04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

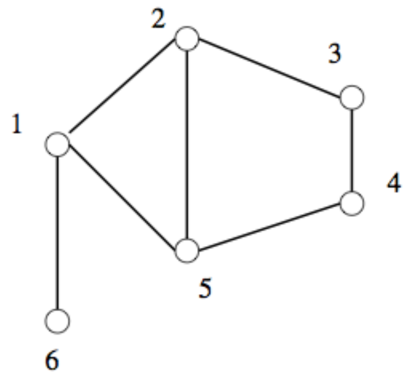
Lecture 19

Graphs

- Types of graph
- Adjacency matrix representation
- Adjacency list representation
- Breadth-First Search (BFS) traversal
- Application of BFS
- Depth-First Search (DFS) traversal
- Topological Sorting
- Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's algorithm
- Shortest Path Algorithms
  - Dijkstra's algorithm
  - Floyd's algorithm

Traversing a Graph

- Visit every vertex and edge in a systematic way



- Key idea: mark each vertex when we first visit it & keep track of what we have not yet completely explored

- Each vertex will exist in one of three states
  1. Undiscovered – the vertex is in its initial untouched state
  2. Discovered – the vertex has been found, but we have not yet processed all its edges
  3. Processed – the vertex after we have visited all its edges

Traversing a Graph

- Keep a record of all the vertices discovered but not yet completely processed
- Begin with a starting vertex
- Explore each vertex
  - Evaluate each edge leaving it
  - If the edge goes to an undiscovered vertex
    - Mark it discovered
    - Add it to the list of work to do
  - If the edge goes to a processed vertex, ignore it
  - If the edge goes to a discovered unprocessed vertex, ignore it

Traversing a Graph

- There are two primary graph traversal algorithms
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- The difference is the order in which they explore vertices

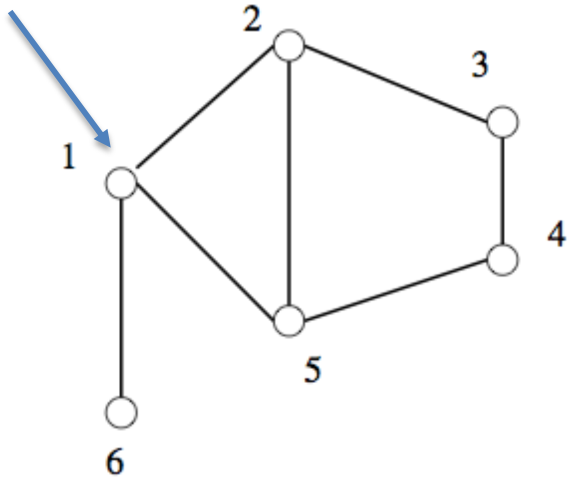
Traversing a Graph

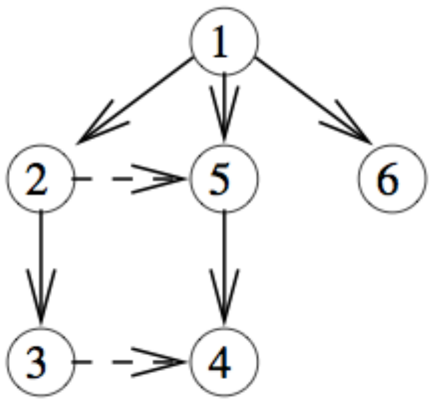
- The order depends completely on the container data structure used to store the discovered but not processed vertices
  - BFS uses a **queue**
    - By storing the vertices in a FIFO queue, we explore the **oldest** unexplored vertices first
    - Thus explorations radiate out slowly from the starting vertex
  - DFS uses a **stack**
    - By storing the vertices in a LIFO stack, we explore the vertices by diving down a path, visiting a new neighbour if one is available, and backing up only when we are surrounded by (i.e. connected by edges to) previously discovered vertices
    - Thus explorations quickly wander away from out starting vertex

Traversing a Graph

Breadth-first search (BFS)

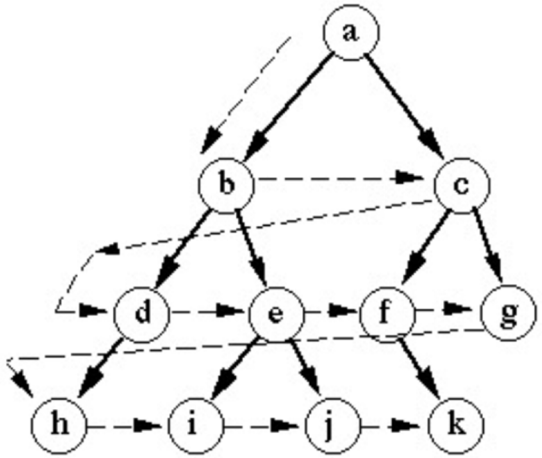
Start at node 1





Traversing a Graph

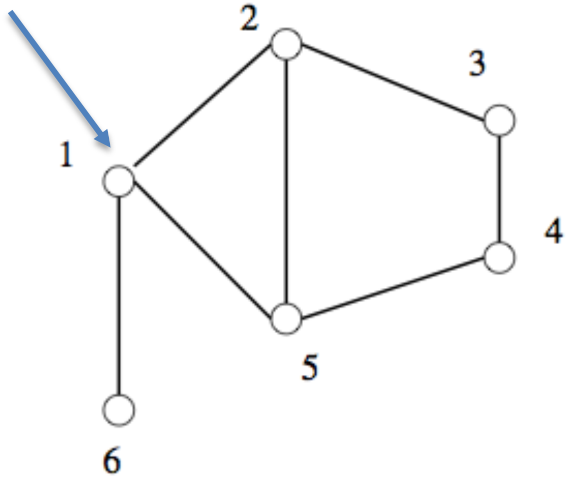
Breadth-first search (BFS)

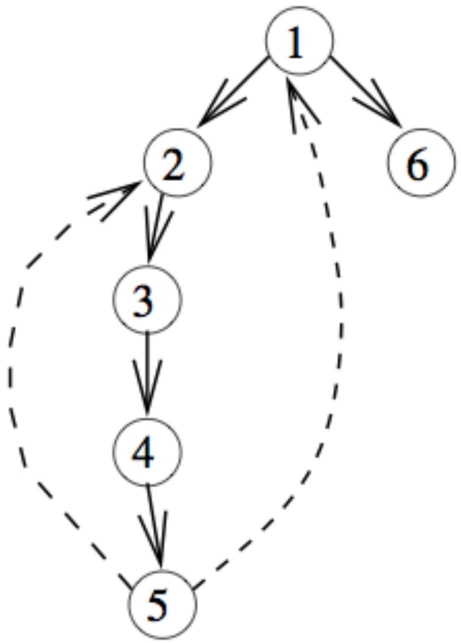


Traversing a Graph

Depth-first search (DFS)

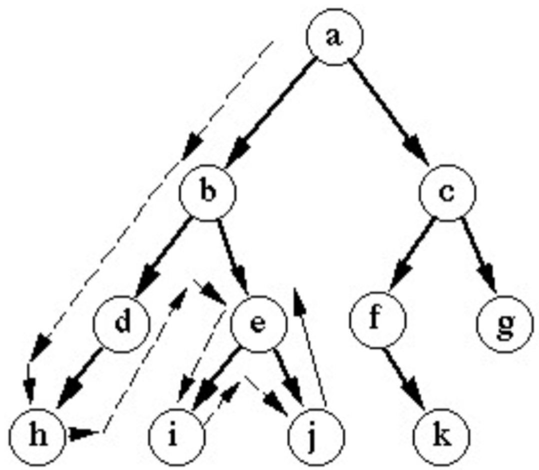
Start at node 1





Traversing a Graph

Depth-first search (DFS)



Breadth-First Search

- Assign a direction to each edge, from discoverer vertex u to discovered vertex v
- Since each node has exactly one parent, except for the root (i.e. start vertex), this defines a tree on the vertices of the graph
- This tree defines the shortest path from the root to every other node in the tree
- This makes the BFS very useful for in shortest path problems (in unweighted graphs)

Breadth-First Search

BFS(G, s)

for each vertex $u \in V[G] - \{s\}$ do

  $state[u]$ = "undiscovered"

  $p[u] = nil$, i.e. no parent is in the BFS tree

$state[s]$ = "discovered"

$p[s] = nil$

$Q = \{s\}$

while $Q \neq \emptyset$ do

  u = dequeue$[Q]$

  process vertex u as desired

  for each $v \in Adj[u]$ do

    process edge (u, v) as desired

    if $state[v]$ = "undiscovered" then

      $state[v]$ = "discovered"

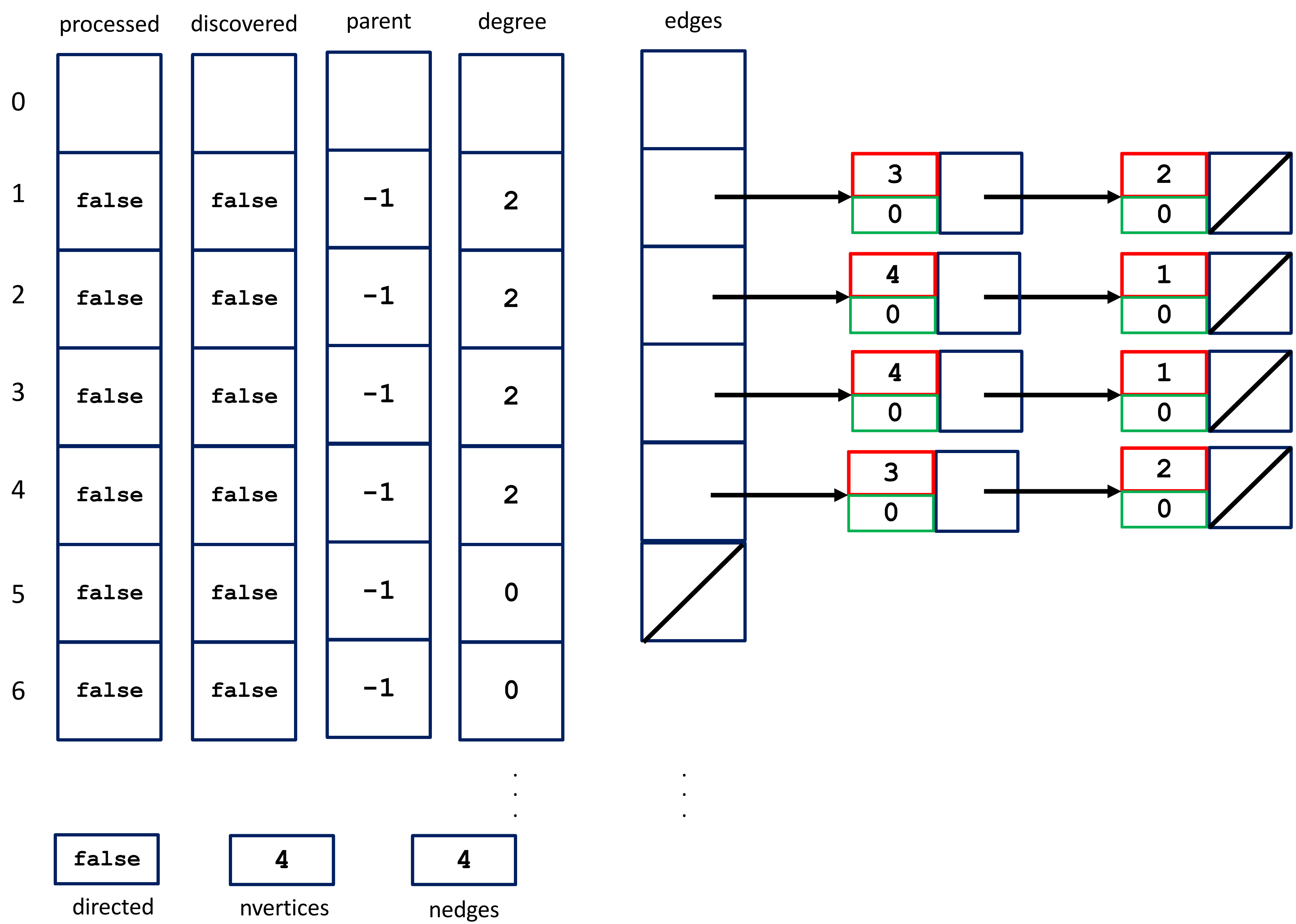
      $p[v] = u$

      enqueue$[Q, v]$

  $state[u]$ = "processed"

