Data Structures

Graphs

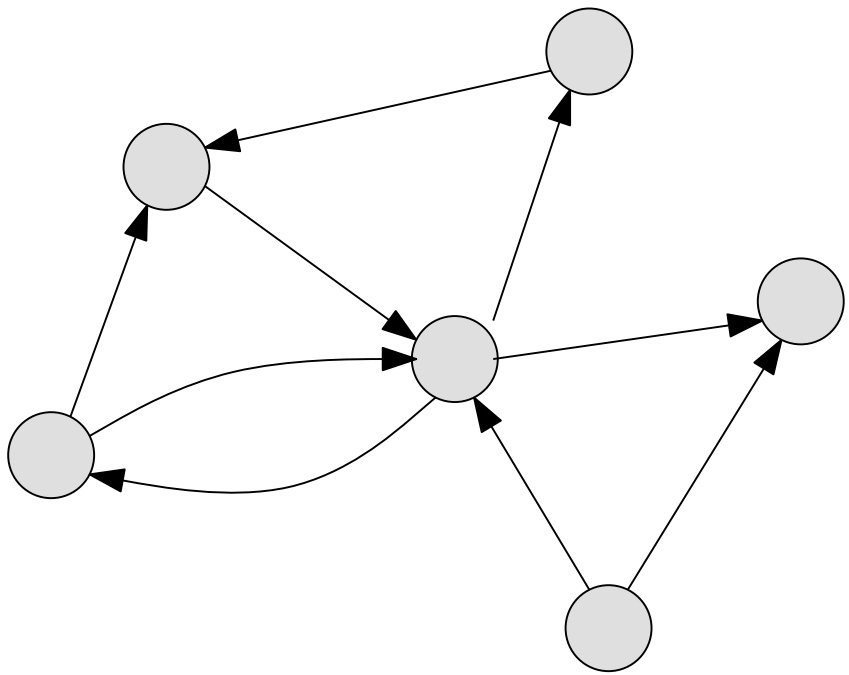
Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Graphs

Directed Graph

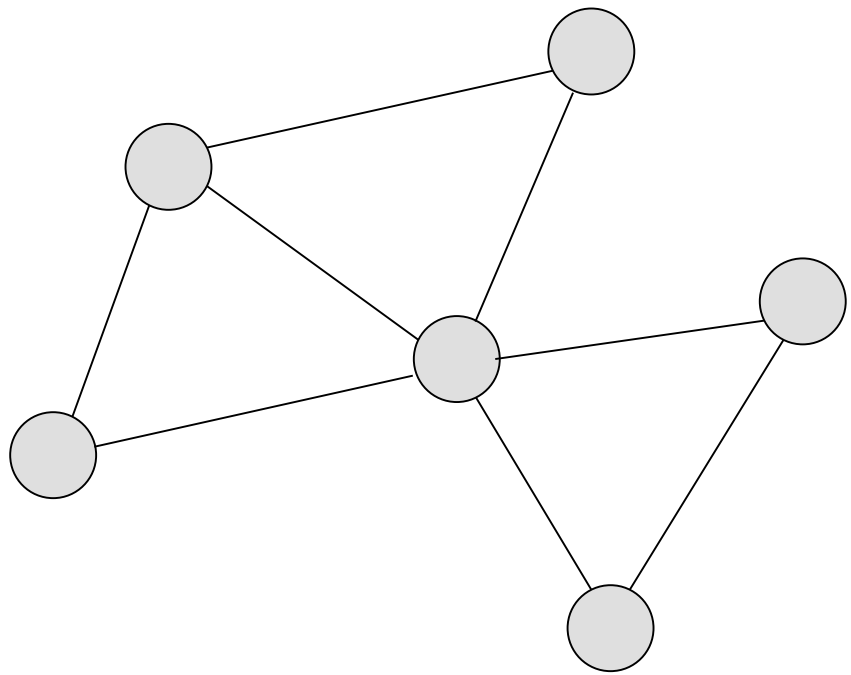
A directed graph G is a pair of sets (V, E) where V is a set of nodes and E is a set of ordered node pairs.



Graphs

Undirected Graph

An undirected graph G is a pair of sets (V, E) where V is a set of nodes and E is a set of node pairs.



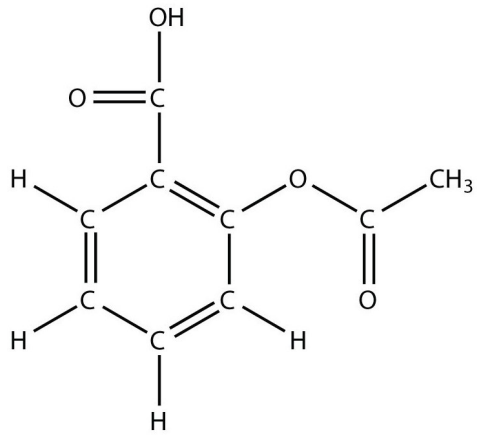
Modelling Problems

A lot of different problems can be modelled with graphs.

Road Networks

Social Networks

Molecules



Basic Operations

- **Accessing information.**
  Given a node or edge, access its information, e.g. edge weight, distance from other node, etc.

Basic Operations

- **Accessing information.**
  Given a node or edge, access its information, e.g. edge weight, distance from other node, etc.
- **Navigation.**
  - Given a node, access its outgoing edges.
  - This operation lies in the heart of most graph algorithms.
  - Sometimes we would also like easy access to the incoming edges of a node.

Basic Operations

- **Accessing information.**
  Given a node or edge, access its information, e.g. edge weight, distance from other node, etc.
- **Navigation.**
  - Given a node, access its outgoing edges.
  - This operation lies in the heart of most graph algorithms.
  - Sometimes we would also like easy access to the incoming edges of a node.
- **Edge Queries.**
  - Given 2 nodes (u, v) we would like to know if such an edge exists in the graph.
  - Sometimes we would like to easily find the opposite edge (v, u) of a directed edge (u, v).

Basic Operations

- **Accessing information.**
  Given a node or edge, access its information, e.g. edge weight, distance from other node, etc.
- **Navigation.**
  - Given a node, access its outgoing edges.
  - This operation lies in the heart of most graph algorithms.
  - Sometimes we would also like easy access to the incoming edges of a node.
- **Edge Queries.**
  - Given 2 nodes (u, v) we would like to know if such an edge exists in the graph.
  - Sometimes we would like to easily find the opposite edge (v, u) of a directed edge (u, v).
- **Construct, convert and output.** Convert the representation to the most natural form for the problem that we are trying to solve.

Basic Operations

- **Accessing information.**
  Given a node or edge, access its information, e.g. edge weight, distance from other node, etc.
- **Navigation.**
  - Given a node, access its outgoing edges.
  - This operation lies in the heart of most graph algorithms.
  - Sometimes we would also like easy access to the incoming edges of a node.
- **Edge Queries.**
  - Given 2 nodes (u, v) we would like to know if such an edge exists in the graph.
  - Sometimes we would like to easily find the opposite edge (v, u) of a directed edge (u, v).
- **Construct, convert and output.** Convert the representation to the most natural form for the problem that we are trying to solve.
- **Update.** Add and remove nodes or edges, change the information of node or edge, etc.

Graph Representation

There are different methods, depending on our needs on space, time and whether we would also like to support dynamic graphs or just static ones.

- Unordered edge list
- Adjacency arrays
- Adjacency matrix
- Adjacency lists

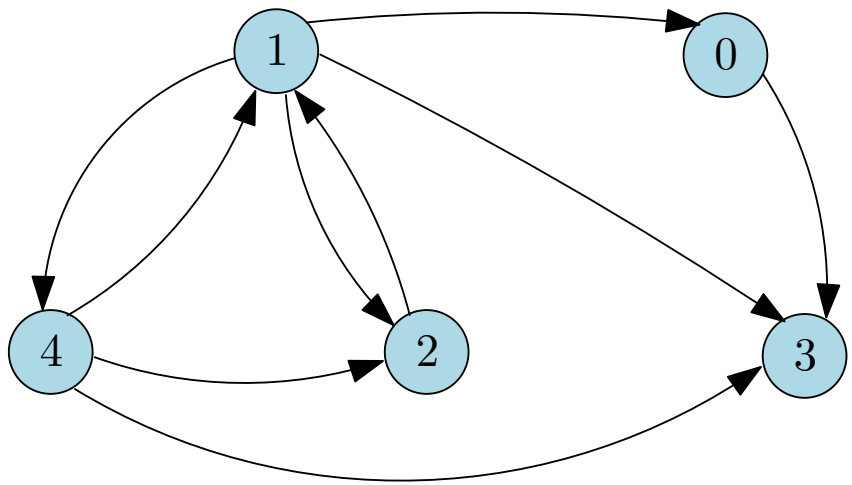
and others which maybe relevant to the application at hand.

Unordered edgelist

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

The representation contains:

- an unordered list of edges



$$((1,0),(0,3),(1,3),(1,2),(4,3),(2,1),(4,2),(1,4),(4,1))$$

Unordered edgelist

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

$$((1,0),(0,3),(1,3),(1,2),(4,3),(2,1),(4,2),(1,4),(4,1))$$

Characteristics

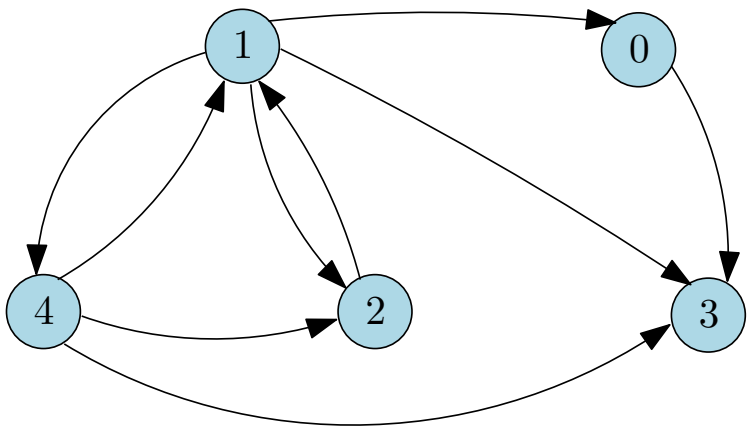
- common for input/output
- each addition of edges or nodes in $\mathcal{O}(1)$
- all the rest (e.g. navigation) $\mathcal{O}(m)$
  - too much!

