Introduction to Graph Algorithms

Given a graph $G(V, E)$, it can be represented in two ways

① Adjacency-List Representation

Consists of

- An array A of $|V|$ lists, one for each vertex in V
- $\forall u \in V$, $A[u]$ contains pointers to all vertices v such that $(u, v) \in E$
- The vertices in A are typically stored in an arbitrary order

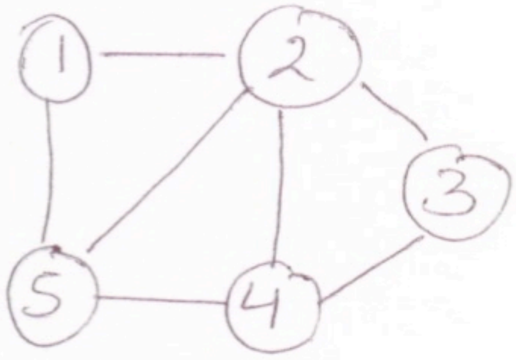
② Adjacency-Matrix Representation

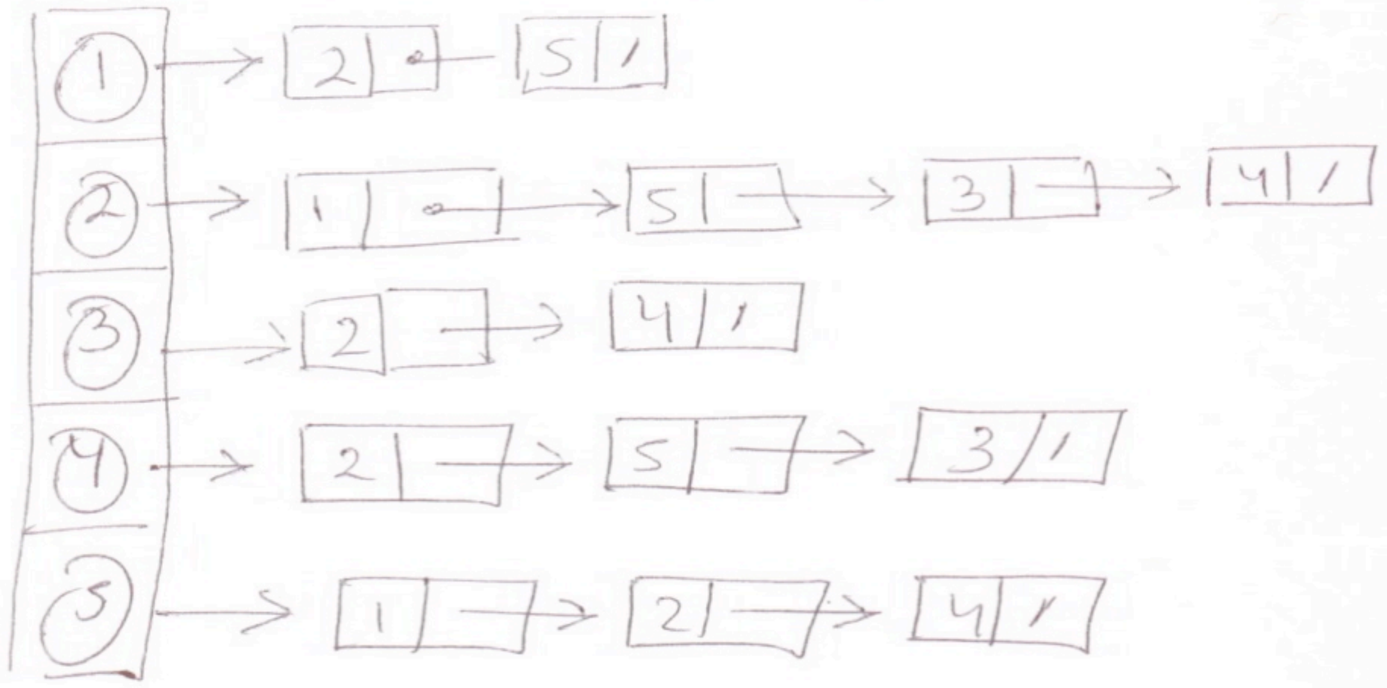
Given $G(V, E)$, assume numbering of vertices as $1, 2, \ldots |V|$ in an arbitrary manner

The adjacency matrix representation is given by a matrix ~~Adj~~ $Adj = (a_{ij})$ of size $|V| \times |V|$:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Example