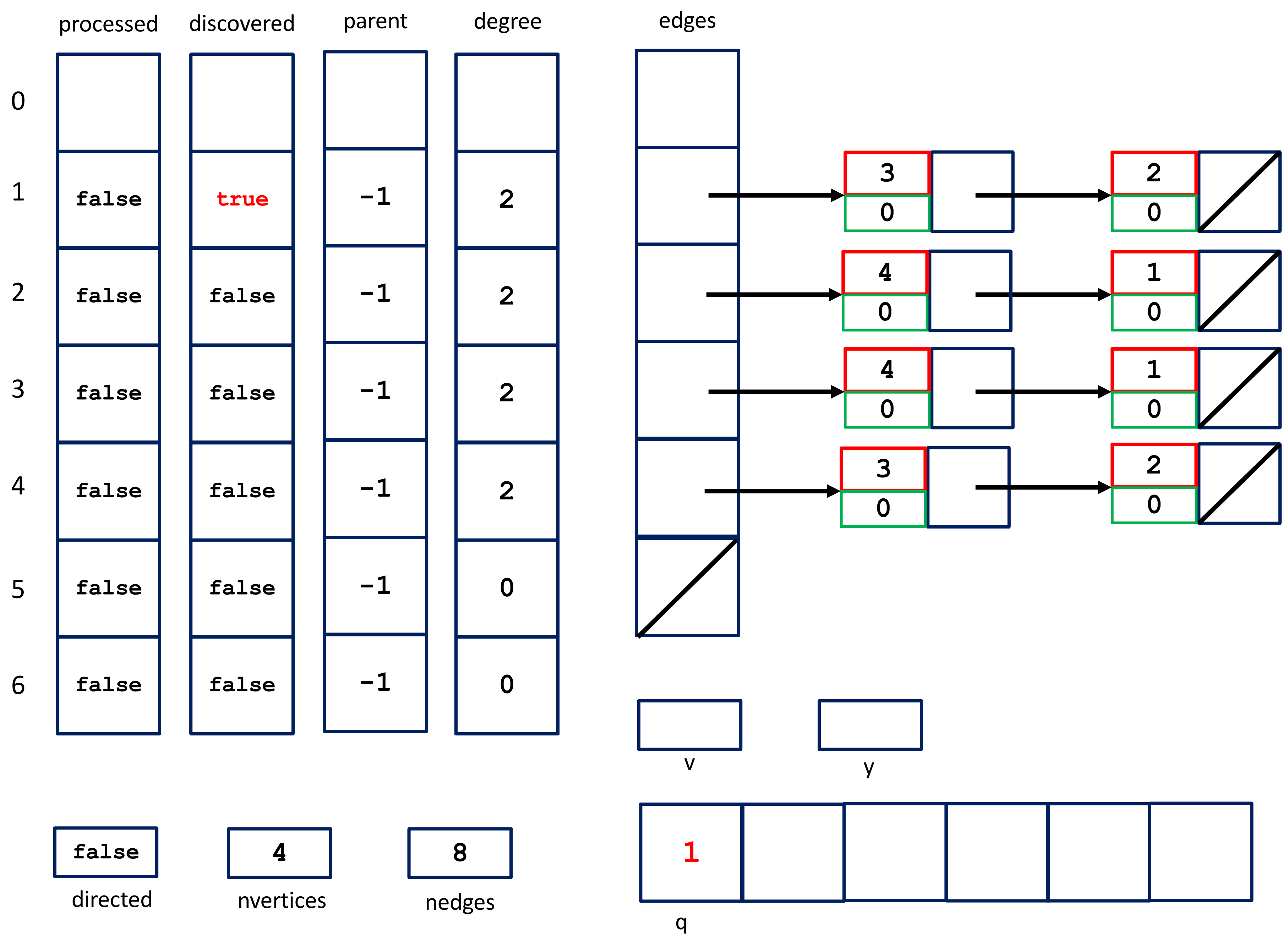
Breadth-First Search

```
/* Breadth-First Search                                        */

bool processed[MAXV+1];    /* which vertices have been processed */
bool discovered[MAXV+1];   /* which vertices have been found */
int parent[MAXV+1];        /* discovery relation */


/* Each vertex is initialized as undiscovered:                 */

initialize_search(graph *g){

   int i;                           /* counter */

   for (i=1; i<=g->nvertices; i++) {
      processed[i] = discovered[i] = false;
      parent[i] = -1;
   }
}
```


processed
discovered
parent
degree
edges
0
1
2
3
4
5
6
false
false
-1
2
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
false
directed
4
nvertices
4
nedges

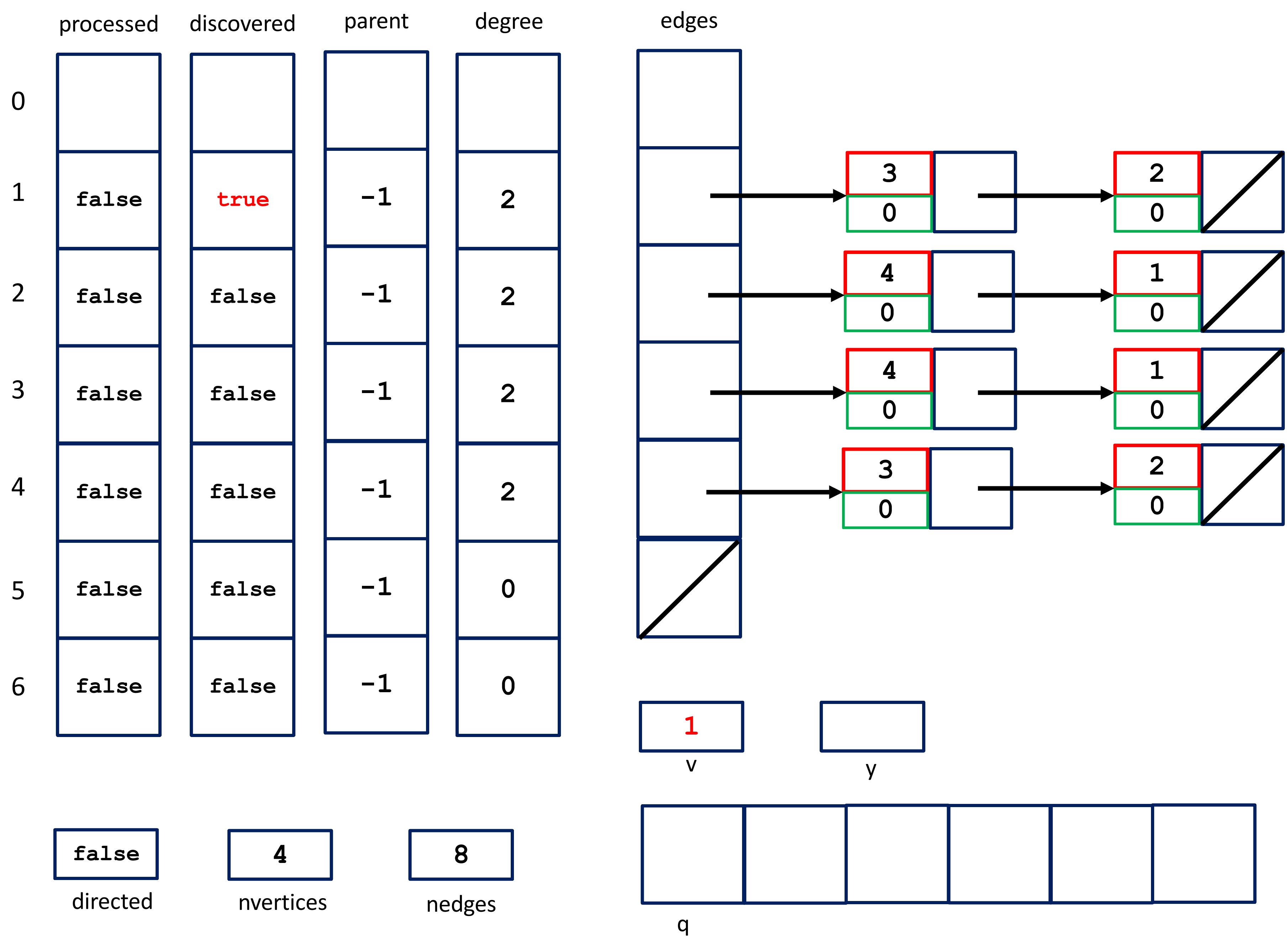
Breadth-First Search

```
/* Once a vertex is discovered, it is placed on a queue.         */
/* Since we process these vertices in first-in, first-out order,  */
/* the oldest vertices are expanded first, which are exactly those */
/* closest to the root                                            */

bfs(graph *g, int start)
{
   queue q;                      /* queue of vertices to visit */
   int v;                        /* current vertex             */
   int y;                        /* successor vertex           */
   edgenode *p;                  /* temporary pointer          */

   init_queue(&q);
   enqueue(&q,start);
   discovered[start] = true;
```


processed
discovered
parent
degree
edges
0
1
2
3
4
5
6
false
true
false
-1
2
0
3
4
1
v
y
q
false
directed
4
nvertices
8
nedges
1

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

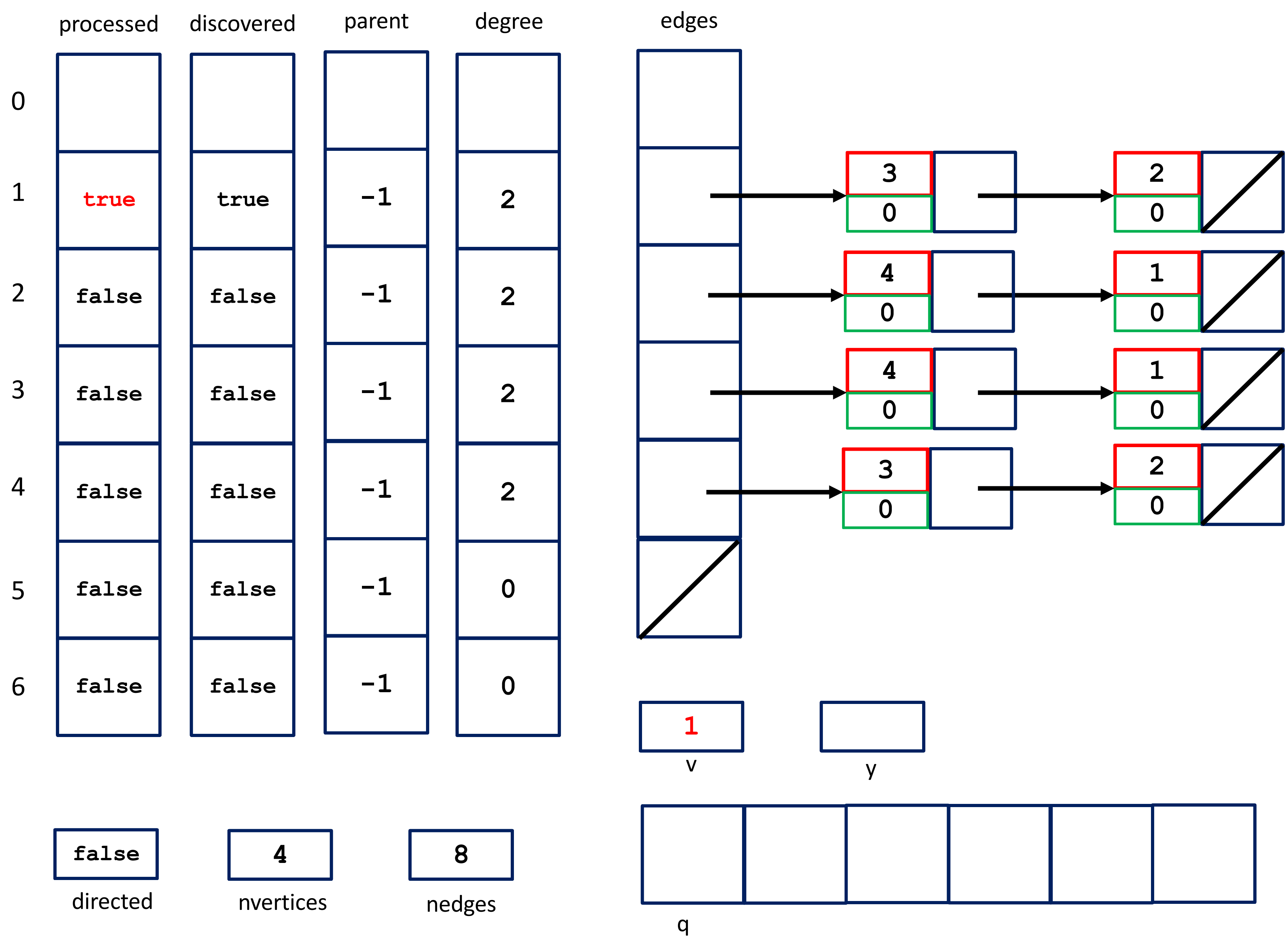

processed
discovered
parent
degree
edges
0
1
2
3
4
5
6
false
true
-1
2
3
0
2
4
1
0
v
y
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

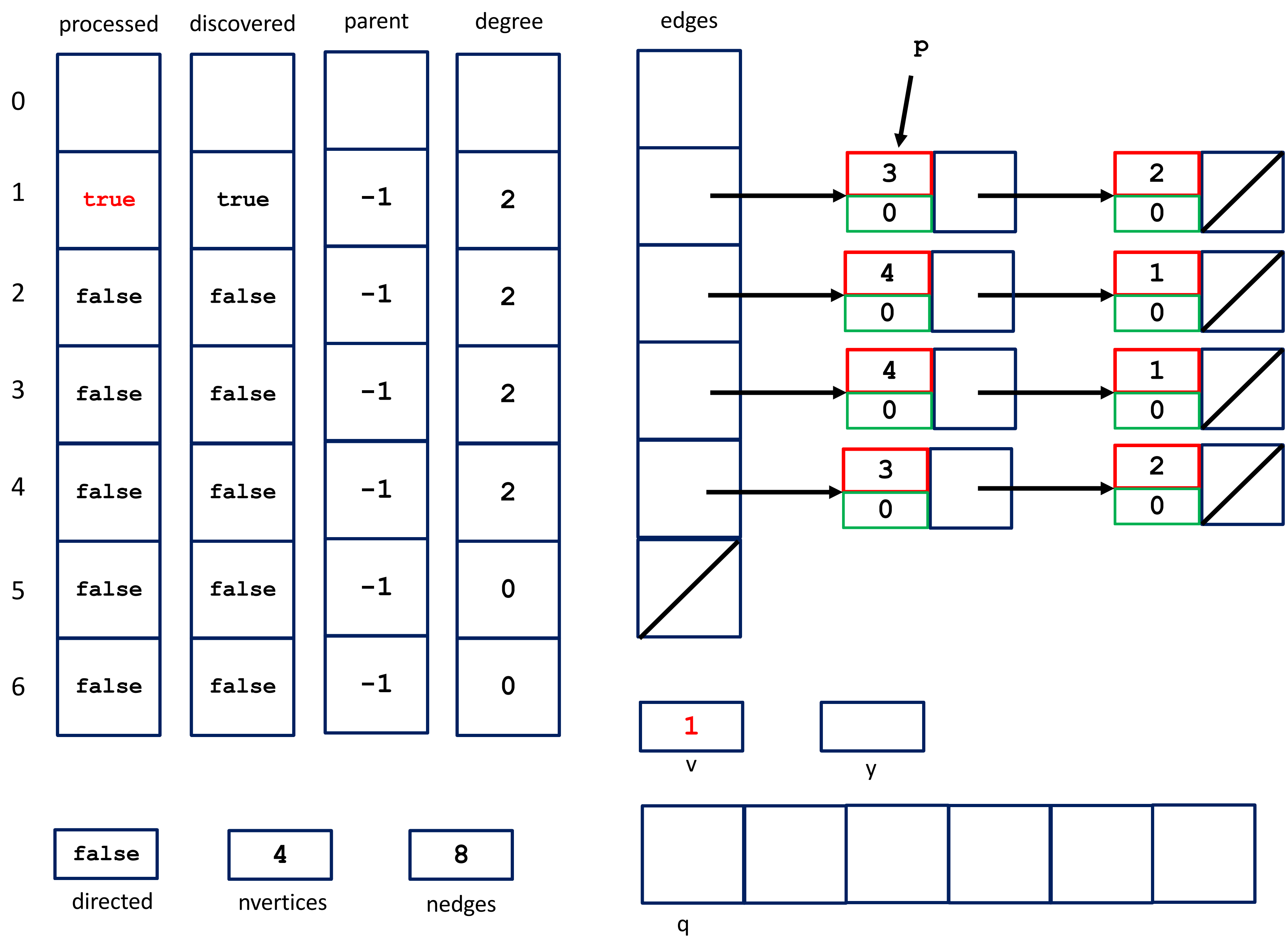
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
0
1
2
3
4
5
6
true
true
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
y
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

