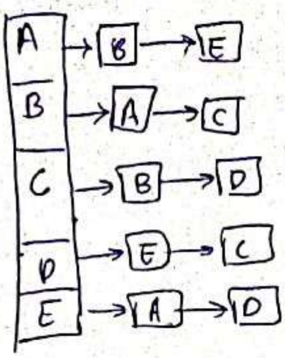
adjacency list

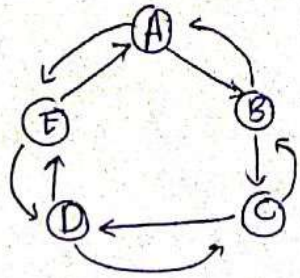
A → B → E
B — A — C
C — B — D
D — C — E
E — D — A



d) A directed cycle graph

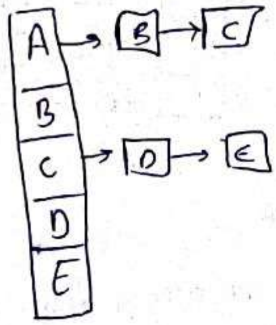
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

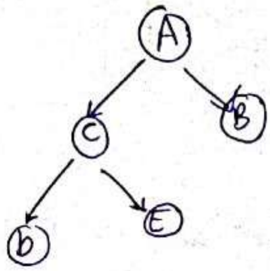




e) A binary tree , edges from parent to child.

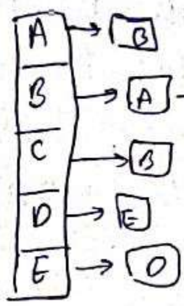
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

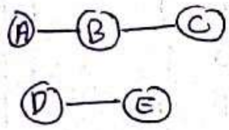




f) Undirected graph with 2 connected components

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$





---

2) a) Single edge connects 2 vertices , Also in representation a edge between x and y is counted twice $\rightarrow$ x and y , y and x

hence for 1 edge $\rightarrow 2 \times 1$'s in adjency matrix

for m edges $\rightarrow 2 \times m \times 1$'s $= 2m$ 1's

b) Same like above , 1 edge contributes 2 new linked list nodes in adjency list representation.

so for 1 edge $\rightarrow 2 \times 1$'s in adjacency list

for m edges $\rightarrow 2 \times m$ 1's.

5) a) degree of node that has most neighbour

$\Rightarrow O(n^2)$

```
int maxDegree(int n, int a[n][n])
{
   int maxDegree = 0;
   for(int i=0; i<n; i++)
   {
      int temp = 0;
      for(int j=0; j<n; j++)
      {
         if (a[i][j] == 1)
            temp++;
      }
      if (temp > maxDegree)
         maxDegree = temp;
   }
   return maxDegree;
}
```

b) check G is an empty Graph

$\Rightarrow O(n^2)$

```
int isEmpty(int n, int a[n][n])
{
   for (int i=0; i<n; i++)
   {
      for(int j=0; j<n; j++)
      {
         if (a[i][j] == 1)
            return 0;
      }
   }
   return 1;
}
```

c) check if (u,v) is in E(G) where u,v ∈ V(G)

$\Rightarrow O(1)$

```
int edgecheck(int u, int v, int a[n][n])
{
   if (a[u][v] == 1)
      return 1;
   else
      return 0;
}
```

d) check if graph contains triangle → $O(V^3)$

```
int Triangle (int n , int a[n][n])
{
  int sum , b[n][n] , x[n][n];
  for(int c=0; c<n; c++) {
    for (int d=0; d<n; d++) {
      for (int e=0 ; e<n ; e++) {
        sum = sum + a[c][e] * a[e][d];
      }
      b[c][d] = sum;
      sum=0;
    }
  }
  for (int c=0 ; c<n ; c++) {
    for (int d=0; d<n ; d++) {
      for (int e=0; e<n ; e++) {
        sum = sum + a[c][e] * b[e][d];
      }
      x[c][d] = sum;
      sum=0
    }
  }
  int true =0
  for (int i=0; i<n ; i++) {
    true += a[i][i];
  }
  if (true /6 >=1)
  {
    printf ("triangle exist");
    return 1;
  }
  else
  {
    printf ("triangle not exist");
    return 0;
  }
```