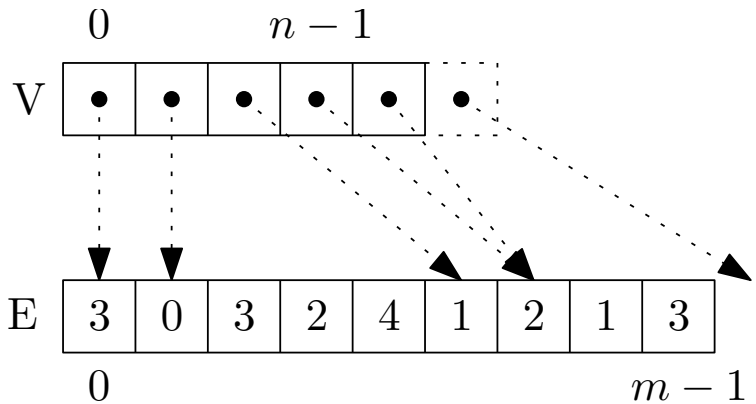
Adjacency Array

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

The representation contains:

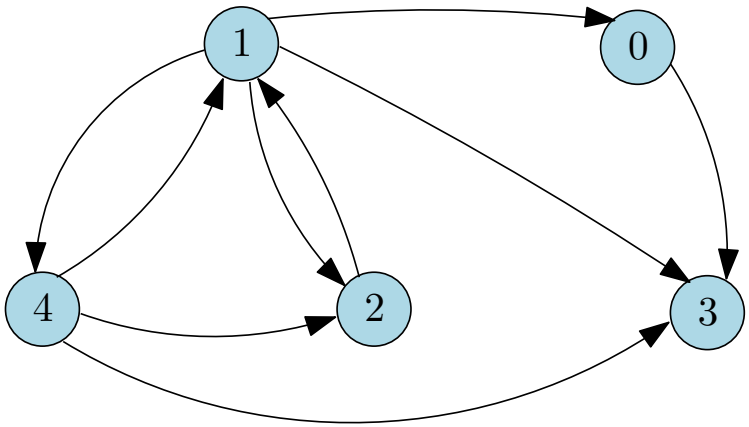
- array $E[0 \ldots m-1]$ with outgoing edges grouped by node
- array $V[0 \ldots n-1]$ with starting positions of subarrays in E (per node)
  - For node v, position $V[v]$ contains the position in E where the first outgoing edge of v is located.
  - Helpful to add virtual entry $V[n] = m$, then outgoing edges of v are at $E[V[v]], \ldots, E[V[v+1]-1]$.

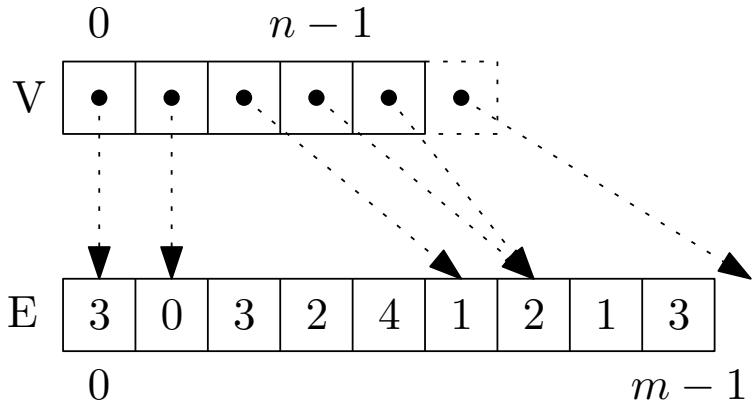




Adjacency Array

graph $G(V, E)$ where $|V| = n$ and $|E| = m$





Characteristics

- space consumption $n + m + \Theta(1)$ words
- addition information for nodes can be stored in the node array
- addition information for edges can be stored in the edge array
- static representation
- navigation (e.e. print all outgoing edges) in time equal to the degree of a node $\mathcal{O}(d)$
- edge query $\mathcal{O}(d)$ time

Adjacency Array

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

Exercise

Design a linear time $\mathcal{O}(n + m)$ algorithm which converts a directed graph from an unordered edge list representation into a representation as adjacency arrays. You must use only $\mathcal{O}(n)$ additional space.

Hint: Consider the problem of sorting edges based on their source vertex. Use, by adjusting appropriately, an algorithm which sorts integers in linear time.

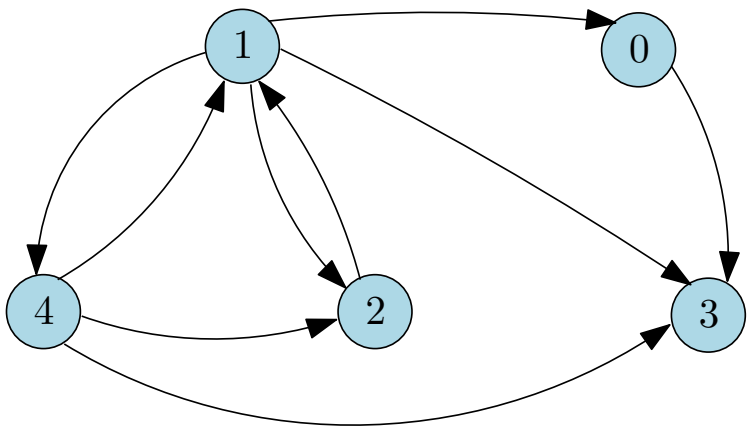
You will need to read the chapter on sorting.

Adjacency Matrix

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

The representation contains:

- a 2 dimensional array A with dimensions $n \times n$
- the array location i, j denotes whether directed edge (i, j) exists or not
- if $e = (i, j) \in E$ then $A[i][j] = 1$ otherwise 0



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency Matrix

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

Characteristics

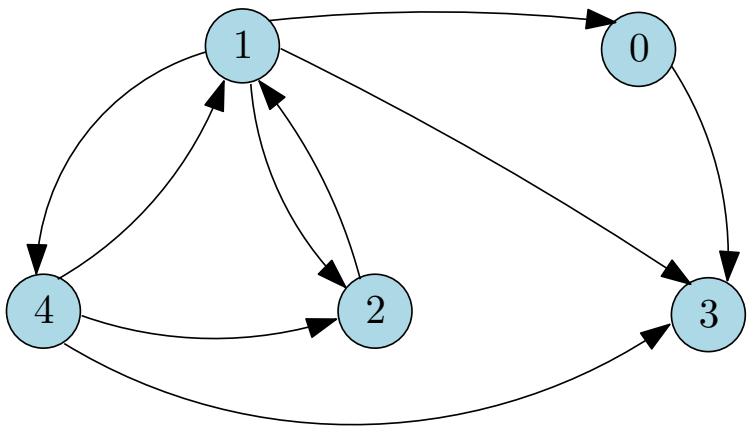
- Insert/delete edges in $\mathcal{O}(1)$ time
- Edge query in $\mathcal{O}(1)$ time
- Navigation in $\mathcal{O}(n)$ time - good only in very dense graphs
- Space n^2 bits
- Connection between graphs and linear algebra, e.g. if $C = A^k$ then C_{ij} counts the total number of paths from node i to node j which have k edges.

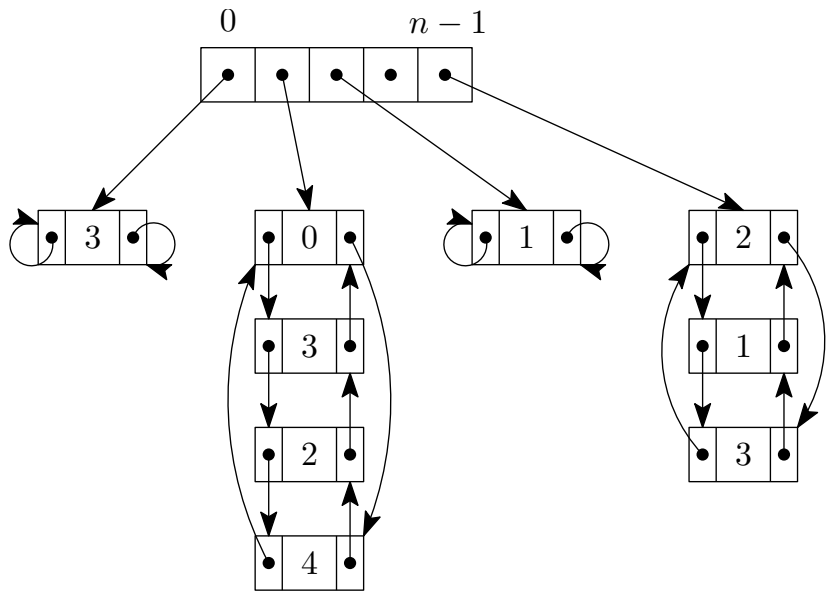
Adjacency Lists

graph $G(V, E)$ where $|V| = n$ and $|E| = m$

The representation contains:

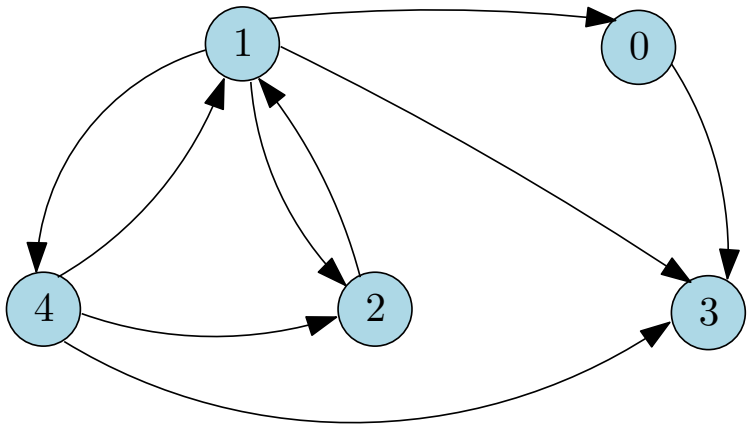
- a one-dimensional array A with one position for each node
- every element of array A is a list of edges which have the corresponding node as its source

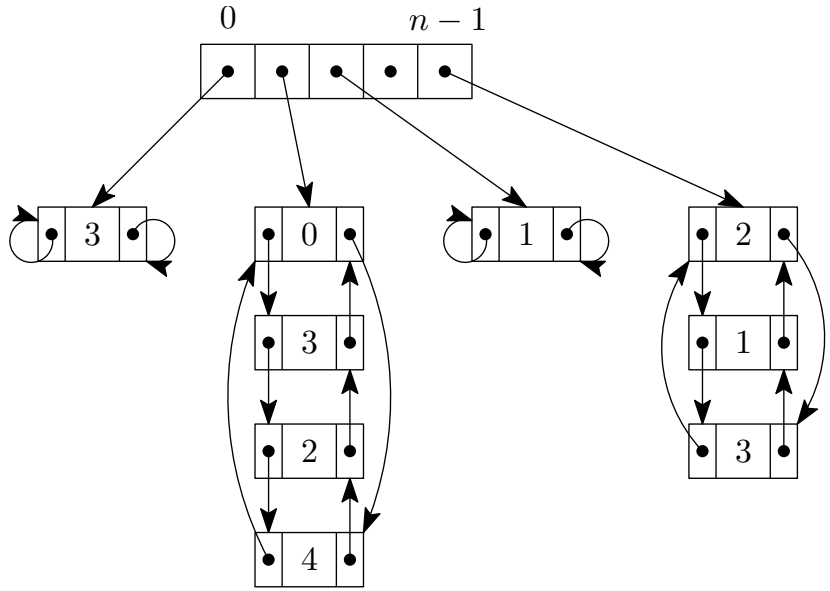




Adjacency List

$G(V, E)$ $|V| = n$ $|E| = m$





Characteristics

- space consumption $n + 3m$ words
- additional information can be stored either at the lists (for the edges) or at the array (for the nodes)
- dynamic representation - each addition/removal of edges
- navigation (e.g. all outgoing edges) in time equal with the vertex degree $\mathcal{O}(d)$
- edge query in $\mathcal{O}(d)$ time

Graph with Weights

Using graphs in order to answer path queries in a GPS device, we would like to find the shortest path between two points:

Graph with Weights

Using graphs in order to answer path queries in a GPS device, we would like to find the shortest path between two points:

- we have pre-calculated an approximation on the average time which we need in order to travel road segments
- we associate each road segment with an edge of the graph, and set its weight equal to the travel time of the segment

In general we can associate information with both vertices and edges. Another example is the association of a name with each edge.

Graph Representation

Exercise

You would like to store a graph which represents the road map of the city that you leave in. Answer the following questions:

1. *What does each graph vertex represent?*
2. *What does each graph edge represent?*
3. *What additional information is required on the vertices and edges, in order to be able to answer shortest path queries and be able to draw the map to the user.*
4. *White representation from the ones we explained is the more relevant? Why? Explain.*

Breadth-first search

BFS is one of the simplest search algorithms in graphs and used as a basis for several other important graph algorithms.

Input

- graph $G(V, E)$, and
- a start node s

Operation

The BFS algorithm explores edges of G in order to discover all vertices which are reachable from s.

Breadth-first search

Output

Distances The distance (least number of edges) from s to all reachable nodes.

BFS Tree A "breadth-first tree" with node s as root which contains all reachable nodes.

For every reachable node v from s, the path on the BFS tree corresponds to a shortest path in the graph.

Breadth-first search

Directed and Undirected

The algorithms works fine with directed and undirected graphs.

Breadth-first search

Directed and Undirected

The algorithms works fine with directed and undirected graphs.

Search Frontier

The search is called breadth-first since it grows the search frontier in all its width.

In order words, the algorithm discovers all nodes at distance k from s before it discovers any node at distance $k + 1$.

Breadth-first search

In order to track down progress, the BFS algorithm colors graph nodes using 3 colors a) white, b) gray, and c) black.

- All nodes start as white, become gray then black.
- A node gets *discovered* the first time that it appears in the search, and becomes non-white.
- All gray and black nodes have been discovered.
- Gray nodes represent the *frontier* of the search and may have white neighbors.

Breadth-first search

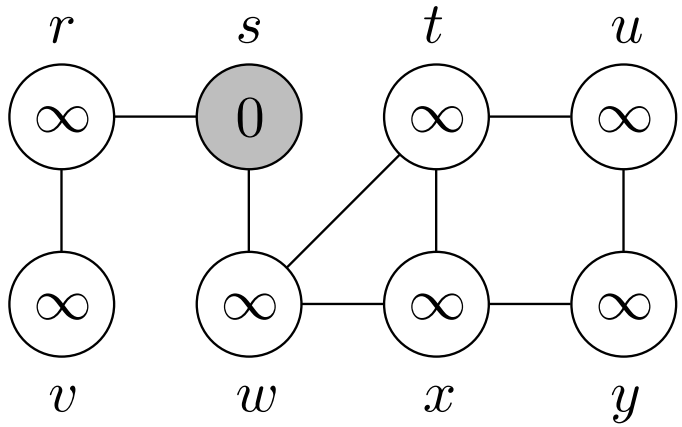
Pseudo Code

$BFS(G, s)$

```
for each vertex u ∈ V \ {s} do
    color[u] = WHITE
    d[u] = ∞
    π[u] = nil
end
color[s] = GRAY
d[s] = 0
π[s] = nil
Q = {s}
while Q ≠ ∅ do
    u = Q.head()
    for each neighbor v of u do
        if color[v] = WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            π[v] = u
            Q.enqueue(v)
        end
    end
    Q.dequeue()
    color[u] = BLACK
end
```