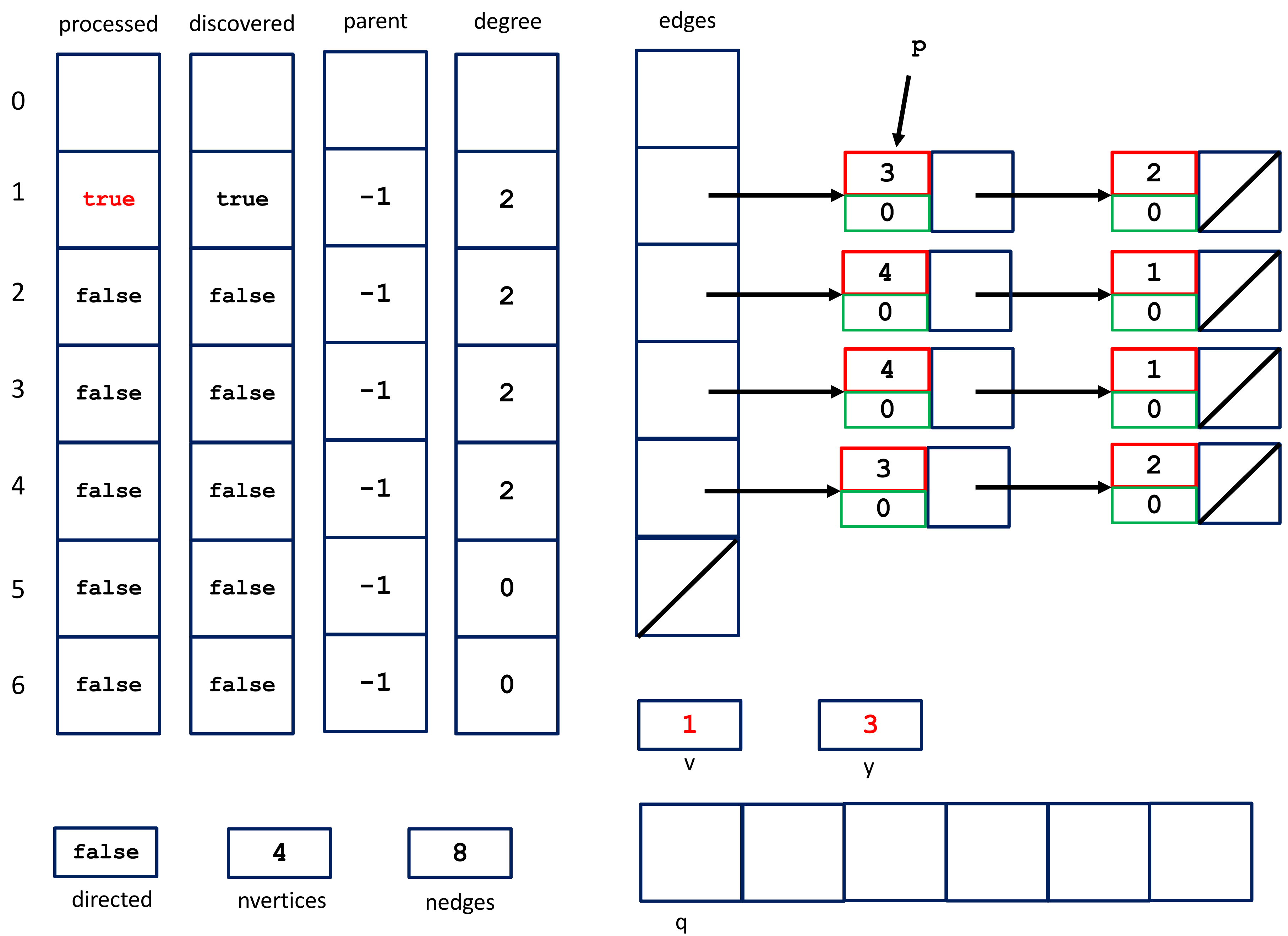
```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```



processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
true
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
y
false
directed
4
nvertices
8
nedges
q

Breadth-First Search

```
  while (empty_queue(&q) == FALSE) {
    v = dequeue(&q);
    process_vertex_early(v);
    processed[v] = TRUE;
    p = g->edges[v];