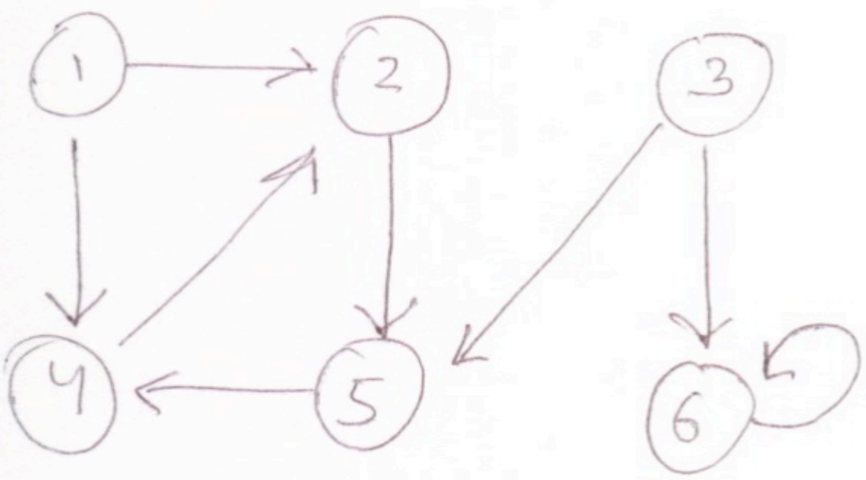


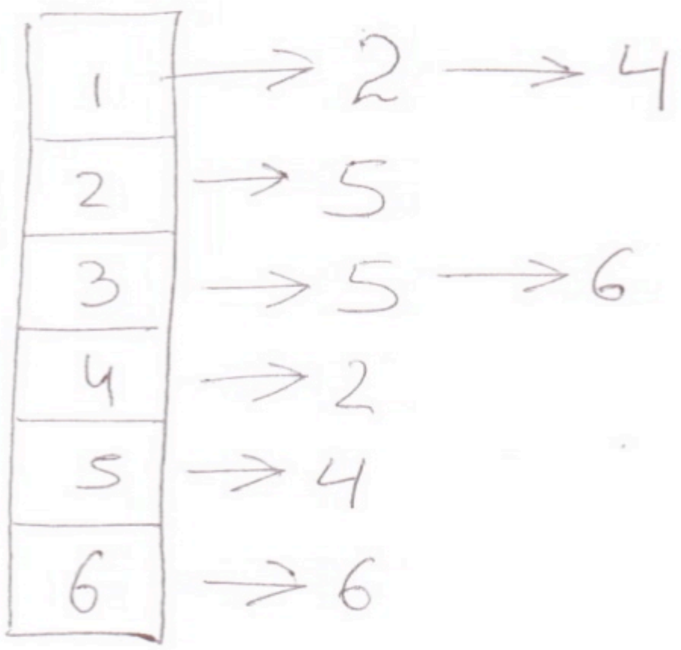


Adjacency List

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	1
4	0	1	1	0	1
5	1	1	0	1	0

For a directed graph





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Properties

- IF G is a directed graph
  - Sum of lengths of all adjacency lists is $|E|$

- IF G is undirected graph
  - Sum is $2|E|$

- Amount of memory required by the adjacency list is $O(V+E)$

- Search presence of edge (u, v) in Adjacency list representation requires (assuming we know, $u, v \in V$)
  - searching the adjacency list of u (or v)

- In an adjacency matrix this can be done in $O(1)$ time

- adjacency matrix requires $\Theta(V^2)$ memory. independent of the number of edges.

- adjacency matorex is wasteful for spursely connected graphs.

Questions

Defn
In-degree : Number of edges pointing to a given vertex
Out-degree : Number of edges pointing out of a given vertex.