Breadth-first search

Example

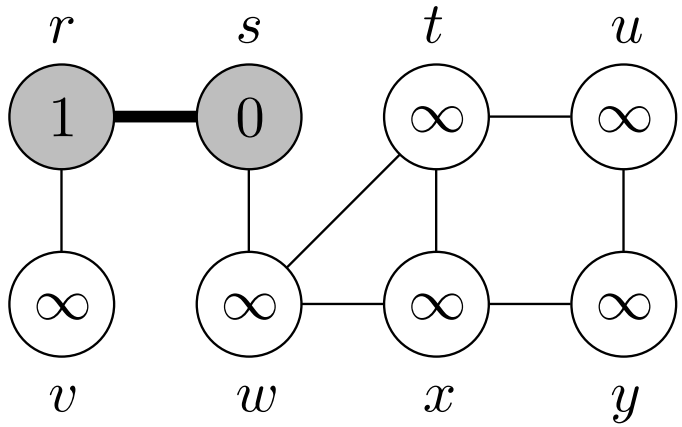


Q $\boxed{s}$

0

Breadth-first search

Example

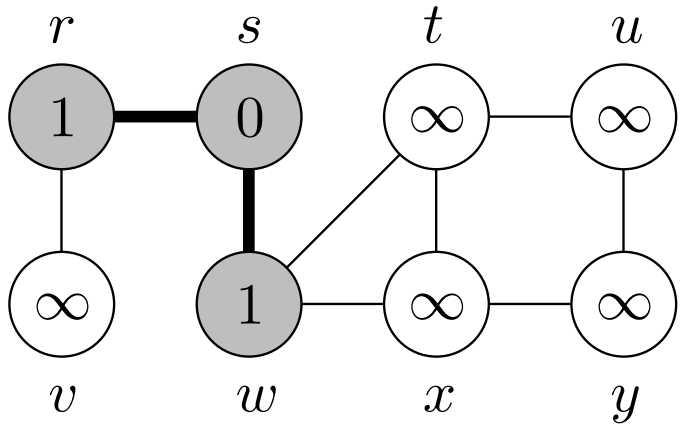


Q $\boxed{r}$

1

Breadth-first search

Example

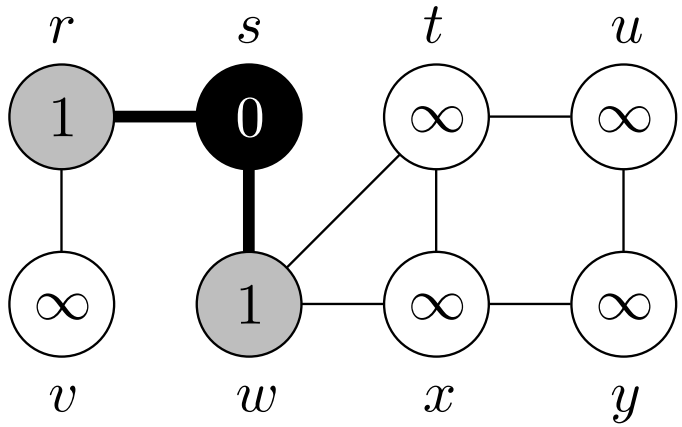


Q $\boxed{r}$ $\boxed{w}$

1 1

Breadth-first search

Example

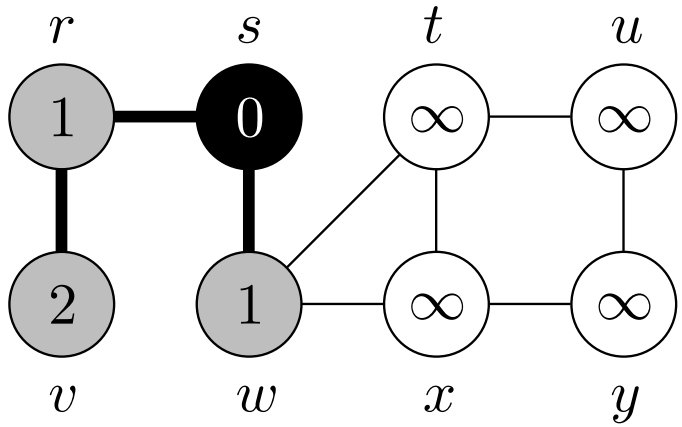


Q | r | w

1 1

Breadth-first search

Example

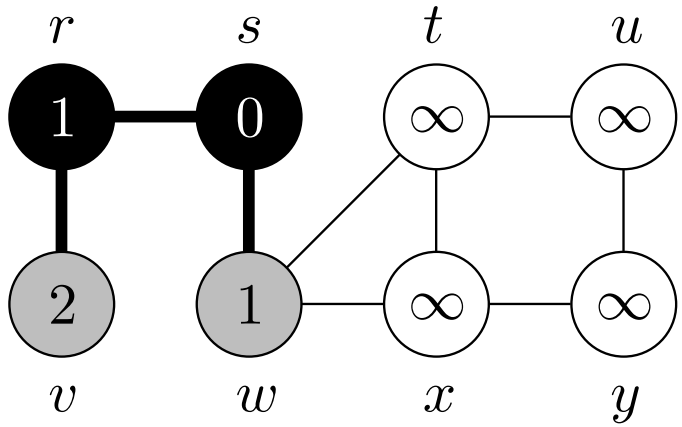


Q $\boxed{w}$ $\boxed{v}$

1 2

Breadth-first search

Example

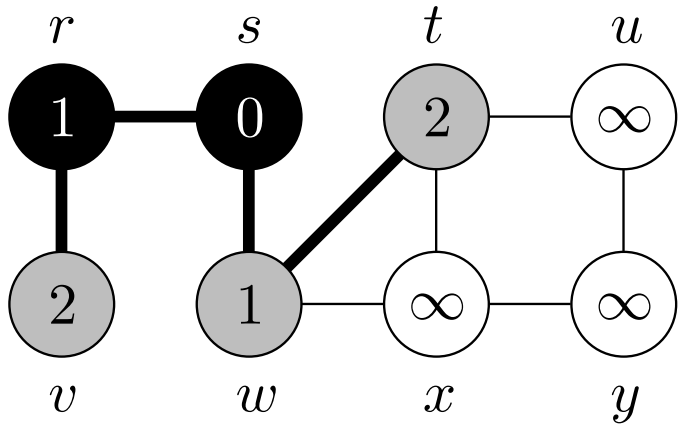


Q $\boxed{w}$ $\boxed{v}$

1 2

Breadth-first search

Example

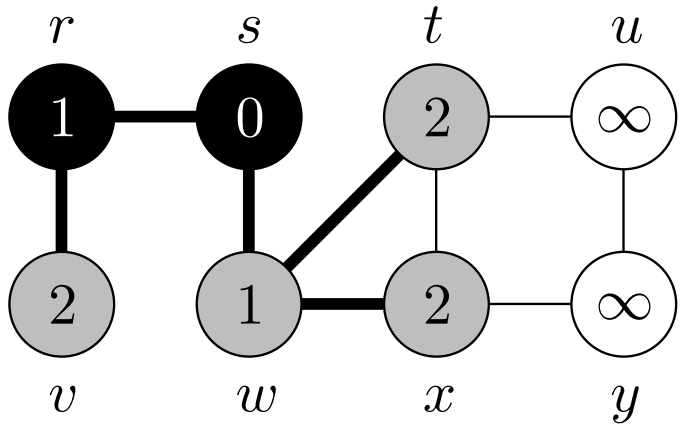


Q | v | t

2 2

Breadth-first search

Example

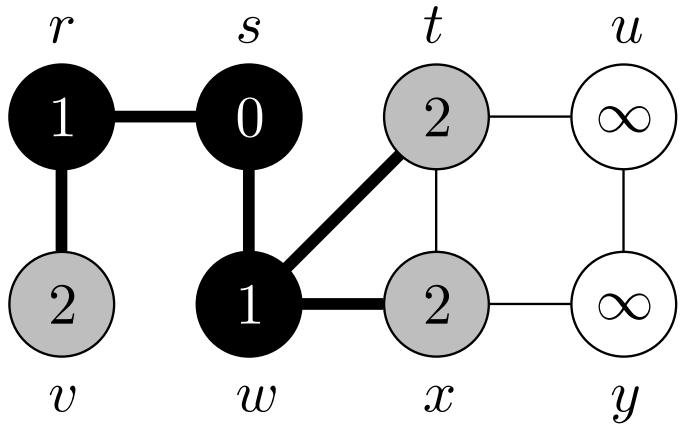


Q $\boxed{v}$ $\boxed{t}$ $\boxed{x}$

2 2 2

Breadth-first search

Example

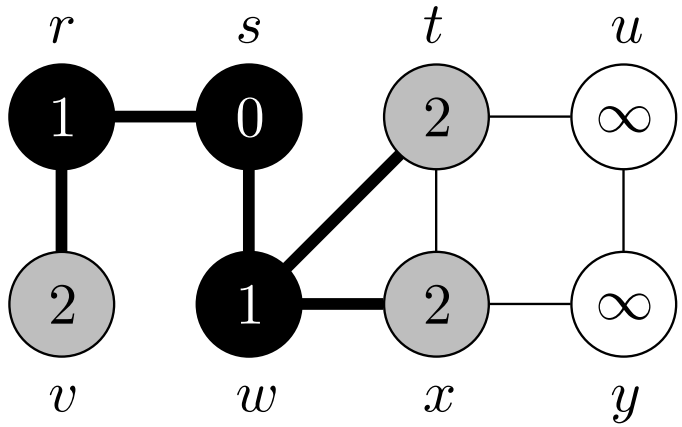


Q | v | t | x

2 2 2

Breadth-first search

Example

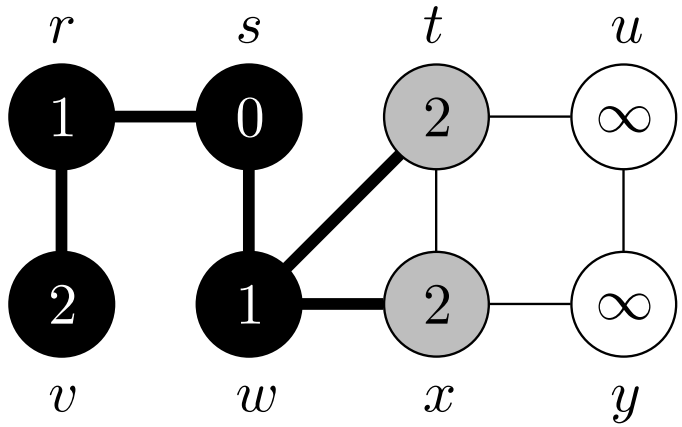


Q $\boxed{t}$ $\boxed{x}$

2 2

Breadth-first search

Example

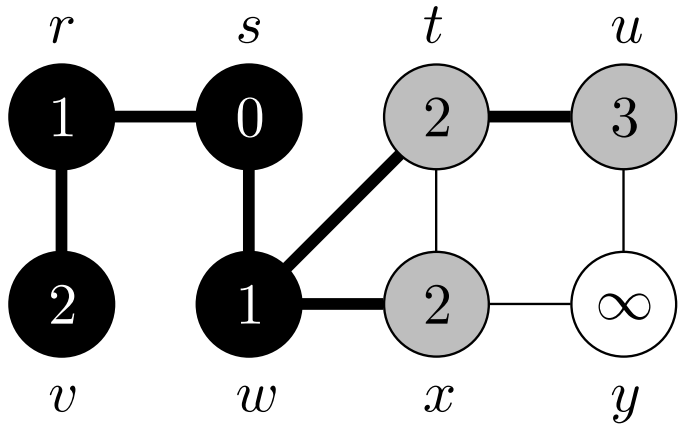


Q $\boxed{t}$ $\boxed{x}$

2 2

Breadth-first search

Example

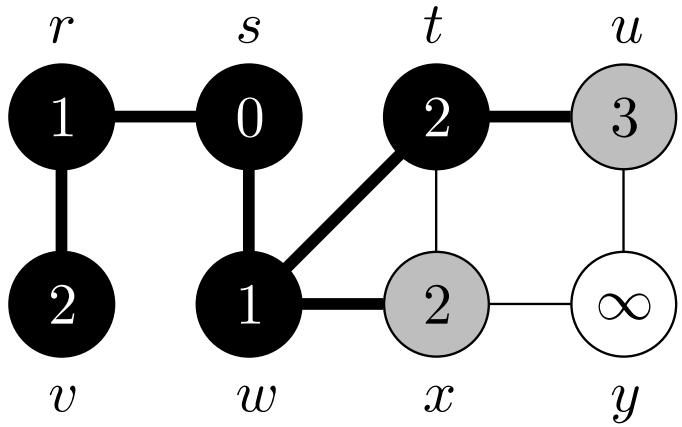


Q $\boxed{x}$ $\boxed{u}$

2 3

Breadth-first search

Example

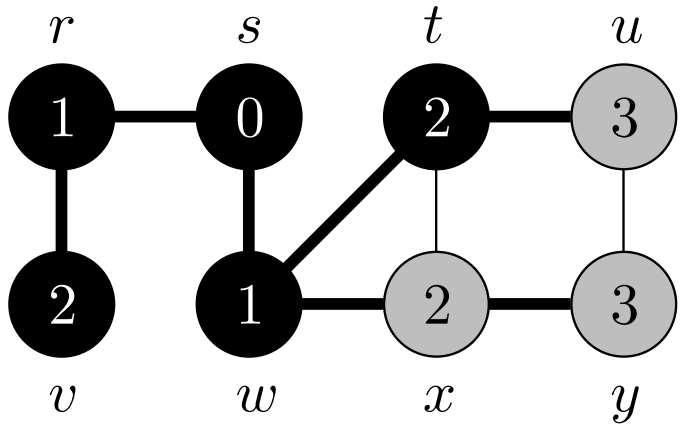


Q $\boxed{x}$ $\boxed{u}$

2 3

Breadth-first search

Example

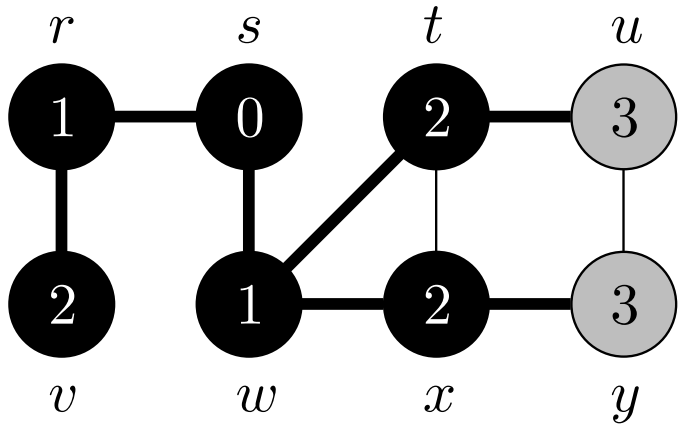


Q $\boxed{u}$ $\boxed{y}$

3 3

Breadth-first search

Example

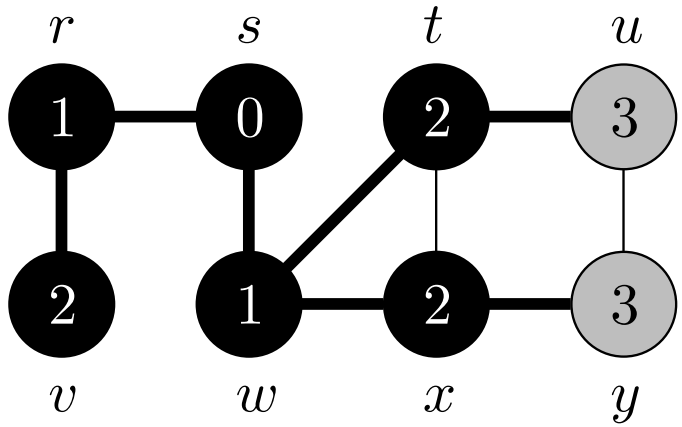


Q $\boxed{u}$ $\boxed{y}$

3 3

Breadth-first search

Example

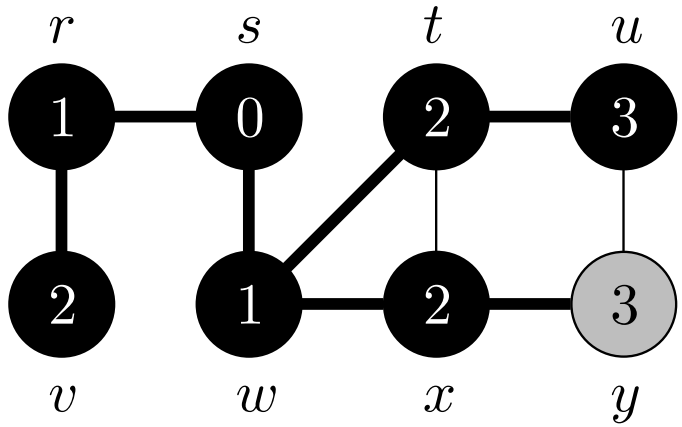


Q $\boxed{y}$

3

Breadth-first search

Example

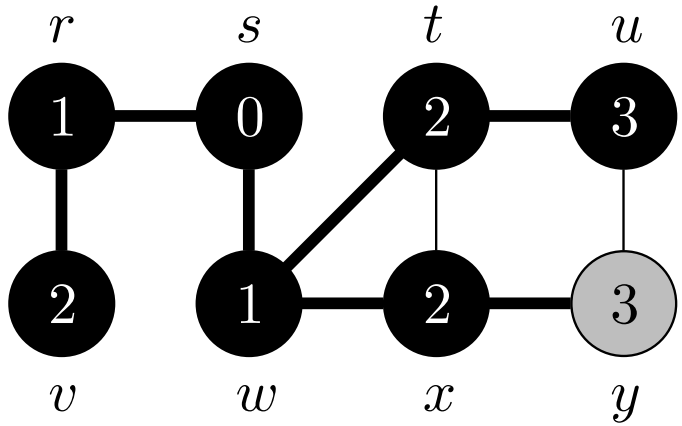


Q $\boxed{y}$

3

Breadth-first search

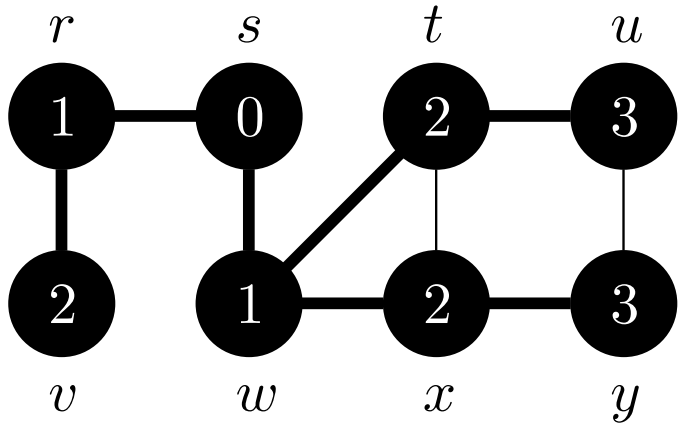
Example



Q $\square$

Breadth-first search

Example



Q $\square$

Breadth-first search

Running time

Assuming that the graph is represented as adjacency lists.

Breadth-first search

Running time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Breadth-first search

Running time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Cost of Queue Operations