e) add an edge (u,v) to E(G) where $u, v \in V(G)$

$O(1)$

```
void add (int n, int a[n][n], int u, int v)
{
  if (u<0 || u>=n || v<0 || v>=n)
  {
    printf ("invalid");
    return;
  }
  else {
    a[u][v] = 1;
    a[v][u] = 1;
  }
}
```

f) Delete a node

→ $O(1)$

```
void delete (int n, int a[n][n], int u, int v)
{
    if (u<0 || u>=n || v<0 || v>=n)
    {
        printf ("invalid");
        return ; }
    else {
        a[u][v] = 0;
        a[v][u] = 0;
    }
}
```

g) Subdivide edge (u,v)

→ $O(n^2)$

```
int* subdivide(int n, int a[n][n], int u, int v)
{
    int b [n+1][n+1] = {0};
    for (int i=0 ; i<n ; i++) {
        for (int j=0 ; j<n ; j++) {
            if((i==u && j==v) || (i==v && j==u))
                continue
            else
                b[i][j] = a[i][j]
        }
    }
    int w = n;
    b [u][w] = 1;
    b [w] [u] = 1;
    b [v][w] = 1;
    b [w] [v] = 1;
    return b;
}
```

h) Output complement of graph

→ $O(n^2)$

```
int* complement (int n, int a[n][n])
{
    for (int i=0; i<n ; i++) {
        for (int j=0 ; j<n ; j++){
            if (i==j)
                continue;
            else
                a[i][j] = !a[i][j];
        }
    }
    return a;
}
```

i) Check graph is Eulerian

$O(n^2)$

```
int is_eulerian(int n , int a[n][n])
{
 for (int i=0 ; i<n < i++)
 {
 int degree = 0;
 for (int j=0 ; j<n ; j++)
 {
   if (a[i][j] ==1)
      degree++;
 }
  if (degree ==0)
  {
   printf ("not eulerian");
   return 0;
  }
  if (degree %2 !=0)
  {
   printf ("not eulerian");
   return 0;
  }
 }
 printf ("eulerian");
 return 1;
}
```

4)

```
struct vertex{
   int index;
   int value;
   struct vertex* next;
};

struct list {
   struct vertex* root;
};
```

9)

```
int maxDegree (struct List* list[] , int n)
{
  int degree =0;
  for (int i=0; i<n; i++)
  {
   int temp = 0;
   struct vertex* x;
    x = list [i] -> root;
   while (x != NULL)
   {
    temp++;    x = x -> next;
   }
   if (temp > degree)
     degree = temp;
  }
  return degree;
}
```

$O(V^2)$

Scanned with CamScanner

b)
```
int isEmpty ( int n , struct list * list [] )
{
  for ( int i = 0 ; i<n ; i++ )
  {
    if ( list [i] -> head ! = NULL )
    {
      return 0;
    }
  return 1;
}
```

$\Rightarrow$ $O(n)$

c)
```
int isEdge (int n , struct list* list [], int u , int v)
{
  struct vertex temp;
  temp = list [u] -> head ;
  while ( temp ! = NULL )
  {
    if ( temp . index == v )
    {
      printf (" is Edge");
      return 1;
    }
    temp = temp -> next ;
  }
  printf ("not a Edge");
  return 0;
}
```

$\Rightarrow$ $O(n)$

d)
```
int Triangle (int n , struct list * list [])
{
  int edge [2] ;
  struct vertex* vertex ;
  for (int i=0 ; i<n ; i++)
  {
    edge [0] = i;
    vertex = list [i] -> head ;
    while ( vertex ! = NULL )
    {
      edge [1] = vertex -> index ;
      for (int j=0 ; j<n ; j++)
      {
        if (is Edge ( edge[0] , j , n , list) && is Edge (edge [1] , j , n, list)
        {
          return 1;
        }
        vertex = vertex -> next ;
      }
    }
  return 0;
}
```