① Given a directed graph $G(V, E)$ and its adjacency list representation

① How long does it take to compute the out-degree of every vertex
$O(E)$

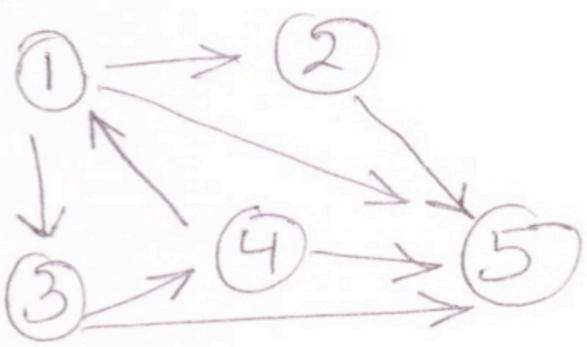
② Compute the in-degree of every vertex.
- Simple : in-degree of 1 vertex is $O(E)$
of V vertices $O(VE)$
- Refined : Maintain an array of size $|V|$
Traverse the adj-list representation and update the array whenever the correspond vertex is in the adjacency list of any vertex.
$O(E)$

Question

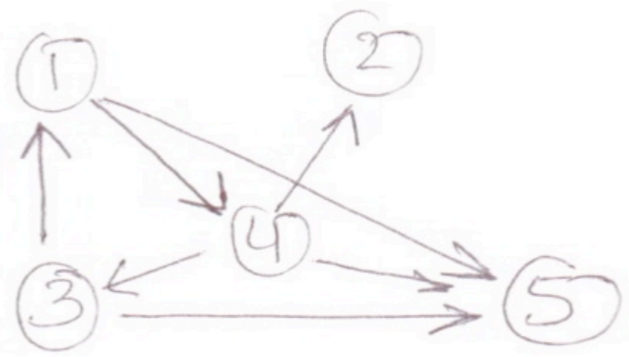
Computing the square of a directed graph $G(V, E)$ is computing the square of $G^2 = (V, E^2)$:

$(u, w) \in E^2$ iff for some $v \in V$, both $(u, v) \in E$ and $(v, w) \in E$.

That is G^2 contains an edge between u and w whenever G has a path with exactly two edges between u and w



$G(V, E)$



$G^2(V, E^2)$

Give an efficient algorithm for computing G^2 from G for both adjacency list and adjacency matrix representations.

For each vertex v in A or adj
  do For each element e in the connectivity list of v
    do scan the connectivity list of e
      update E^2

$O(V^3)$ for connectivity matrix

$O(V) + O(E) + O(E)$?

Breadth-First Search (BFS)

- One of the most fundamental traversal algorithms
  - Used in shortest path algorithms
  - Used in MST algorithms

Idea: Given $G(V, E)$ and a "source" vertex s
BFS searches G to find every vertex reachable from s. It also computes the shortest path (in terms of number of edges) from s to all reachable vertices. It represents this information in a "breadth-first tree" rooted at s.
BFS works on both directed and undirected graphs.

(a) The algorithm is called "breadth-first" because it discovers all vertices at distance k from s before discovering any vertex at distance $k+1$

(b) BFS uses a coloring scheme for each vertex to track progress (W, G, B): White, Grays, Black
  - All vertices start with color W (White)
  - When a vertex is encountered for the first time it is considered "discovered"
    Once a vertex is discovered, it becomes non-white.

- G and B vertices are therefore discovered.

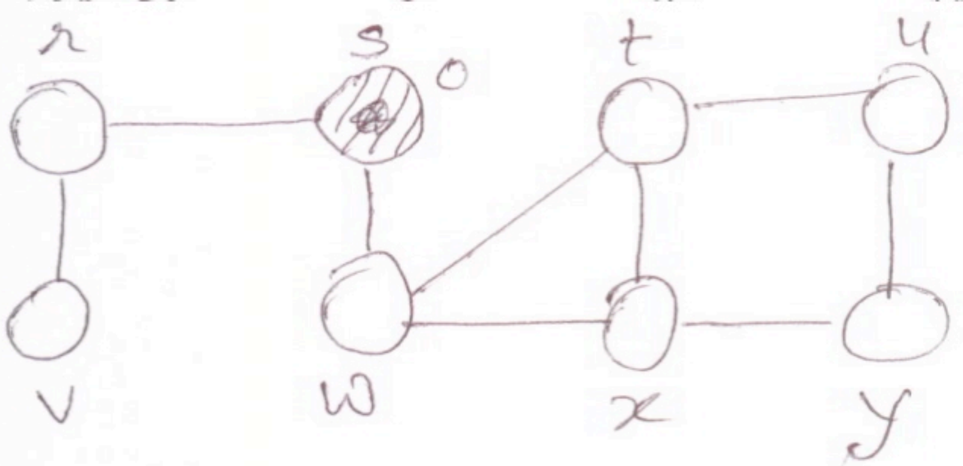
- To ensure that the search progresses in a breadth-first manner, G and B vertices are distinguished:

- If $(u, v) \in E$ and u is black, then v is either gray or black.
    - All vertices adjacent to B-vertices have been discoverd.
    - G-vertices may have some W-vertices as neighbors and represent the boundary between discovered and undiscovered vertices

- BFS constructs a breadth-first tree, initially containing only s. When a white vertex v is discovered, ~~the edge~~ while scanning the adjacency list of a discovered vertex u, the vertex v and edge (u, v) are added to the tree.
    - u is called the predecessor of v
    - Similarly: If u is on a path from s to v
        - u is an ancestor of v
        - v is a descendant of u

- Let $G(V, E)$ be represented using adjacency list.
- Let the color of each vertex u be stored in $color[u]$ $u \in V$
- Let the predecessor of u be stored in $\pi[u]$. If u has no predecessor, then $\pi[u] = NIL$.
- The distance from source s to vertex u is stored in $d[u]$
- The algorithm also uses a FIFO queue (Q) to manage the gray vertices

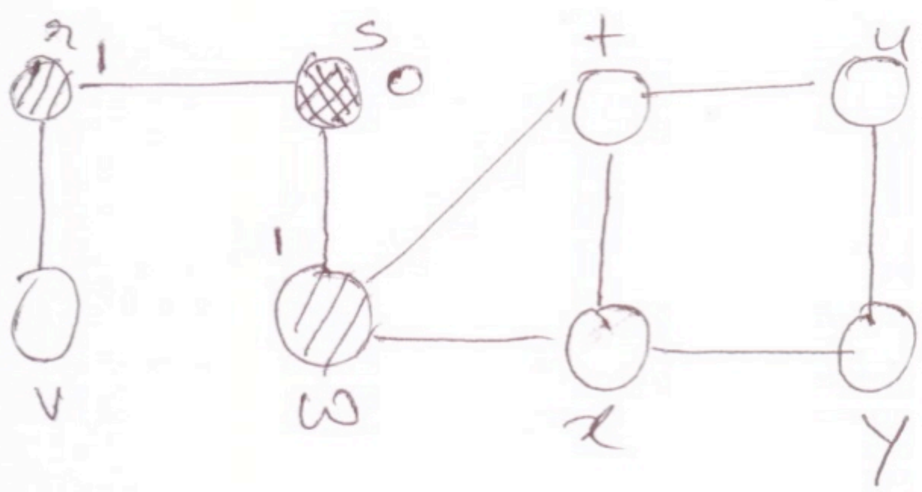
```
BFS(G, s)
LINE#
1     For each vertex u ∈ V[G] - {s}                ]  Paint every vertex white
2          do color[u] ← W                          ]  Initialize d[u] and π[u]
3             d[u] ← ∞                              ]
4             π[u] ← NIL                            ]
5     color[s] ← G                                  ]  s is gray.
6     d[s] ← 0                                      ]
7     π[s] ← NIL                                    ]
8     Q ← {s}                                       ]  Start with s in the Q
9     while Q ≠ 0                                   ]  Hence for in all vertices in
10         do u ← head[Q]                           ]  Q will be gray.
11            for each v ∈ Adj[u]                   ]
12               do if color[v] = W                 ]  The loop continues
13                  then color[v] ← G               ]  till there are vertices
14                       d[v] ← d[u] + 1            ]  (Gray) which have not
15                       π[v] ← u                   ]  had their adjacency
16                       Enqueue(Q, v)              ]  lists examined.
17            Dequeue(Q)
```