    while (p != NULL) {
      y = p->y;
      if ((processed[y] == FALSE) || g->directed)
        process_edge(v,y);
      if (discovered[y] == FALSE) {
        enqueue(&q,y);
        discovered[y] = TRUE;
        parent[y] = v;
      }
      p = p->next;
    }
    process_vertex_late(v);
  }
}
```



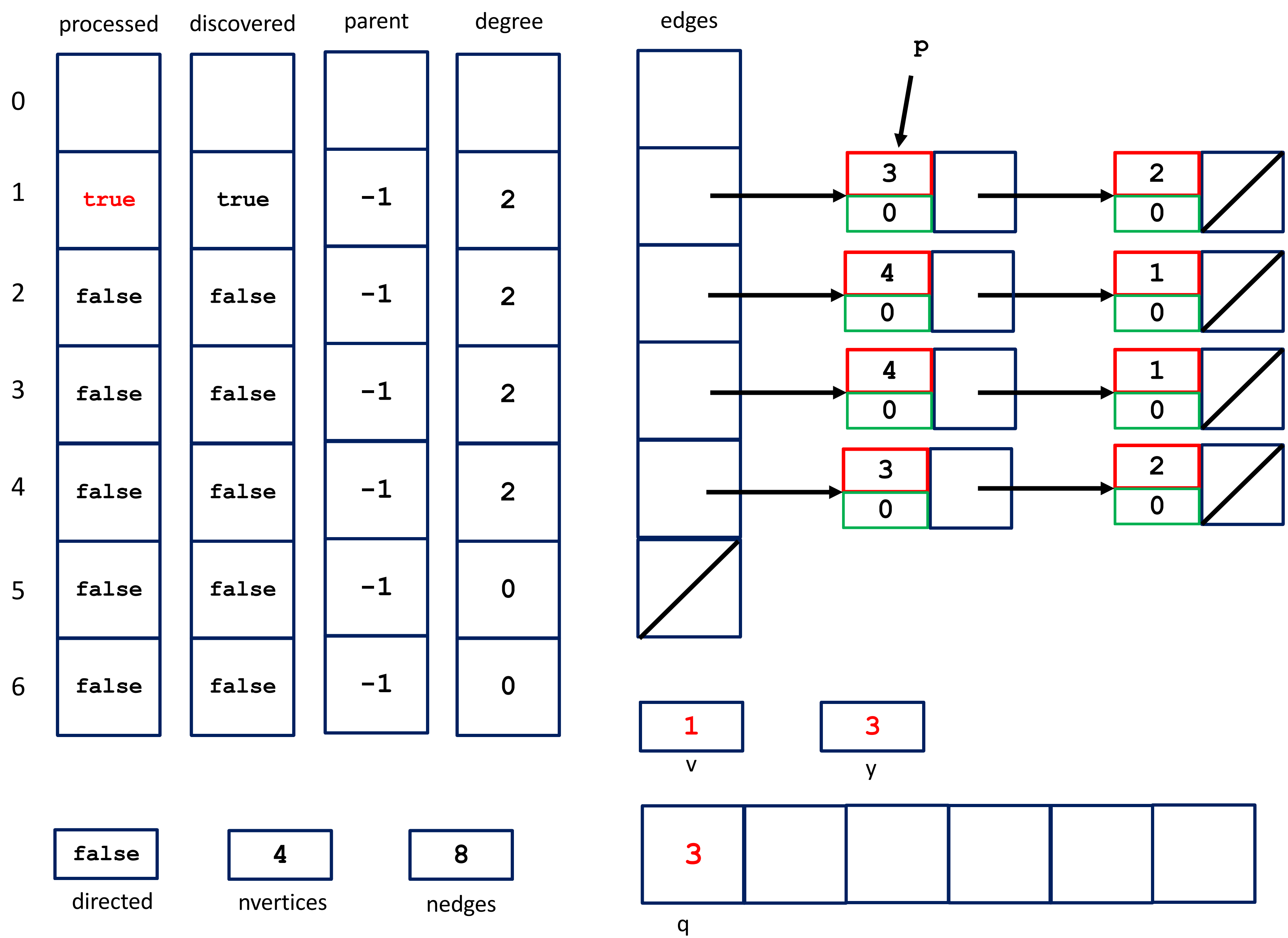
processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
false
false
false
false
false
-1
-1
-1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
3
y
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

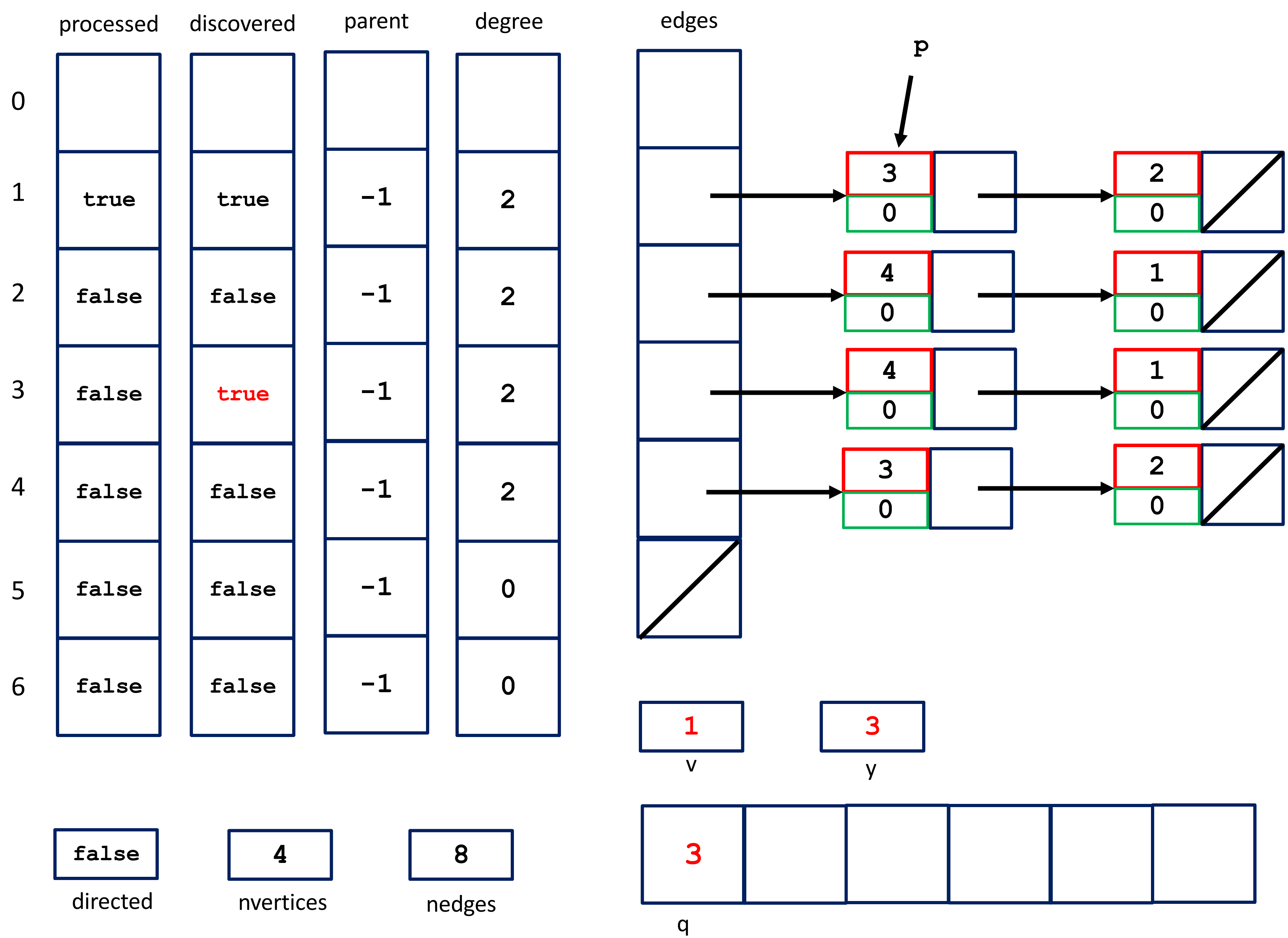
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
true
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
3
y
false
directed
4
nvertices
8
nedges
3
q

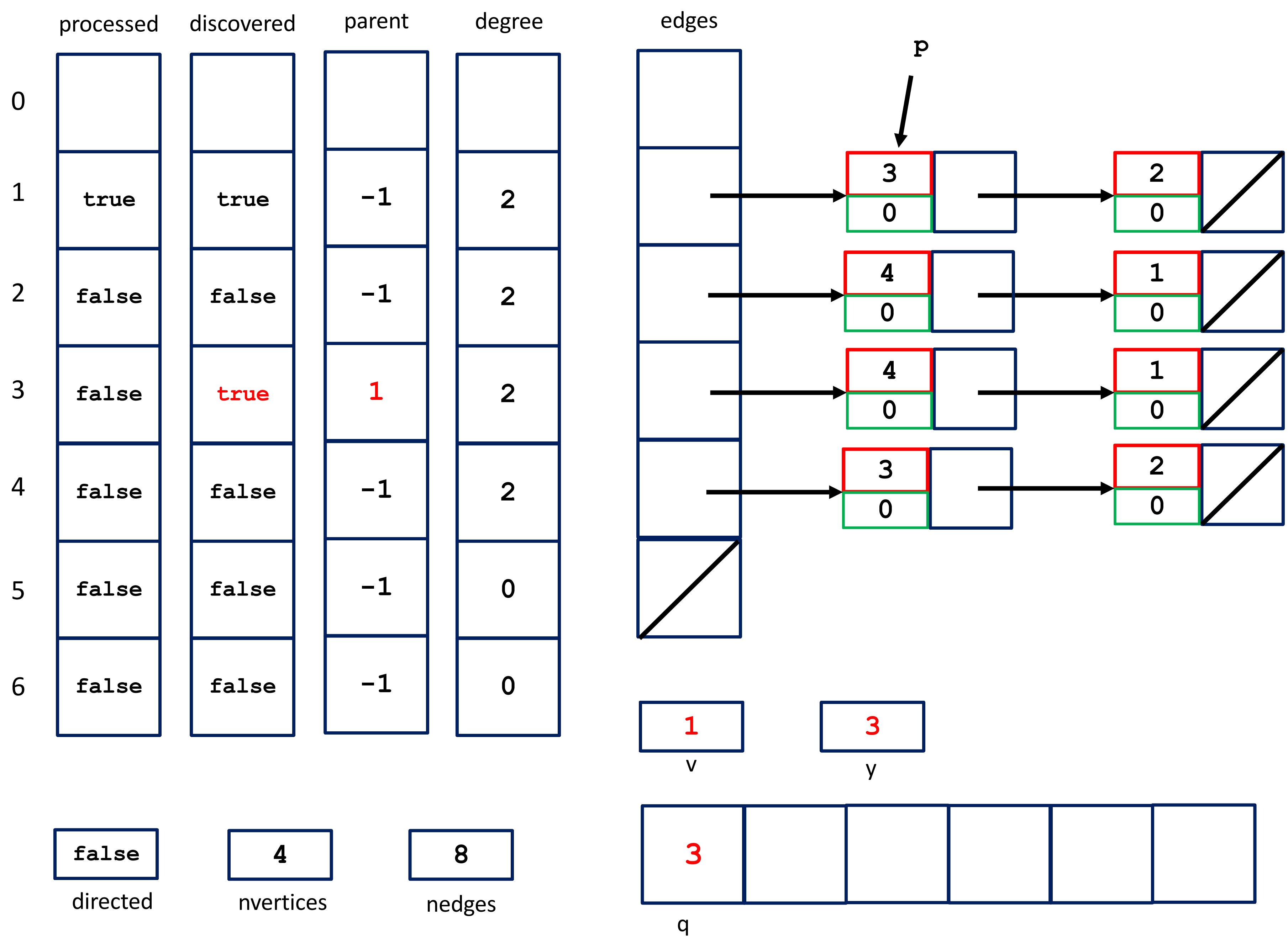
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
true
-1
2
false
false
-1
2
false
true
-1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
3
y
3
q
false
directed
4
nvertices
8
nedges

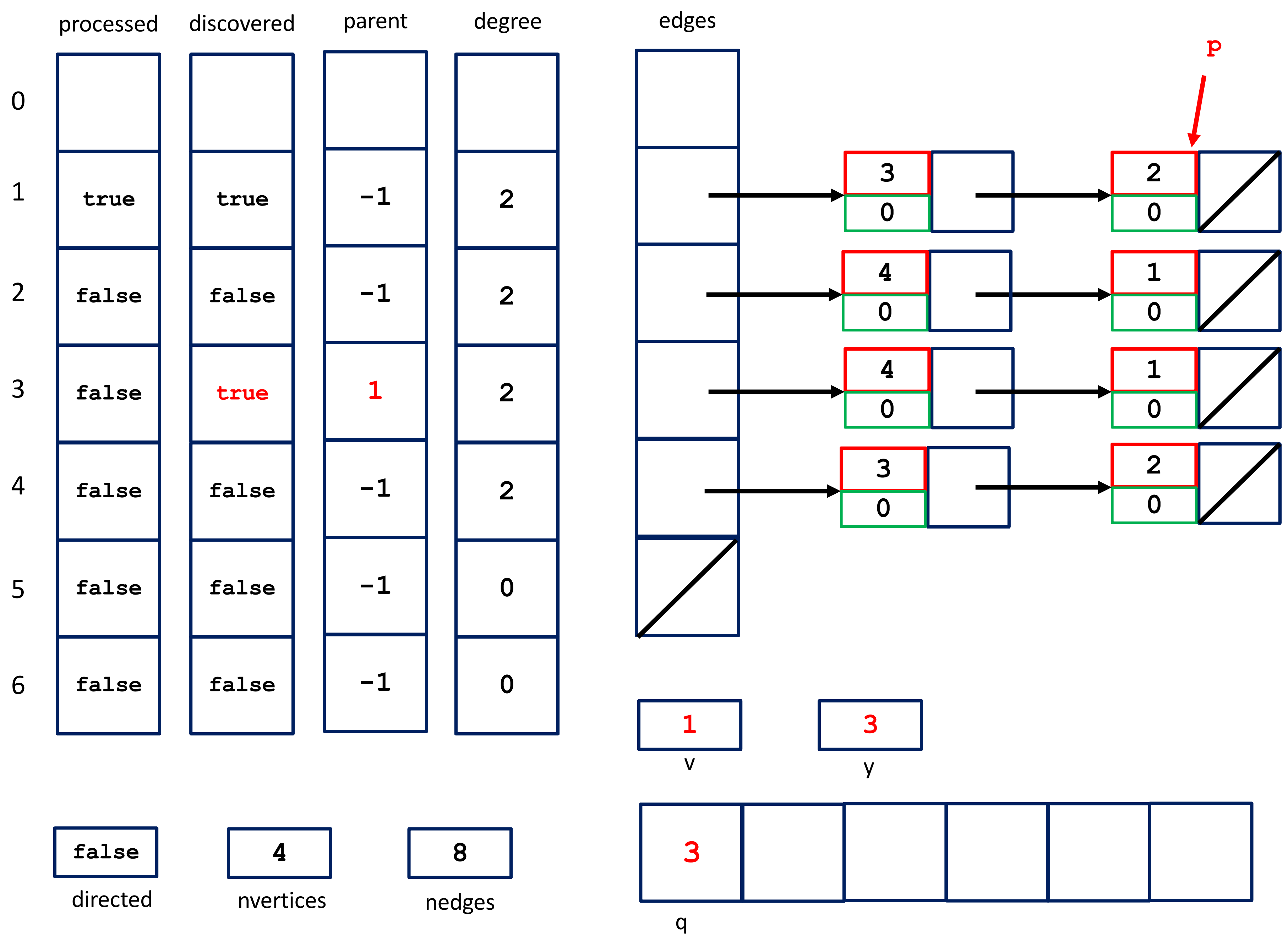
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