$\Rightarrow$ $O(n^4)$

e) void add (int u, int v, int n, struct list* list[])
{
    struct vertex* temp, *ptr;
    temp = (struct vertex) malloc (sizeof (struct v));
    temp → index = v;
    temp → next = null;
    ptr = list[u] → head;   ⇒ O(V)
    if (ptr == NULL)
        list[u] → head = temp;
    else {
        while (ptr → next != NULL)
        {
            ptr = ptr → next;
        }
        ptr → next = temp;
    }
    struct vertex* temp2;
    temp2 = (struct vertex*) malloc (sizeof (struct vertex));
    temp2 → index = u;
    temp2 → next = NULL;
    ptr = list[V] → head;
    if (ptr == NULL)
    {
        List[V] → head = temp2;
    }
    else
    {
        while (ptr → next != NULL)
            ptr = ptr → next;
        ptr → next = temp2;
    }
}

f) void delete (int n, struct list* list[], int u, int v) {
    struct vertex* ptr, *temp, *temp2;
    ptr = list[u] → head;   ⇒ O(V)
    if (ptr → index == v)
        list[u] → head = list[u] → head → next;
    else {
        while (ptr → next → index != v) {
            ptr = ptr → next;
        }
        temp = ptr → next
        ptr → next = ptr → next → next;
        free (temp);
    }
    ptr = list[v] → head;

```
if (ptr → index = = u)
    list [v] → head = list [v] → head → next ;
else {
    while (ptr → next → index != u)
        ptr = ptr → next;
    }
    temp2 = ptr → next ;
    ptr → next = ptr → next → next ;
    free (temp2);
  }
}
```

g)

```
struct list* subdevision (int n, struct list* list[], int u, int v)
{
    struct list* list2 [n+1];
    for (int i =0 ; i<n ; i++)                    → O(n)
    {
        list 2 → head = list → head ;
    }
  list2 [n] = (struct vertex*) malloc (sizeof(struct vertex));
  int w = n ;
  list2 [n] → head = NULL ;
  add (u, v, n+1, list2 ) ;
  add (u, w, n+1, list2);
  add (w, v, n+1, list2);
  return list 2;
}
```

h)

```
struct list* complement (int n, struct list* list[] )
{
  int temp [n] ;
  struct list* list2 = NULL ;
  for (int i=0; i<n ; i++)                      ⇒ O(n²)
  {
    for (int j =0 ; j<n ; j++)
        temp [j] = 1;
    temp[i] =0;
    struct vertex* ptr;
    ptr = list [i] → head;
  while (ptr != NULL)
  {
    temp [ptr → index ] =0
    ptr = ptr → next ;
  }
  for ( k=0 ; k<n ; k++)
  {
   if (temp[k] = = 1)
  {
```

```
Struct vertex* t, *t2;
t = (struct vertex*) malloc( sizeof(struct vertex));
t -> index = K;
t -> next = NULL;
t2 = list2 [i] -> head;
if (t2 = = NULL)
    list2 [i] -> head = t;
else {
    while (t2 -> next != NULL)
        t2 = t2 -> next;

    t2 -> next = t;
  }
 }
}
 return list2 ;
}
```

i)

```
int Eulerian (int n , struct list* list [])
{
  int degree ;
  for (int i=0; i<n ; i++)
  {
  degree = 0;
  struct vertex* ptr ;
  ptr = list [i] -> head ;
  while (ptr != NULL)
    {
     degree++ ;
     ptr = ptr -> next ;
    }
  if (degree = = 0 || degree % 2 != 0)
    {
     printf ("not Eulerian");
     return 0 ;
    }
  else
     continue;
  }
  printf ("Eulerian");
  return 1 ;
}
```

$\Rightarrow \underline{\underline{O(n^2)}}$

5) 22.1-6 in CIRS page. 593

Ans) if vertex K is a universal sink then row K in adjacency matrix is all 0's and column K is all 1's except for position (K, K) which is a 0.

Lets start from (1, 1) in matrix, If in examining position (i, j), if a 1 is encountered examine (i+1, j), if 0 is examined examine (i, j+1) once either i or j is equal to |V|, terminate.

Let graph be a universal sink with vertex i. Once vertex K is hit algorithm will continue to increment j until j = |V|. To be sure row K is eventually hit, note that once column K is reached, algorithm continue to increment i until it reaches K.

This algorithm run in O(V) & checking whether i corresponds to sink or not is done in O(V). Therefore entire process takes O(V).