$Q = \{s\}$

$d[s] = 0$

$\pi[s] = NIL$

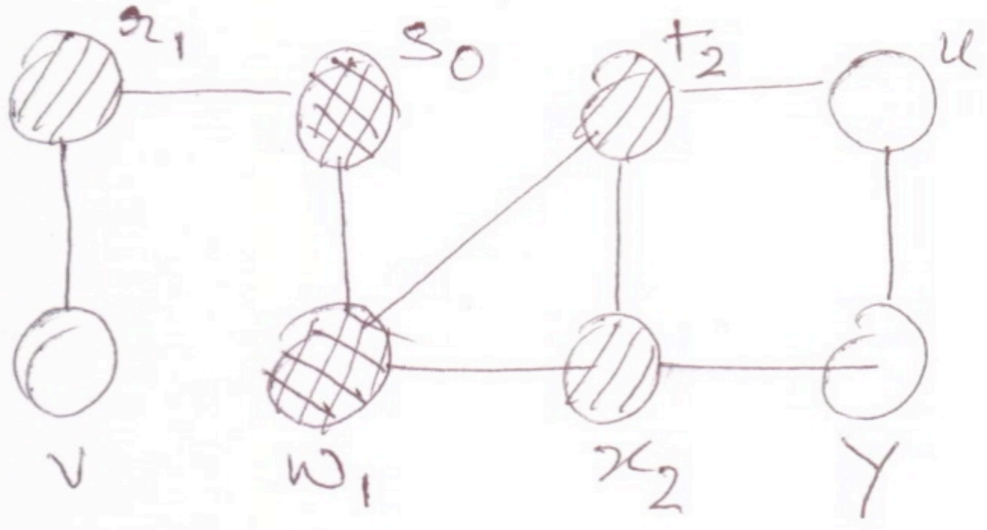


head = s

$Q = \{w, r\}$, ~~head = s~~

$\pi(r) = s$

$\pi(w) = s$

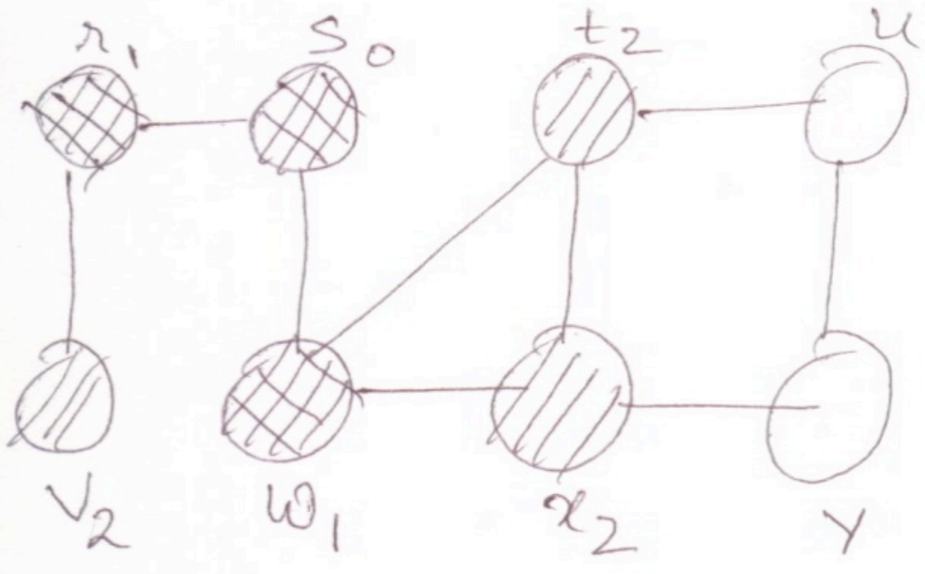


head = w

$Q = \{r, t, x\}$ ~~head = w~~

$\pi(t) = w$

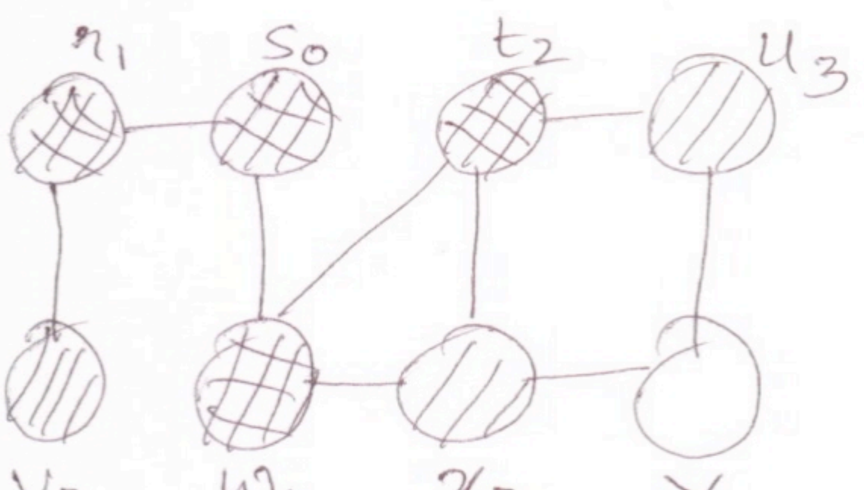
$\pi(x) = w$



head = r

$\pi(v) = r$

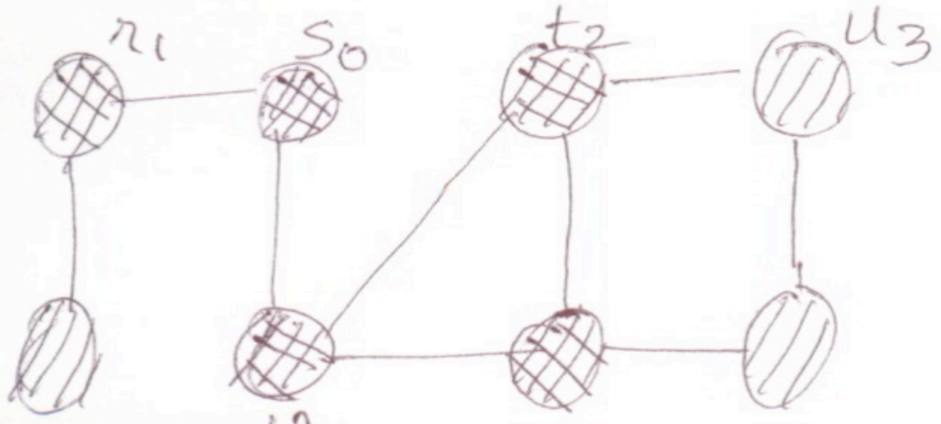