false
true
false
false
false
-1
-1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
3
y
3
q
false
directed
4
nvertices
8
nedges

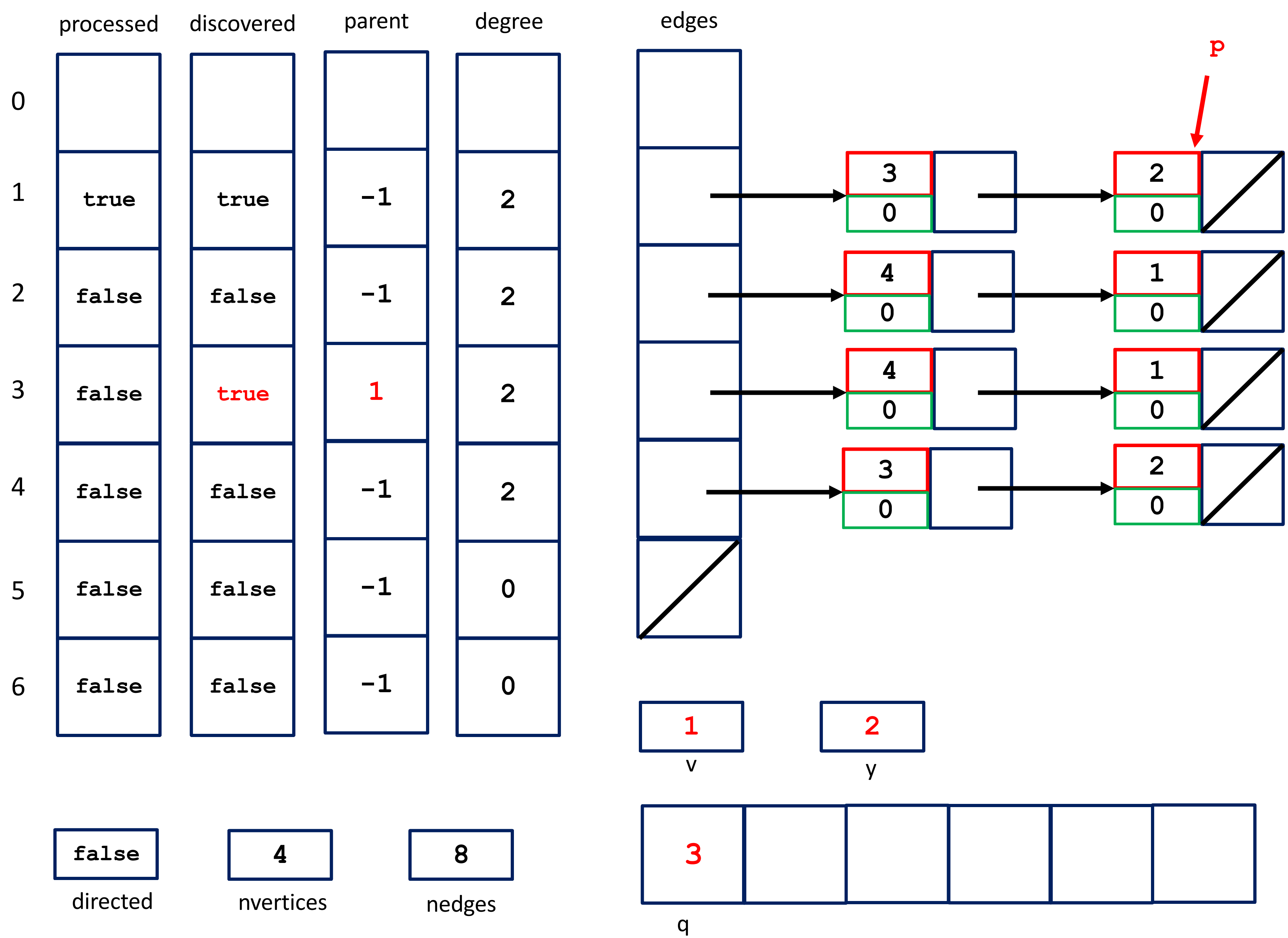
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
false
true
false
false
false
-1
-1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
3
y
3
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

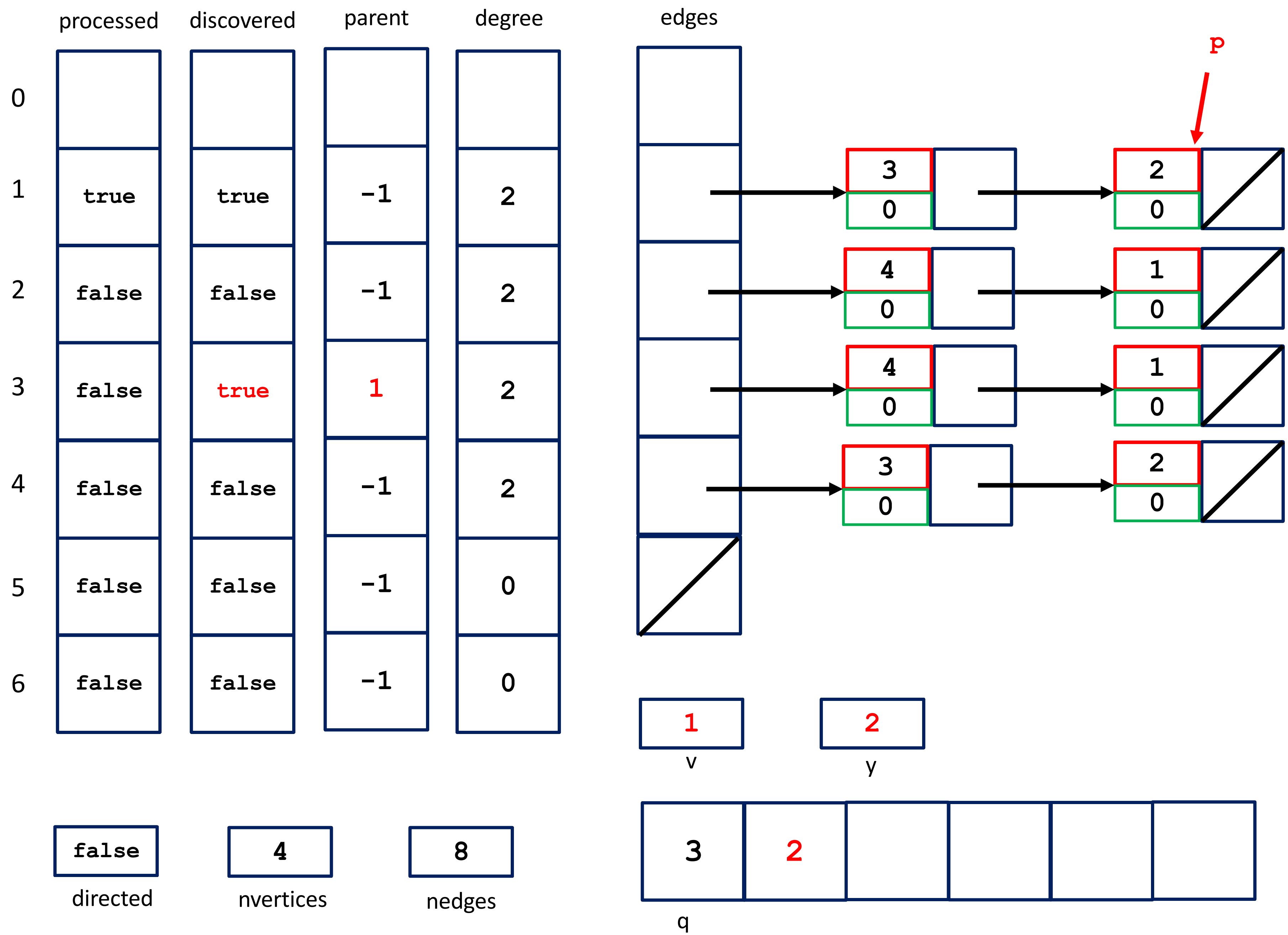

processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
false
true
false
false
false
-1
-1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
3
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

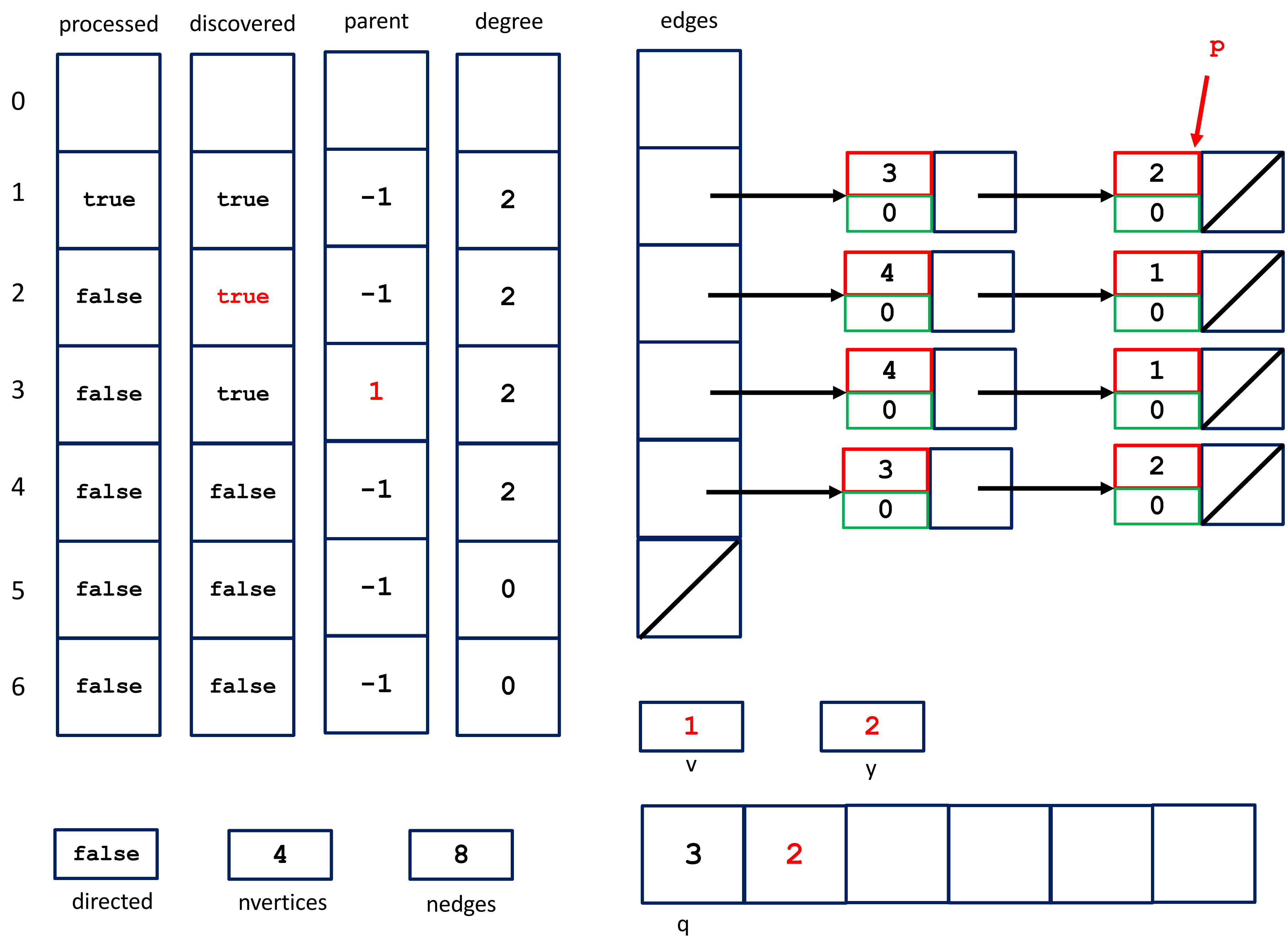
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
true
-1
2
false
false
-1
2
false
true
1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
false
directed
4
nvertices
8
nedges
3
2
q

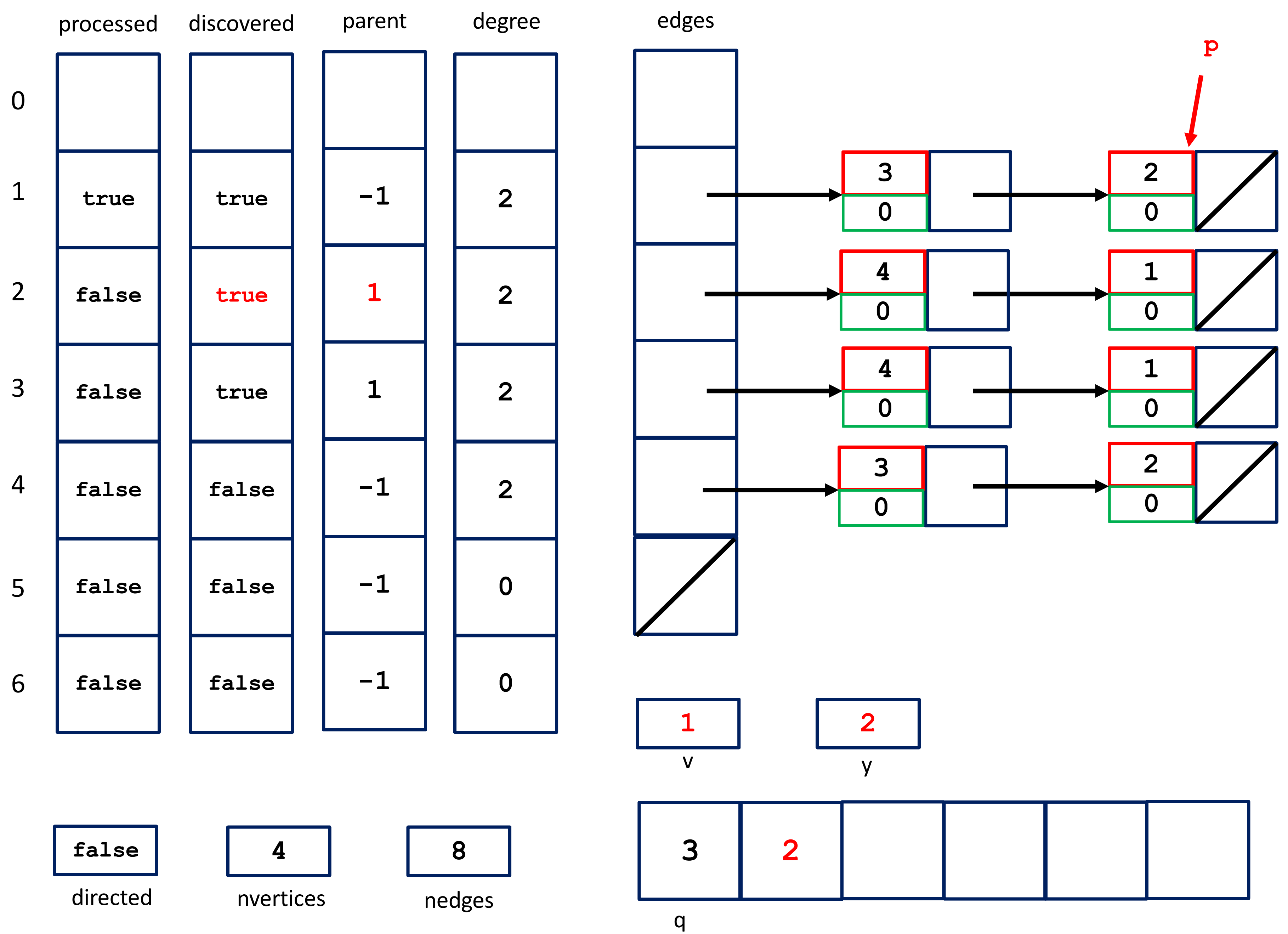
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
true
true
false
false
false
-1
-1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