After initialization no node becomes white again. Thus, each node is added to the queue at most one, which also means that it gets removed at most once.
The queue needs $\mathcal{O}(1)$ for these operations, thus, $\mathcal{O}(n)$ time in total.

Breadth-first search

Running time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Cost of Queue Operations

After initialization no node becomes white again. Thus, each node is added to the queue at most one, which also means that it gets removed at most once.
The queue needs $\mathcal{O}(1)$ for these operations, thus, $\mathcal{O}(n)$ time in total.

Cost of Edge Traversals

The outgoing edges of each node is iterated at most once when the node is removed from the queue. The sum of the lengths of all outgoing edge lists is $\mathcal{O}(m)$.

Breadth-first search

Running time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Cost of Queue Operations

After initialization no node becomes white again. Thus, each node is added to the queue at most one, which also means that it gets removed at most once.
The queue needs $\mathcal{O}(1)$ for these operations, thus, $\mathcal{O}(n)$ time in total.

Cost of Edge Traversals

The outgoing edges of each node is iterated at most once when the node is removed from the queue. The sum of the lengths of all outgoing edge lists is $\mathcal{O}(m)$.

Total running time $\mathcal{O}(n + m)$.

Depth-first search

Depth-first search tries to go as deep in the graph as possible.

Input

- graph $G(V, E)$, and
- start node s

Operation

Befores from s and follows one by one all outgoing edges of s, calling the same function recursively.
If at a node v there is no other outgoing edge to follow, it returns back to the node it came from and continues to follow unexplored edges.

Depth-first search

In order to keep track of progress, the algorithm colors the vertices of the graph using three colors a) white, b) gray, and c) black.

- All nodes start as white and they become gray and black afterwards.
- When a node first gets discovered it becomes gray and remains gray until all its neighbors are recursively explored. Then it becomes black.
- The algorithm records two times for each node, the time it becomes gray and the time that it becomes black.