$Q \{r, t, x, v\}$



head = t

$Q = \{x, v, u\}$

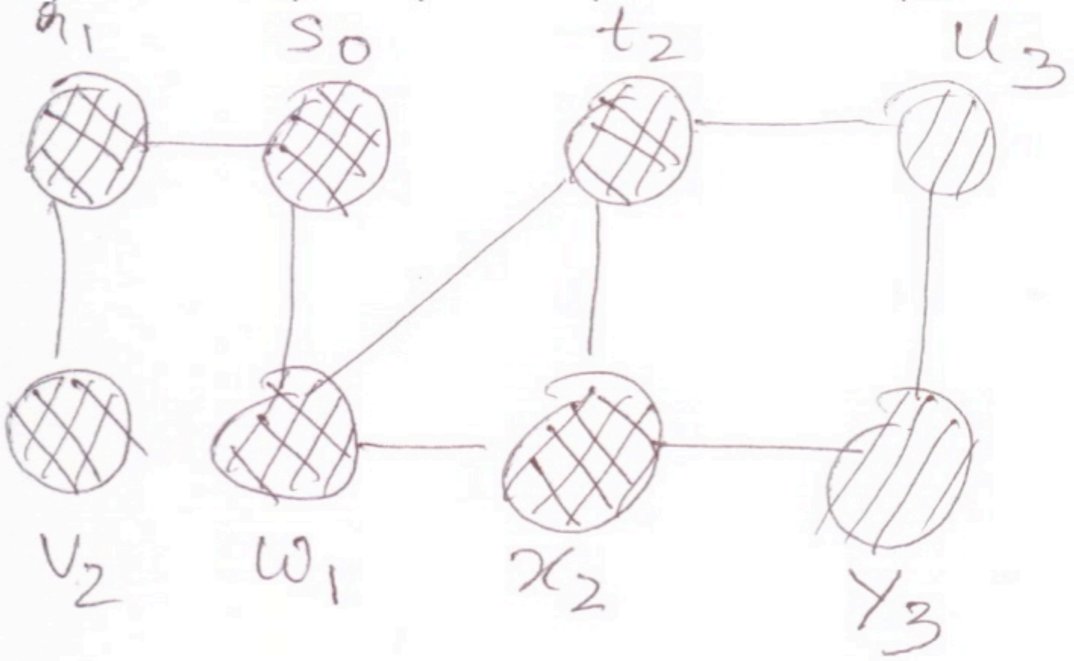
$\pi(u) = t$



head = x

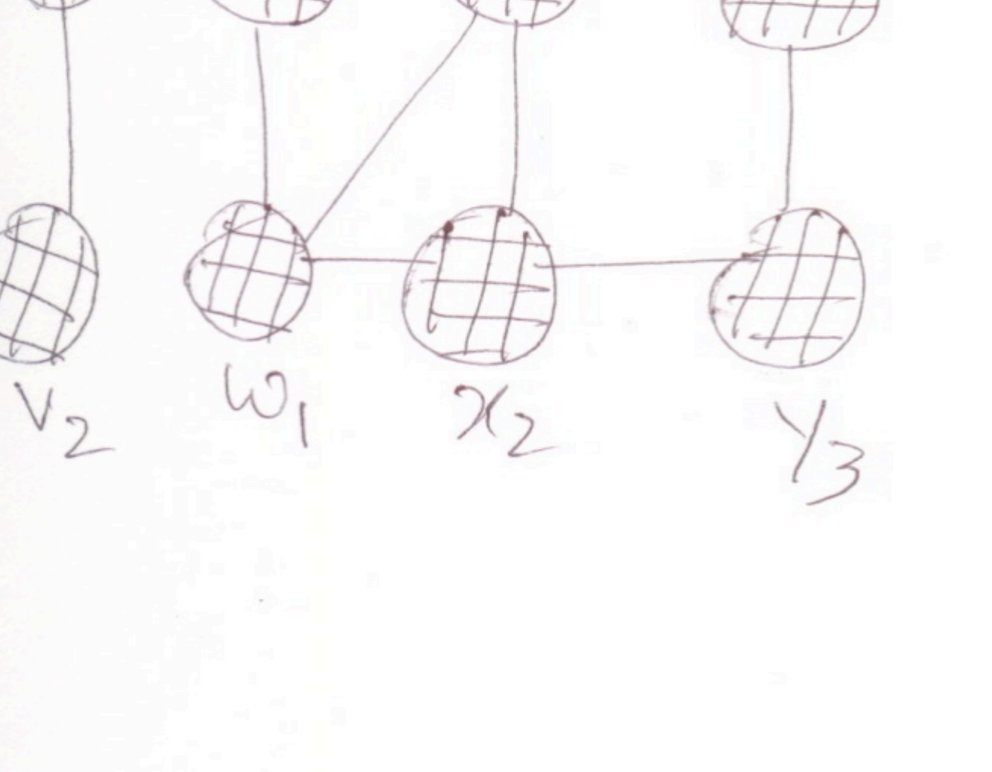
$Q \{v, u, y\}$

$\pi(y) = x$



head = v

$Q = \{u_3, y_3\}$

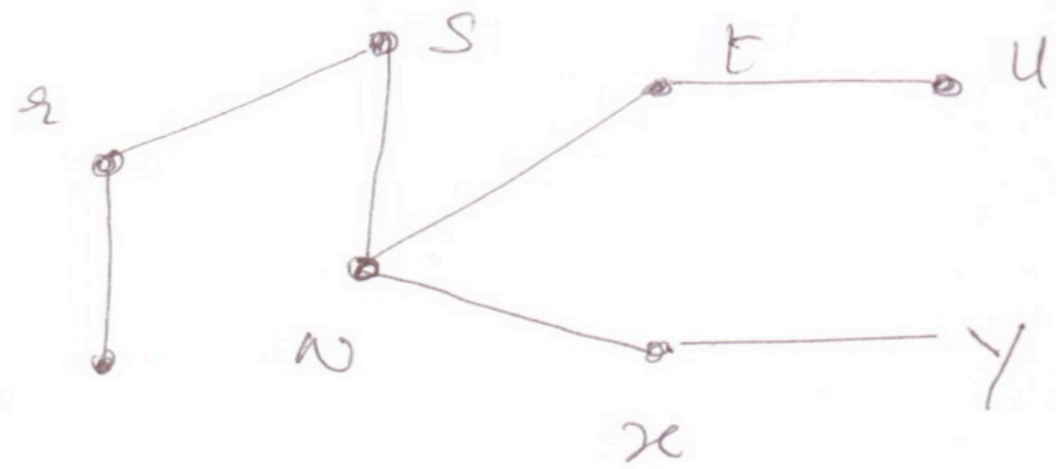


head = u

$Q = \{y_3\}$

head = u

$Q = \{\}$.



Analysis

- After initialization, no vertex is ever whitened
- The test in line # 12 thus ensures that every vertex is enqueued once and (therefore dequeued once).

  Enqueue and dequeue is $O(1)$
  so queue operations is $O(V)$

- Adjacency list of a vertex is scanned only when it is dequeued.

  Sum of lengths of all adjacency lists is $O(E)$
  Therefore total time spent in scanning adjacency lists is $O(E)$

- Initialization is $O(V)$
- Total : $O(V+E)$