3
2
q
false
directed
4
nvertices
8
nedges

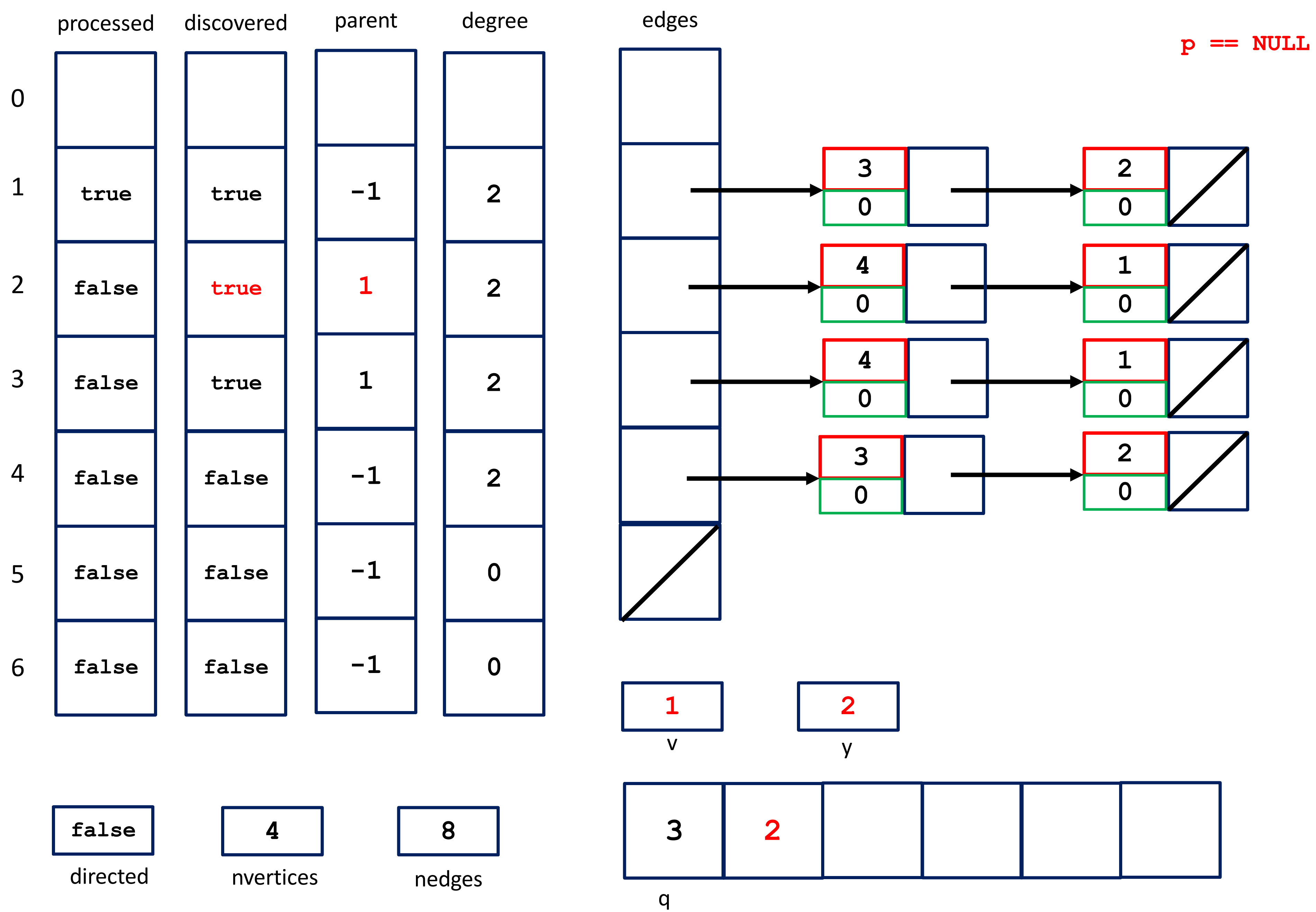
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p
0
1
2
3
4
5
6
true
false
false
false
false
false
true
true
true
false
false
false
-1
1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
3
2
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

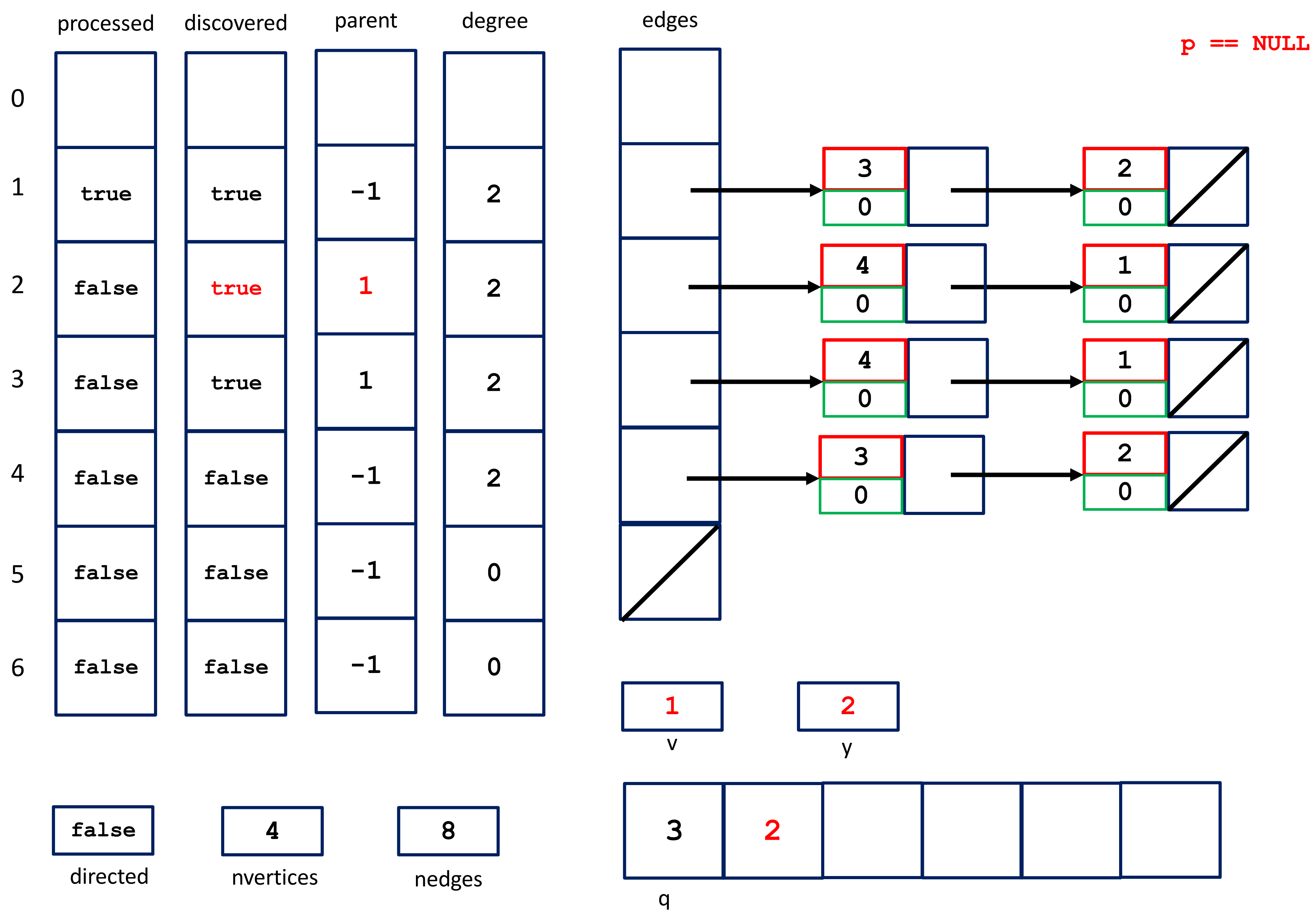

processed
discovered
parent
degree
edges
p == NULL
0
1
2
3
4
5
6
true
false
false
false
false
false
true
true
true
false
false
false
-1
1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
3
2
q
false
directed
4
nvertices
8
nedges

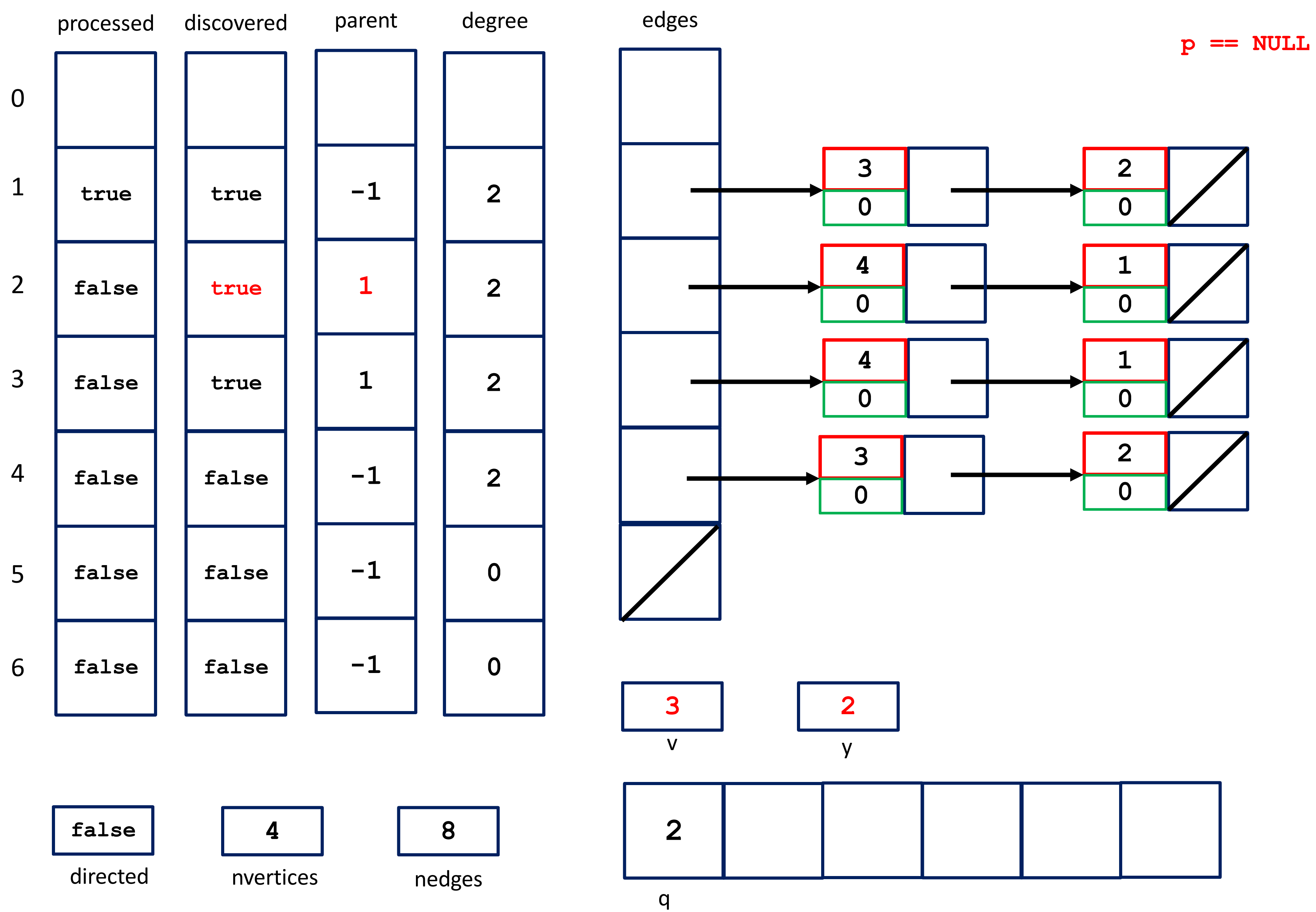
Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

Breadth-First Search

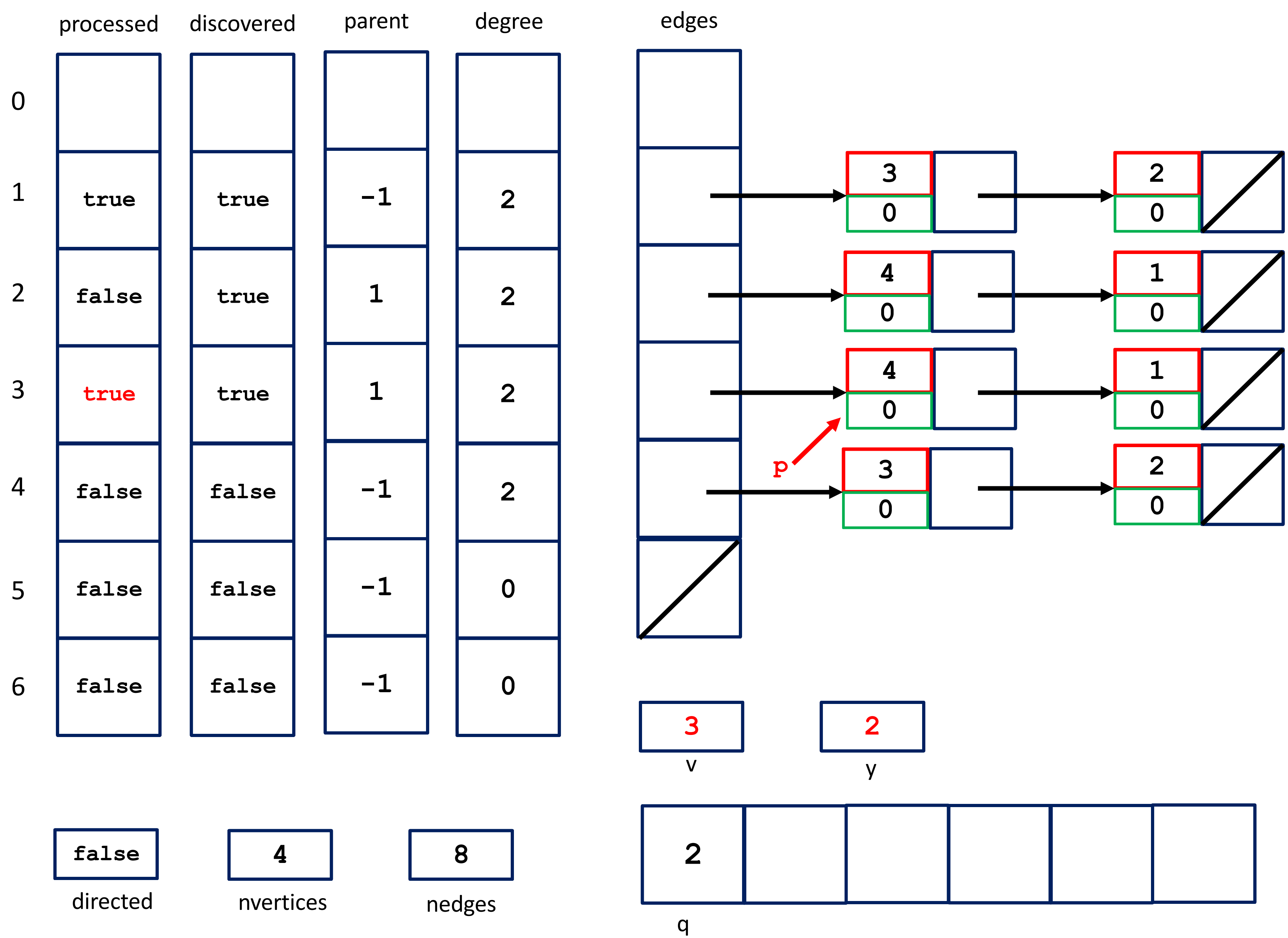
```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
p == NULL
0
1
2
3
4
5
6
true
true
-1
2
false
true
1
2
false
true
1
2
false
false
-1
2
false
false
-1
0
false
false
-1
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
1
v
2
y
3
2
q
false
directed
4
nvertices
8
nedges