Depth-first search

Pseudocode

---

dfs(G, s)

---

for *every vertex* $u \in V$ **do**
    $color[u] = \text{WHITE}$
    $\pi[u] = nil$
end
$time = 1$
dfsvisit(G, s);

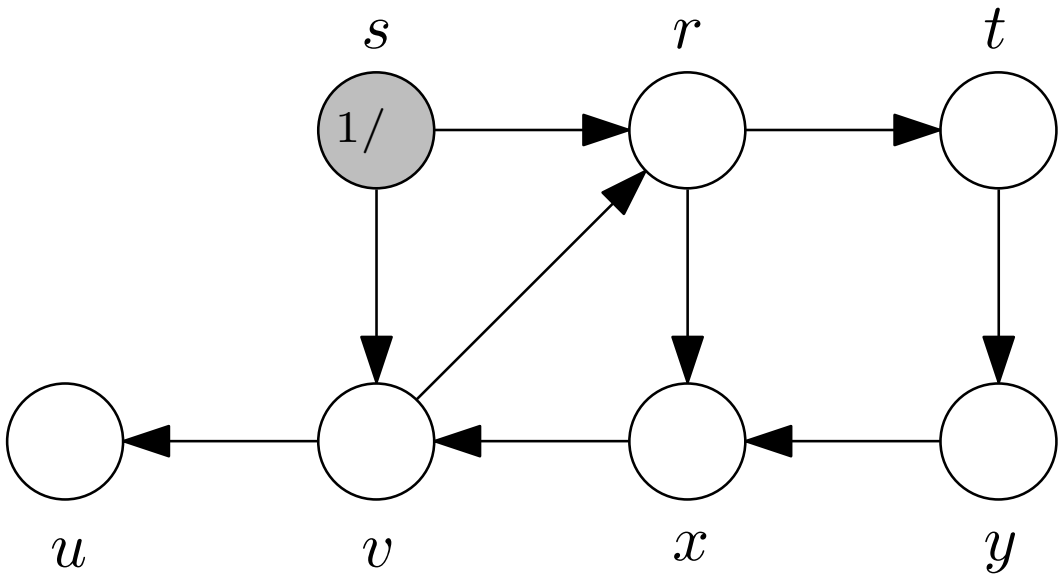
---

dfsvisit(G, u)

$color[u] = \text{GRAY}$
$d[u] = time$
$time = time + 1$
for *every neighbor* v *of* u **do**
    if $color[v] = WHITE$ **then**
        $\pi[v] = u$
        dfsvisit(G, v)
    end
end
$color[u] = \text{BLACK}$
$f[u] = time$
$time = time + 1$