processed
discovered
parent
degree
edges
p == NULL
0
1
2
3
4
5
6
true
false
false
false
false
false
true
true
true
false
false
false
-1
1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
3
0
2
0
3
v
2
y
2
q
false
directed
4
nvertices
8
nedges

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = true;
        p = g->edges[v];

        while (p != NULL) {
            y = p->y;
            if ((processed[y] == false) || g->directed)
                process_edge(v,y);
            if (discovered[y] == false) {
                enqueue(&q,y);
                discovered[y] = true;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```


processed
discovered
parent
degree
edges
0
1
2
3
4
5
6
true
false
true
false
false
false
true
true
true
false
false
false
-1
1
1
-1
-1
-1
2
2
2
2
0
0
3
0
2
0
4
0
1
0
4
0
1
0
p
3
0
2
0
3
v
2
y
false
directed
4
nvertices
8
nedges
2
q

Breadth-First Search

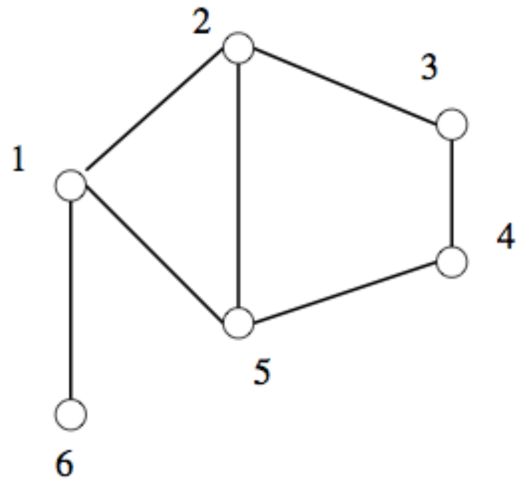
```
/* The exact behaviour of bfs depends on the functions            */
/*    process vertex early()                                       */
/*    process vertex late()                                        */
/*    process edge()                                               */
/* These functions allow us to customize what the traversal does  */
/* as it makes its official visit to each edge and each vertex.    */
/* Here, e.g., we will do all of vertex processing on entry        */
/* (to print each vertex and edge exactly once)                    */
/* so process vertex late() returns without action                 */

process_vertex_late(int v) {
}

process_vertex_early(int v){
   printf("processed vertex %d\n",v);
}

process_edge(int x, int y) {
   printf("processed edge (%d,%d)\n",x,y);
}
```

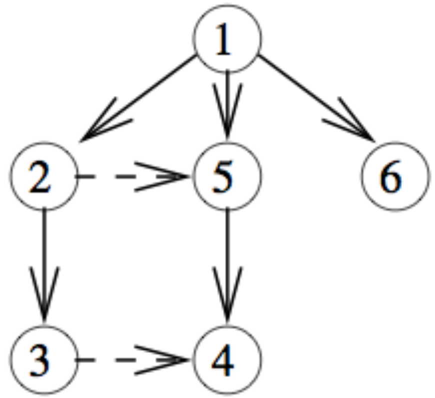
Breadth-First Search

```
/* this version just counts the number of edges                         */

process_edge(int x, int y) {
   nedges = nedges + 1;
}
```

Breadth-First Search

- Finding Paths
  - The `parent` array in `bfs()` is very useful for finding interesting paths through a graph
  - The vertex that discovered vertex `i` is defined as `parent[i]`





vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Breadth-First Search

- Finding Paths
  - Every vertex is discovered during the course of a traversal so every node has a parent (except the root)
  - The parent relation defines a tree of discovery with the initial search node as the root of the tree
  - Because vertices are discovered in order of increasing distance from the root, this tree has a very important property
    - The unique tree path from the root to each node uses the smallest number of edges (and intermediate nodes) possible on any path from the root to that vertex
    - Thus BFS can be used to find shortest paths in an **unweighted** graph

Breadth-First Search

- Finding Paths
  - To reconstruct a path we follow the chain of ancestors from the destination node x to the root
  - Note we have to work backwards (we only know the parents)
  - We find the path from to the root and
    - Either store it and explicitly reverse it using a stack
    - Or construct the path recursively (in which case the stack is implicit)

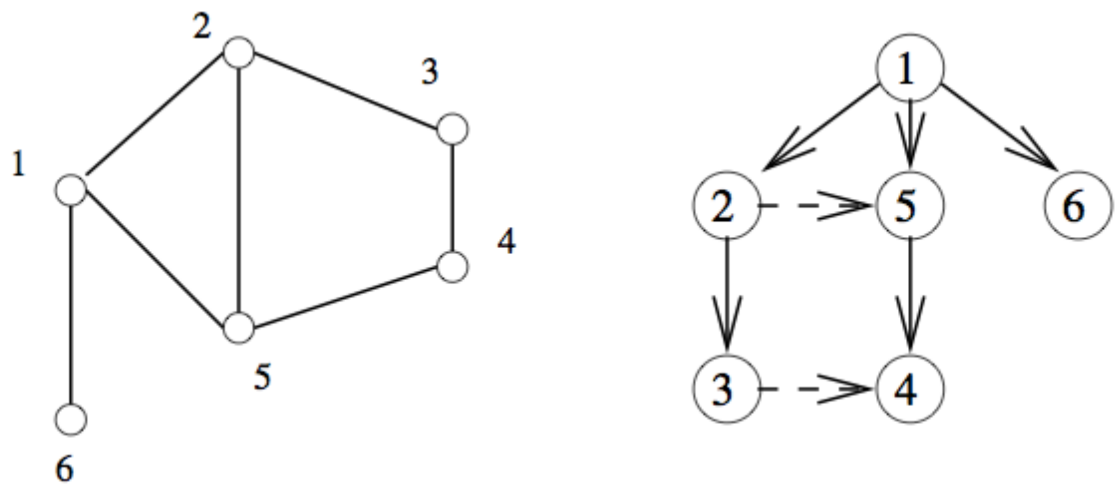
Breadth-First Search

```
bool find_path(int start, int end, int parents[]) {

   bool is_path;

   if (end == -1) {
      is_path = false; // some vertex on the path back from the end
                       // has no parent (not counting start)
   }
   else if ((start == end)) {
       printf("\n%d",start);
       is_path = true; // we have reached the start vertex
   }
   else {
      is_path = find_path(start,parents[end],parents);
      printf(" %d",end);
   }
   return(is_path);
}
```

vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Breadth-First Search

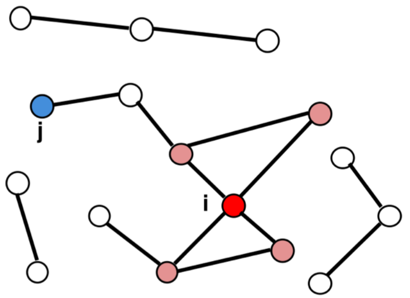
```
find_path(1,4,parent)

-> find_path(1,5,parent) -> find_path(1,1,parent) -> printf(1)
   printf(4)                  printf(5)
```



vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Breadth-First Search

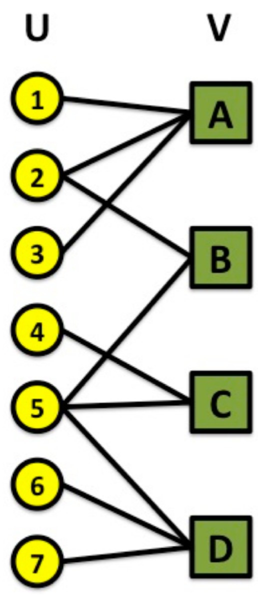
- Applications of Breadth-First Search



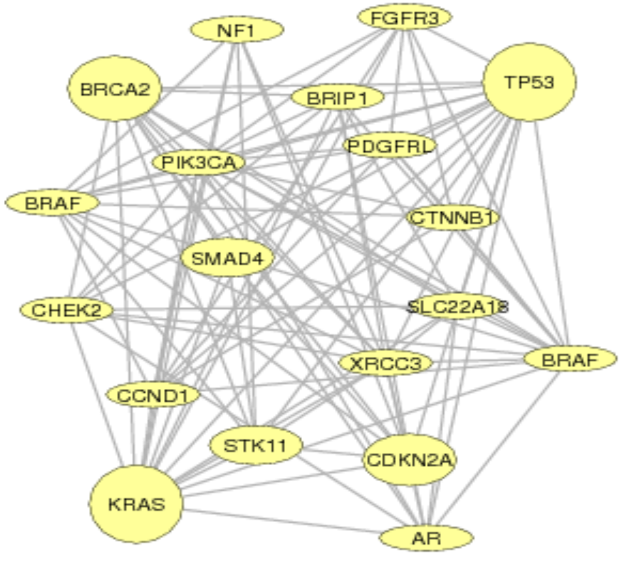
  - Identifying connected components
    - A graph is connected if there is a path between any two vertices
    - A connected component of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices
    - The components are separate "pieces" of the graph such that there is no connection between the pieces
    - Many complicated problems reduce to finding or counting connected components
    - How would you find and label all the components in a graph?

Breadth-First Search

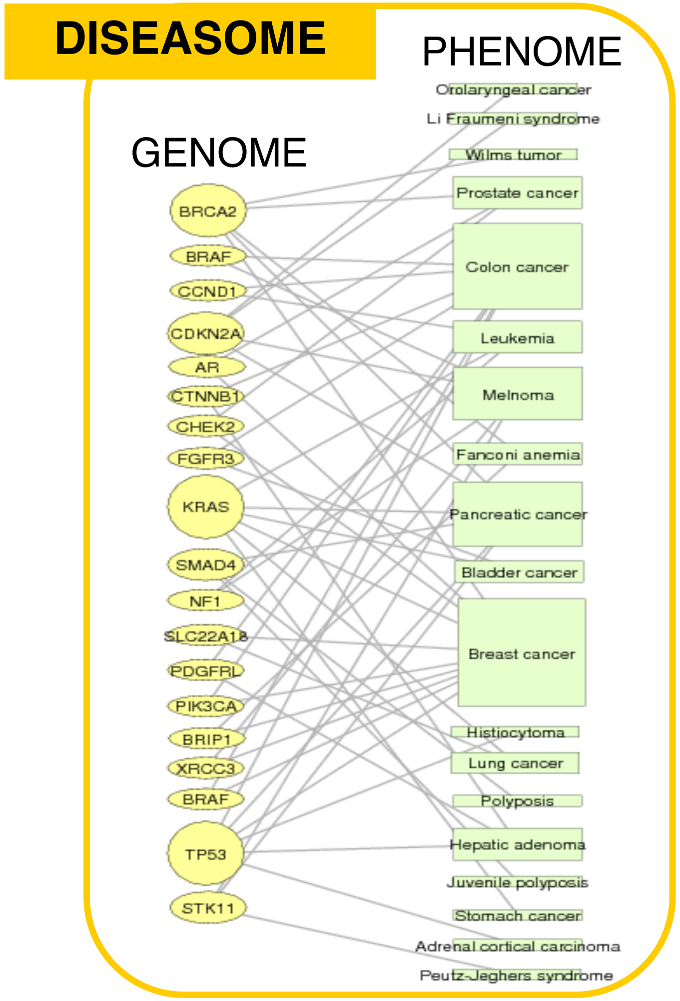
- Applications of Breadth-First Search
  - Two-Colouring Graphs
    - The *vertex-colouring* problem seeks to assign a label (or colour) to each vertex of a graph such that no edge links any two vertices of the same colour
    - The goal is use as few colours as possible
    - A graph is bipartite if it can be coloured without conflicts using only two colours

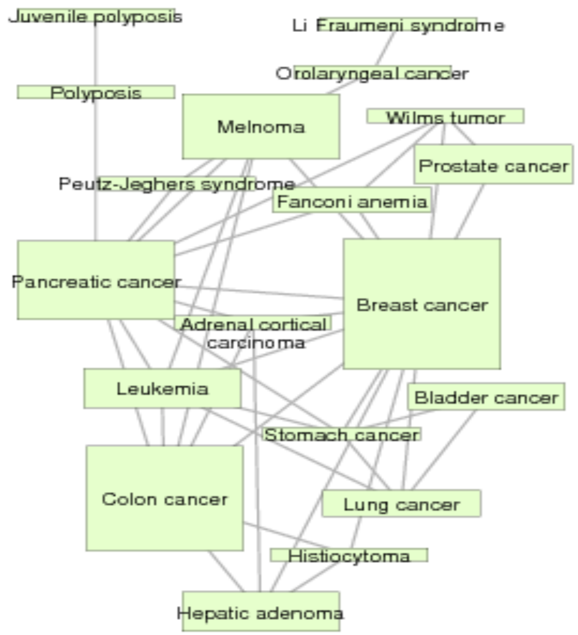


Breadth-First Search



Gene network





Disease network

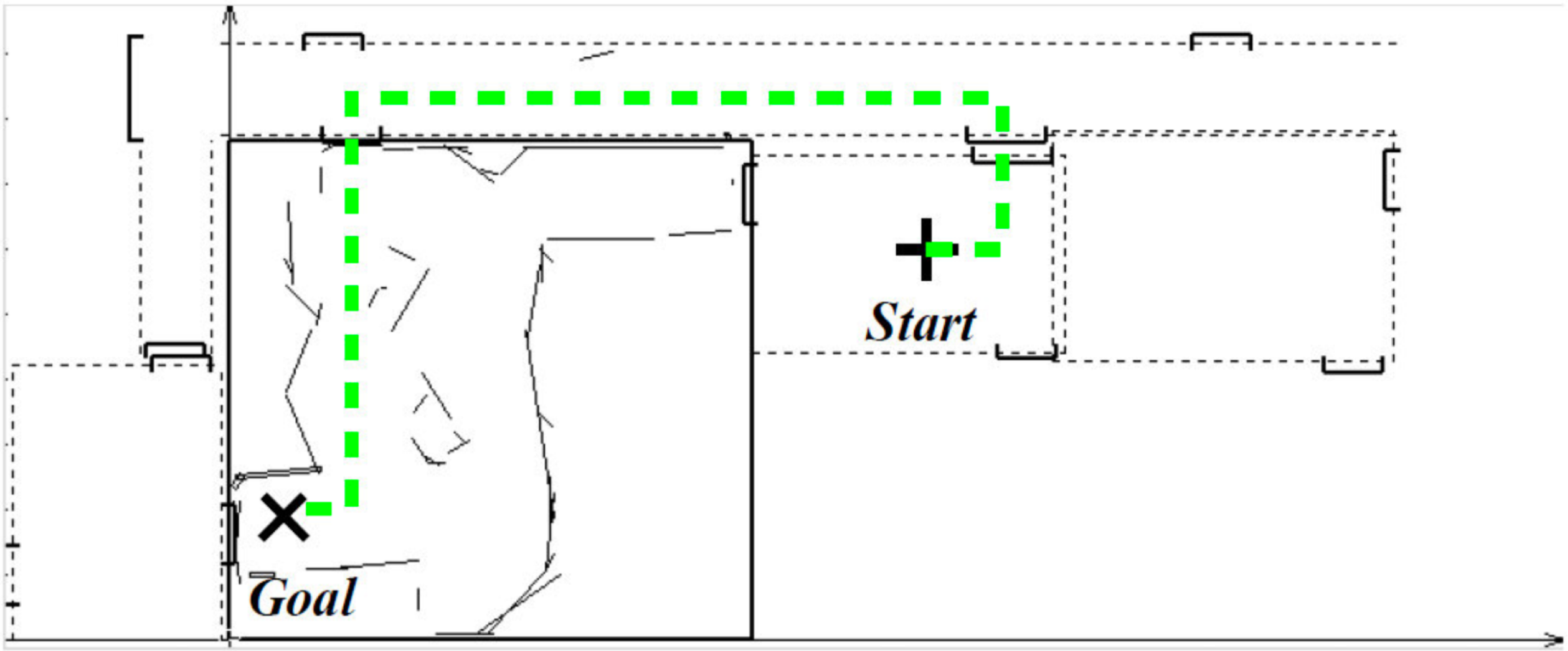
Goh, Cusick, Valle, Childs, Vidal & Barabási, PNAS (2007)

Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

Breadth-First Search

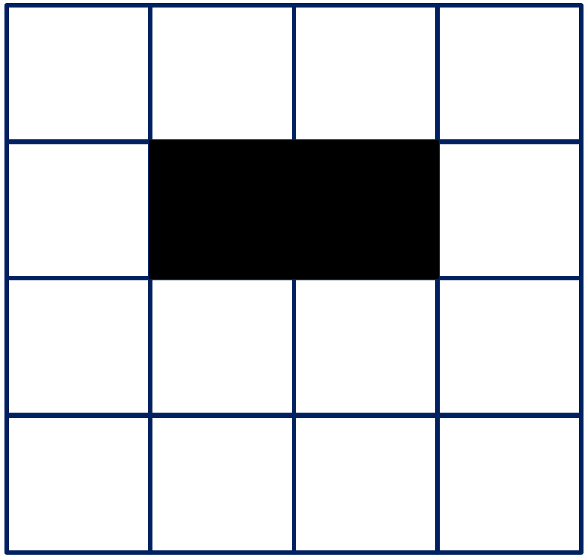
- Applications of Breadth-First Search
  - Robot path-planning



Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Represent the map of the environment as an occupancy grid



0	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Represent the map of the environment as an occupancy grid

0	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

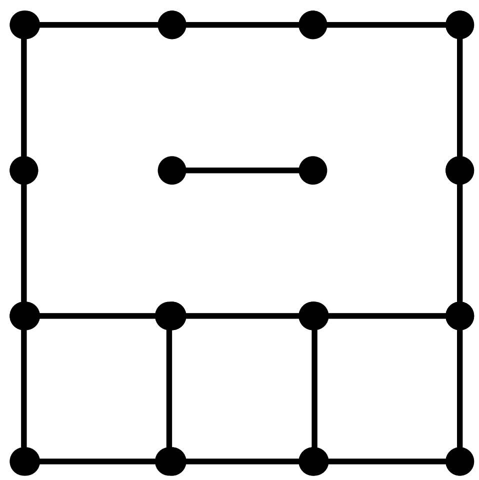
0	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Convert this to a graph

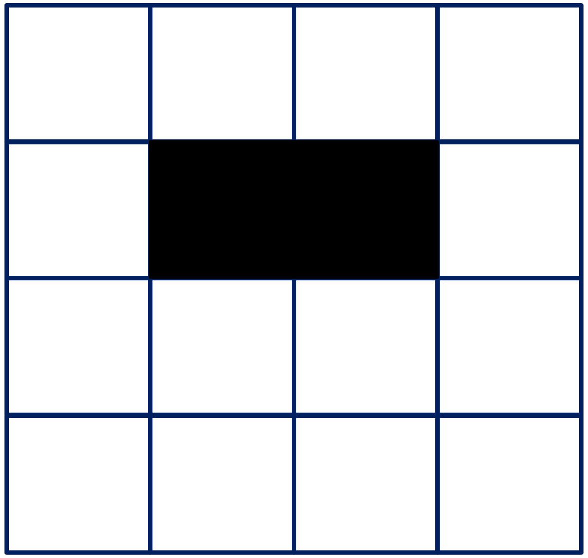
0	0	0	0
0	1	1	0
0	0	0	0
0	0	0	0

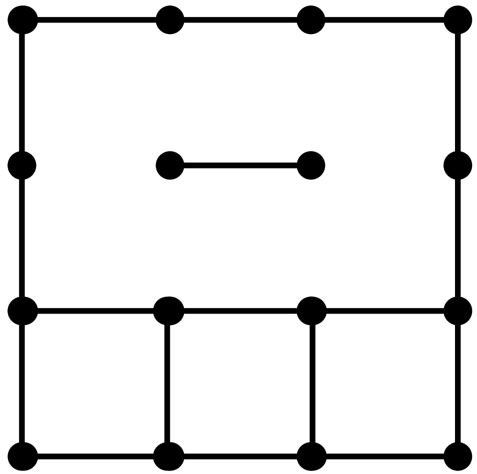


Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Convert this to a graph



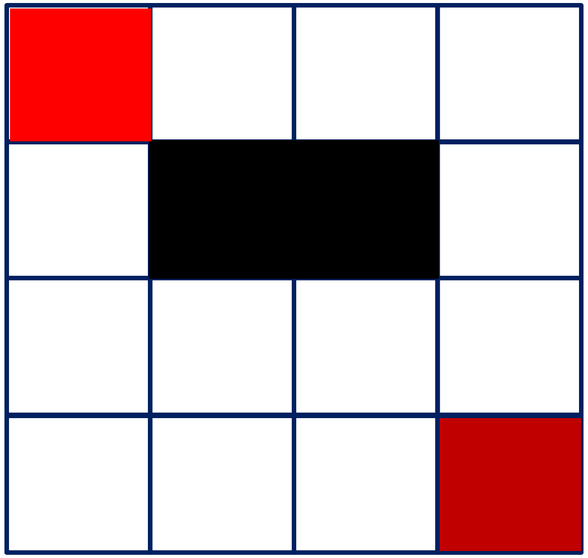


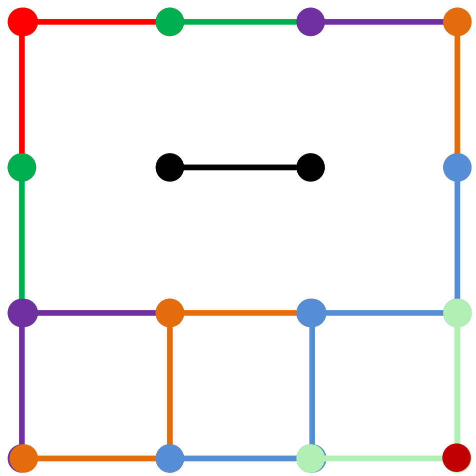
Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Do a BFS from the robot start position …

    To find the shortest path to all other vertices



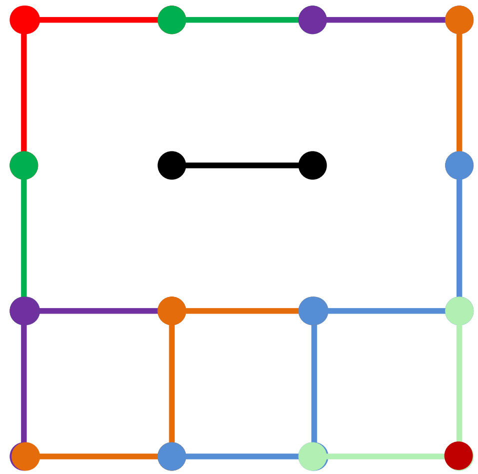


Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

    Mark the path from the robot start position to the goal position on the occupancy grid

2	0	0	0
2	1	1	0
2	0	0	0
2	2	2	2



Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning

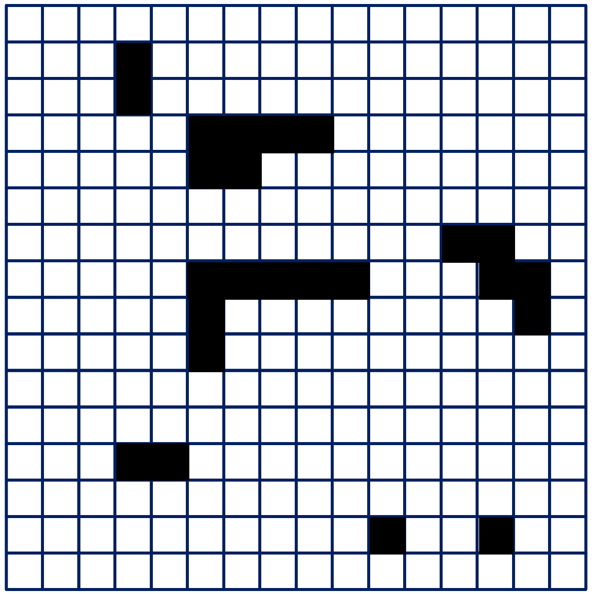
    Mark the path from the robot start position to the goal position on the occupancy grid

2	0	0	0
2	1	1	0
2	0	0	0
2	2	2	2

2	0	0	0
2	1	1	0
2	0	0	0
2	2	2	2

Breadth-First Search

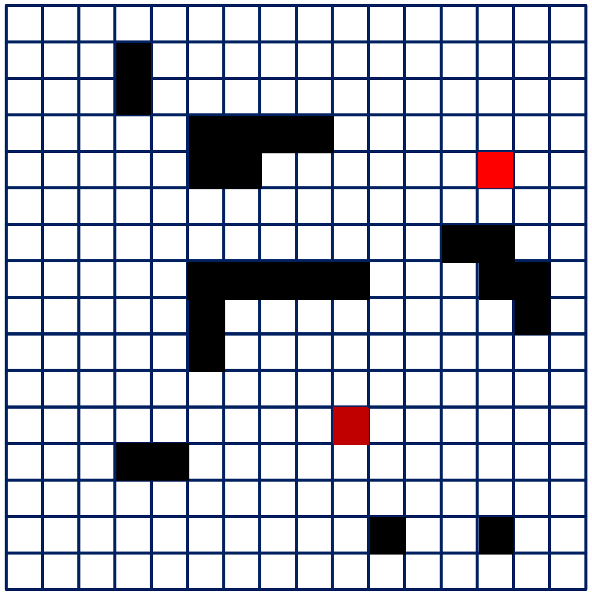
- Applications of Breadth-First Search
  - Robot path-planning



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Breadth-First Search

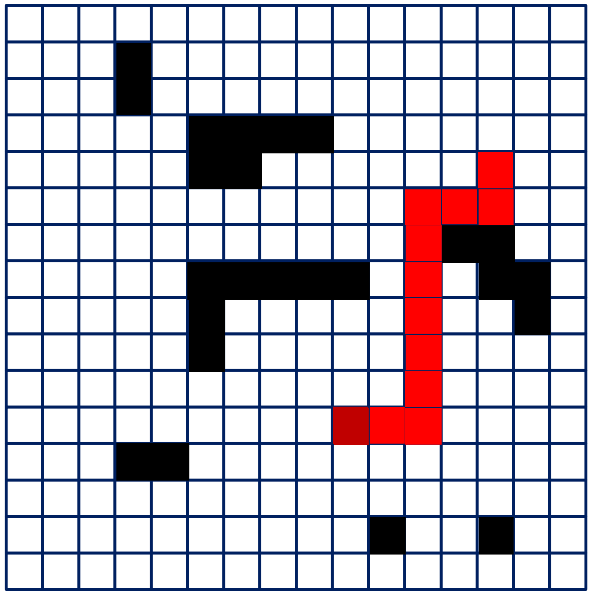
- Applications of Breadth-First Search
  - Robot path-planning



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Breadth-First Search

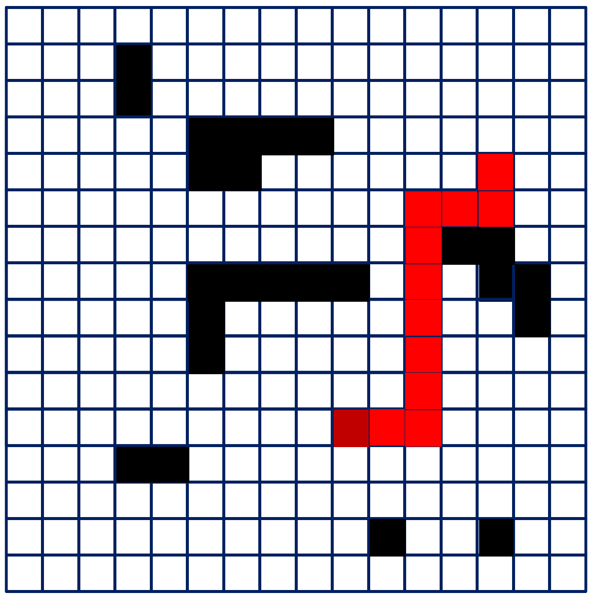
- Applications of Breadth-First Search
  - Robot path-planning



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	0	0	0	2	2	2	0	0
0	0	0	0	0	0	0	0	0	0	0	2	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	2	0	1	1	0
0	0	0	0	0	1	0	0	0	0	1	2	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	2	2	2	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Breadth-First Search

- Applications of Breadth-First Search
  - Robot path-planning



```
#
#
    # # # #
    # #             @
                * * *
                * # #
    # # # # #   *   # #
    #         # *     #
    #           *
                *
            $ * *
# #

              #     #
```