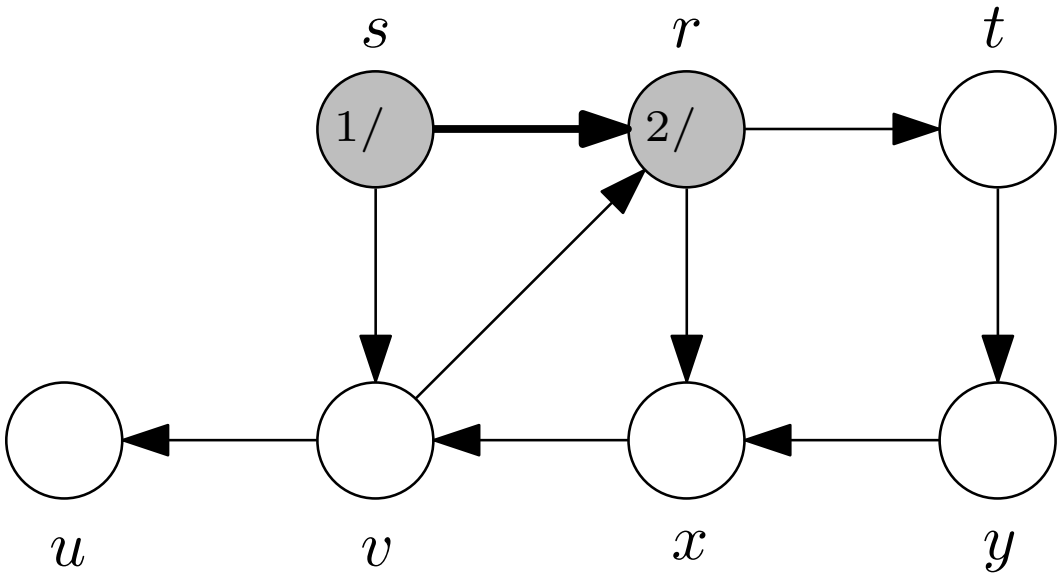
Graph Search

Depth-first search



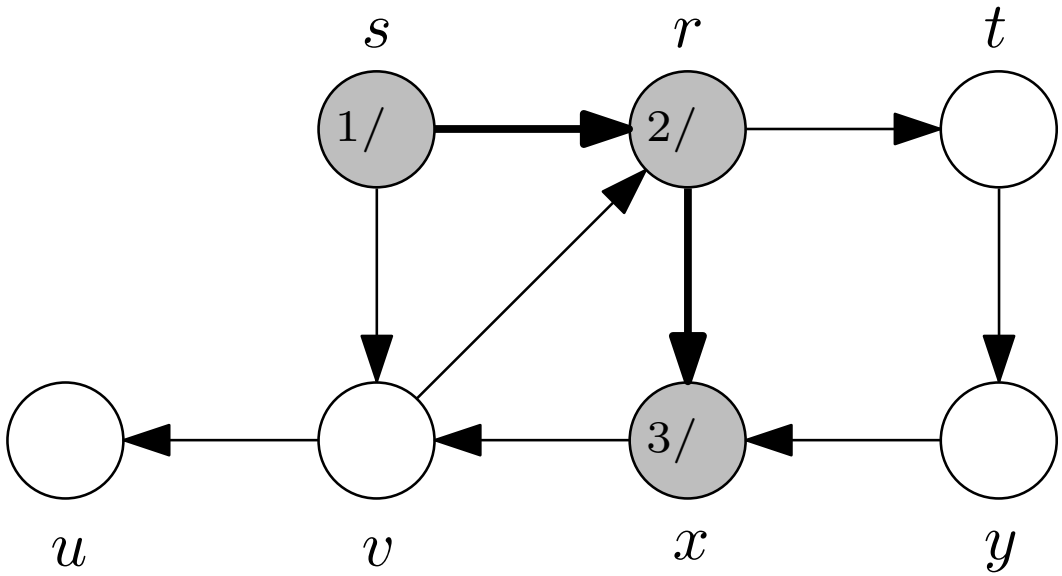
Graph Search

Depth-first search



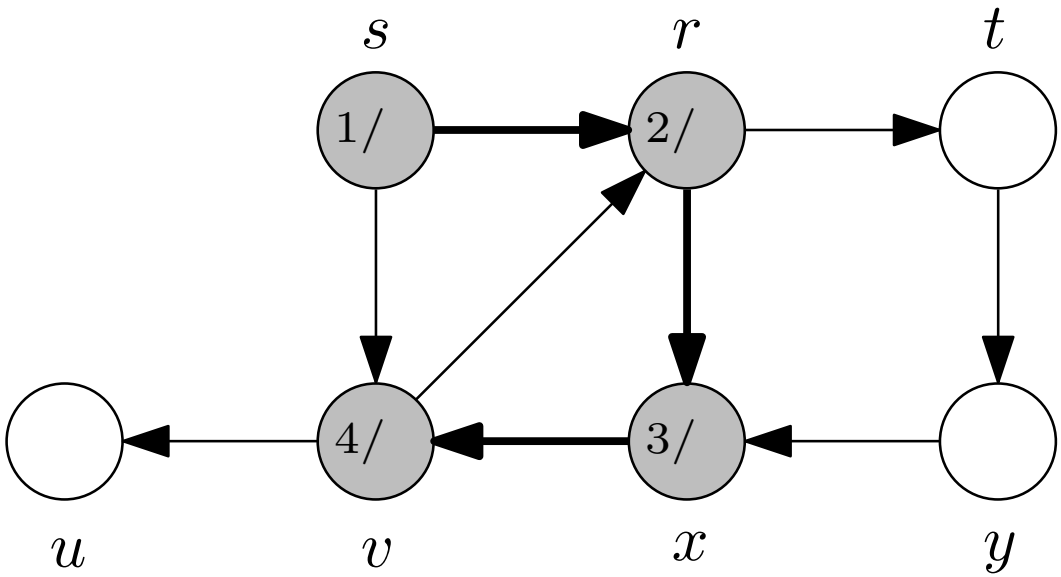
Graph Search

Depth-first search



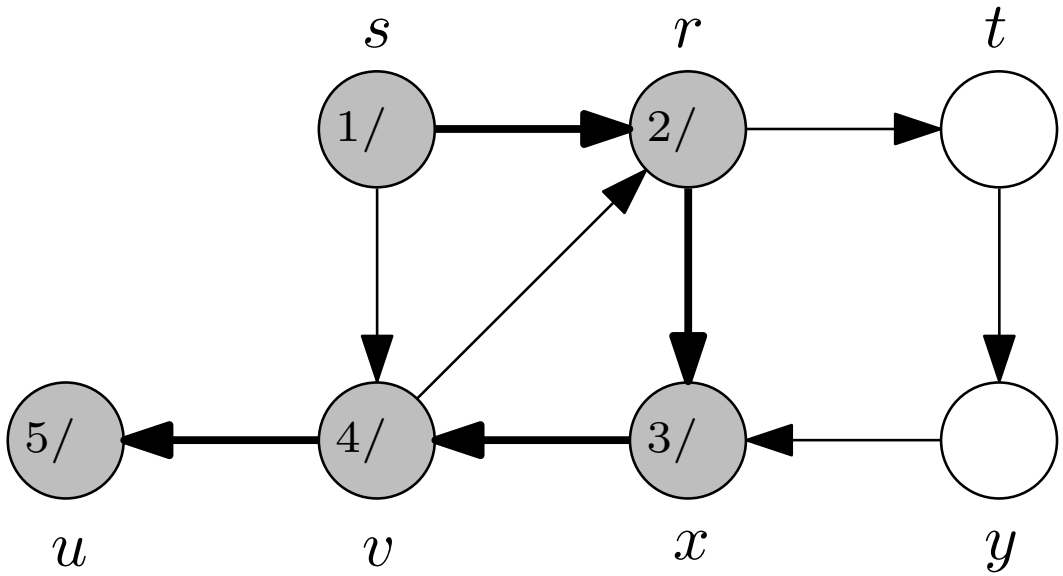
Graph Search

Depth-first search



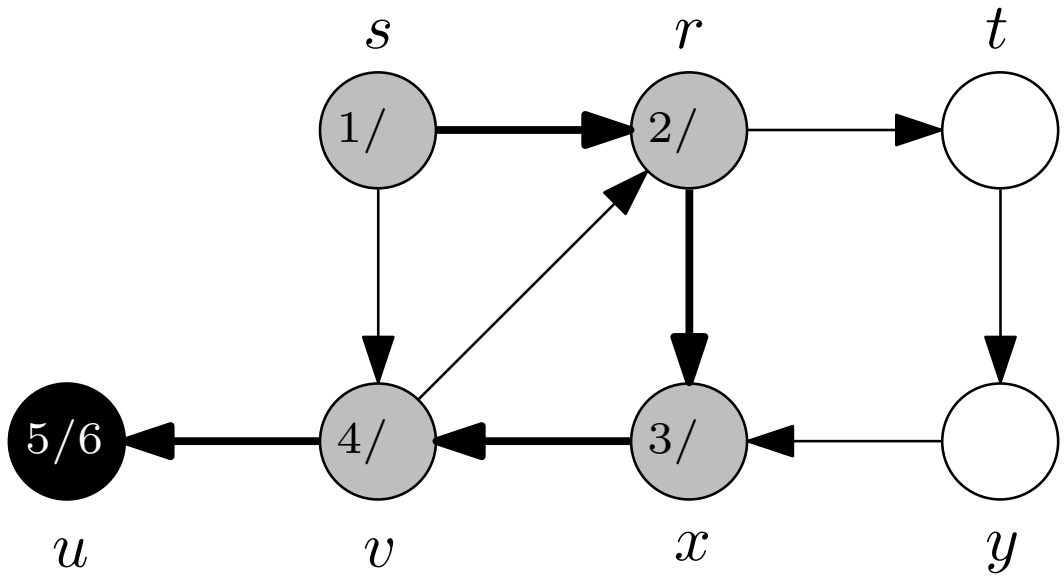
Graph Search

Depth-first search



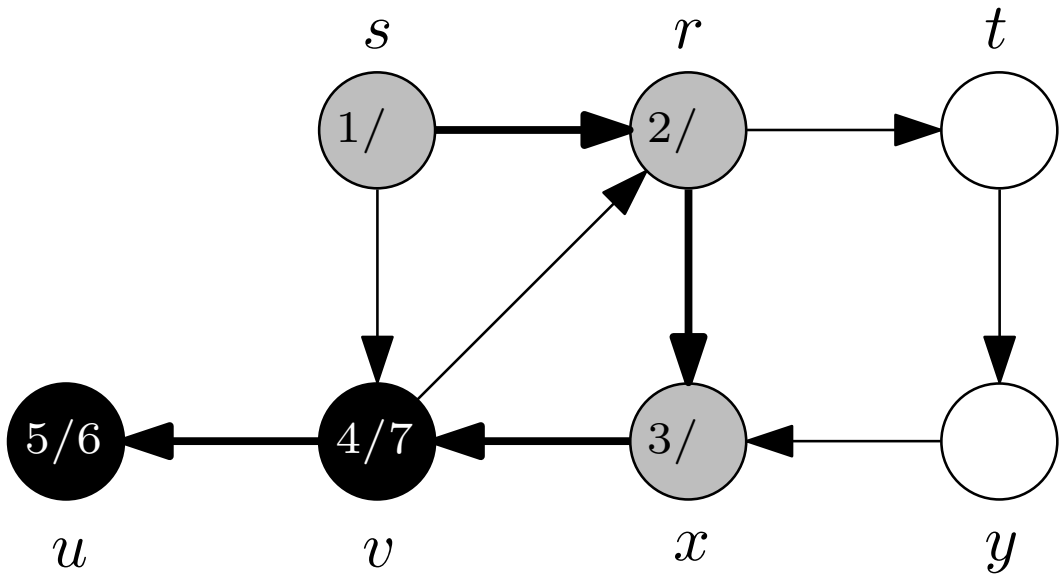
Graph Search

Depth-first search



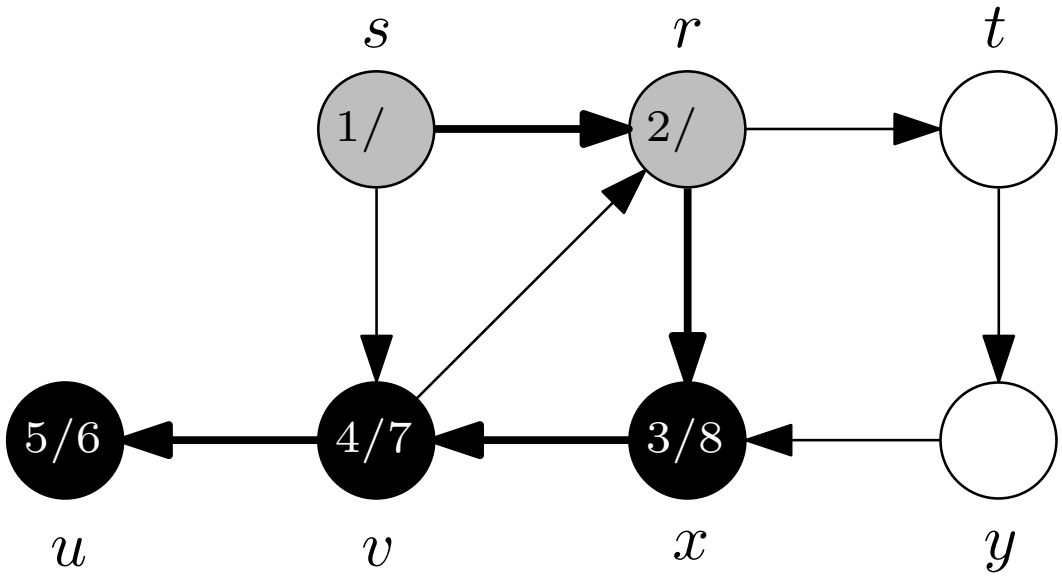
Graph Search

Depth-first search



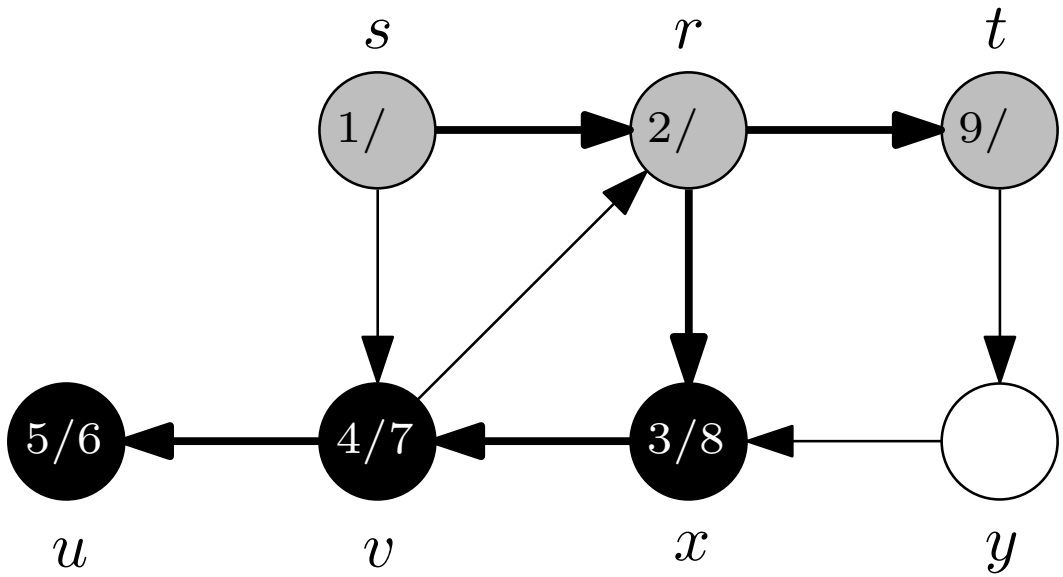
Graph Search

Depth-first search



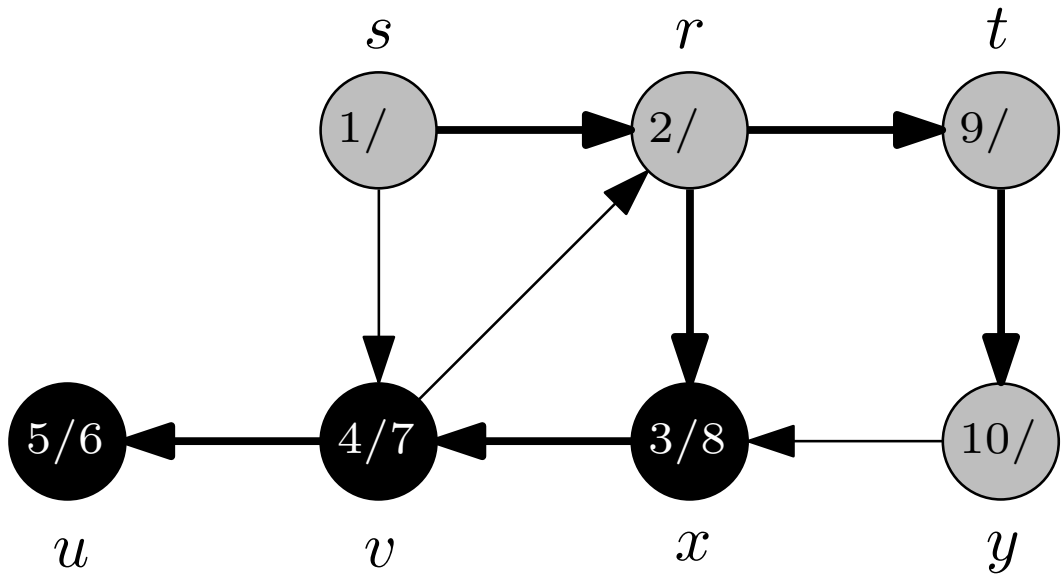
Graph Search

Depth-first search



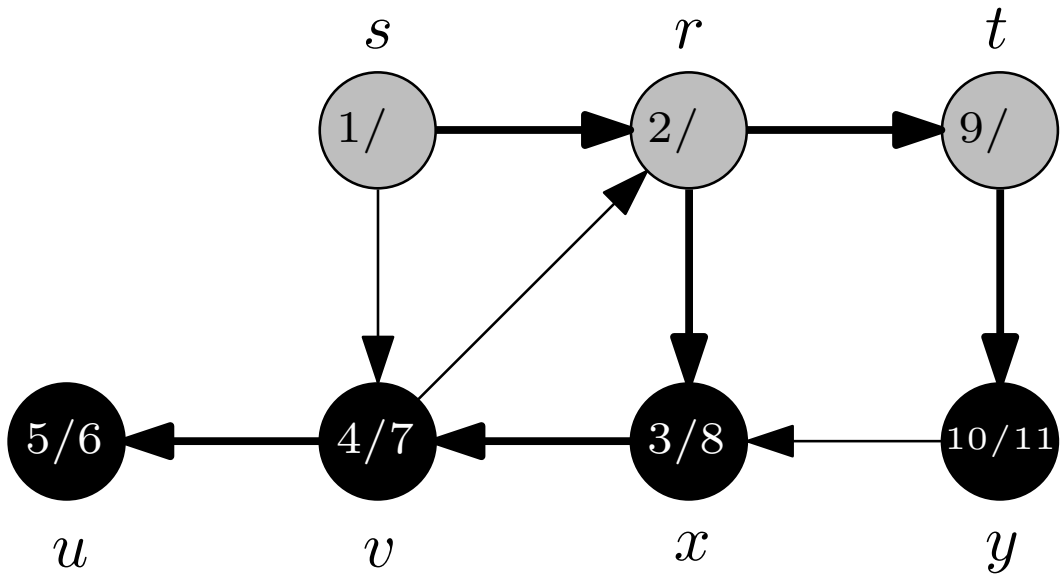
Graph Search

Depth-first search



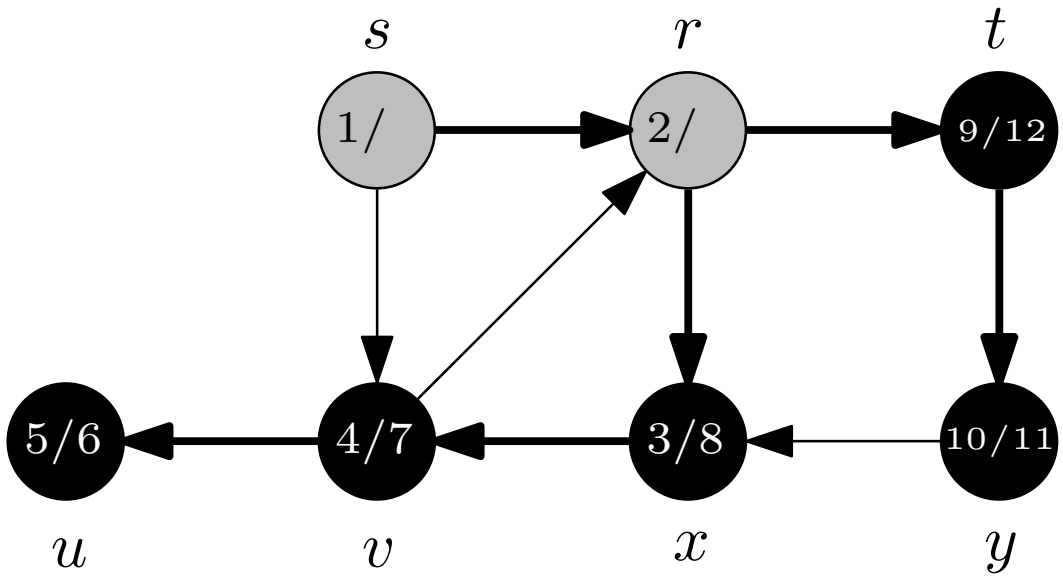
Graph Search

Depth-first search



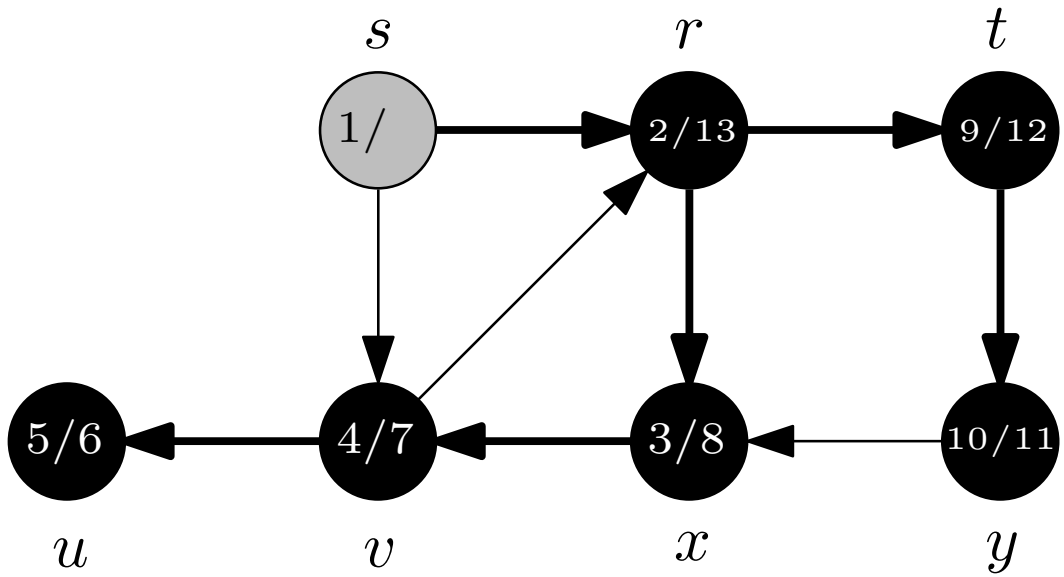
Graph Search

Depth-first search



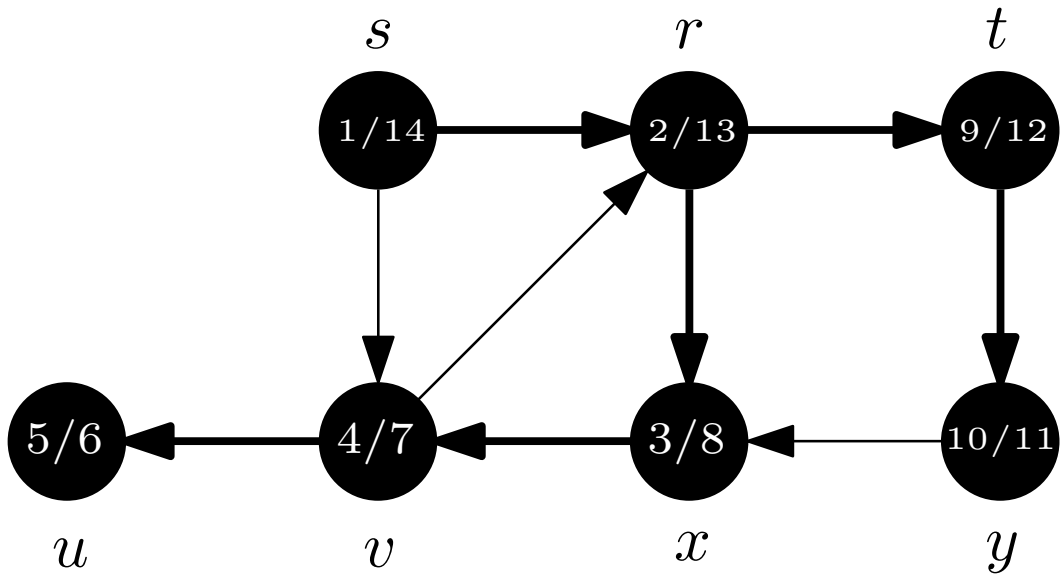
Graph Search

Depth-first search



Graph Search

Depth-first search



Depth-first search

Running Time

Assuming that the graph is represented as adjacency lists.

Depth-first search

Running Time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Depth-first search

Running Time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Calls to *dfsvisit*(G, v)

The recursive function *dfsvisit*(G, v) is called once per node since it is called only on white nodes and the first thing it does is to change their color to gray.

A call to *dfsvisit*(G, v) costs as must as the length of the outgoing edge list of v. The sum of these outgoing edge lists for all vertices is (m).

Depth-first search

Running Time

Assuming that the graph is represented as adjacency lists.

Initialization

$\mathcal{O}(n)$ since we initialize arrays which have one element per node.

Calls to *dfsvisit*(G, v)

The recursive function *dfsvisit*(G, v) is called once per node since it is called only on white nodes and the first thing it does is to change their color to gray.

A call to *dfsvisit*(G, v) costs as must as the length of the outgoing edge list of v. The sum of these outgoing edge lists for all vertices is (m).

Total running time $O(n + m)$.

Reading and Sources

- Sections 8.1, 8.2, 8.3, 8.4, 9.1, and 9.2

  Kurt Mehlhorn Peter Sanders. Algorithms and Data Structures, The basic toolbox, 1/e, 2014.

- Sections 3.7, and 5.8

  Robert Sedgewick. Algorithms in C: Fundamentals, Data Structures, Sorting, Searching. 3/e, Addison-Wesley Professional, 1997.

- Sections 3.1, 3.2, 3.3, 4.1, and 4.2

  Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani, Algorithms, McGraw-Hill Education, 1/e, 2006

- Sections 22.1, 22.2, and 22.3

  Cormen, Leiserson, Rivest and Stein. Introduction to Algorithms. 3/e, MIT Press. 2009.