04-630
Data Structures and Algorithms for Engineers

George Okeyo
Carnegie Mellon University Africa
gokeyo@andrew.cmu.edu

Lecture 17: Graphs

Outline

- Graphs: Preliminaries
  - Applications
  - Definitions
- Graph representation:
  - adjacency matrix
  - adjacency list
- Graph traversal
  - BFS, DFS
- Applications of graph search algorithms

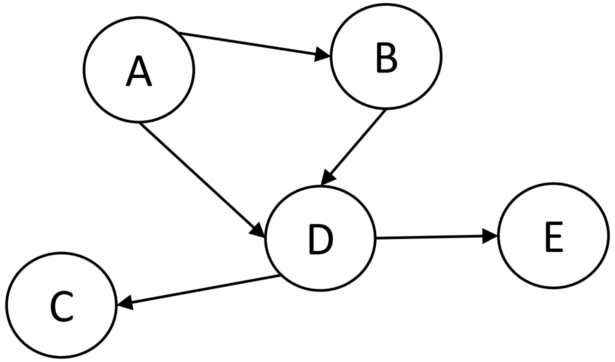
Graphs: Applications

- Mapping
- Transportation
- Electrical engineering
- Computer networks

Graphs: Definitions

- A graph ***G***, formally ***G=(V,E),*** is composed of a set of ***vertices V*** and a set of ***edges*** $\boldsymbol{E \subset V \times V}$.
  - A ***vertex*** refers to a node in a graph.
  - ***Edges*** connect vertices.
- An edge ***e=(u,v)*** is a pair of vertices for ***directed*** graphs.
- For an undirected graph, an edge between **u** and **v** is represented by two pairs $\boldsymbol{(u,v) \in E}$ and $\boldsymbol{(v,u) \in E}$.

Graphs: directed graphs (digraphs)



V={A, B, C, D, E}

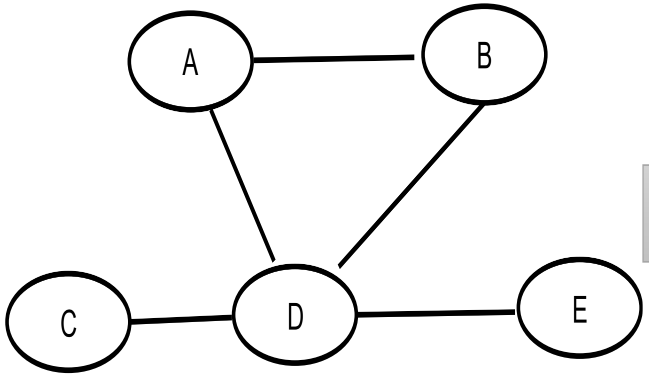
E={(A,B), (A,D),(B,D),(D,C), (D,E)}

Has ***directed edges***.

An edge ***(u,v)*** is **directed** if the pair ***(u,v)*** is ***ordered*** such that ***u*** precedes ***v***.

End points: u and v. In the case of ***directed graphs,*** u is ***origin*** and v is the ***destination***.

Graphs: undirected graphs



Has ***undirected edges***.

An edge ***(u,v)*** is un**<u>directed</u>** if the pair ***(u,v)*** is ***not ordered*** such that ***u*** precedes ***v***.

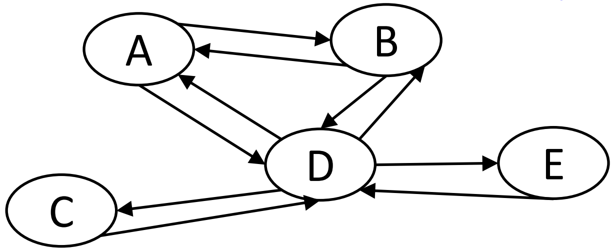
V={A, B, C, D, E}

E={(A,B),(B,A),(A,D),(D,A),(B,D),(D,B),(D,C),(C,D), (D,E),(E,D)}

Or

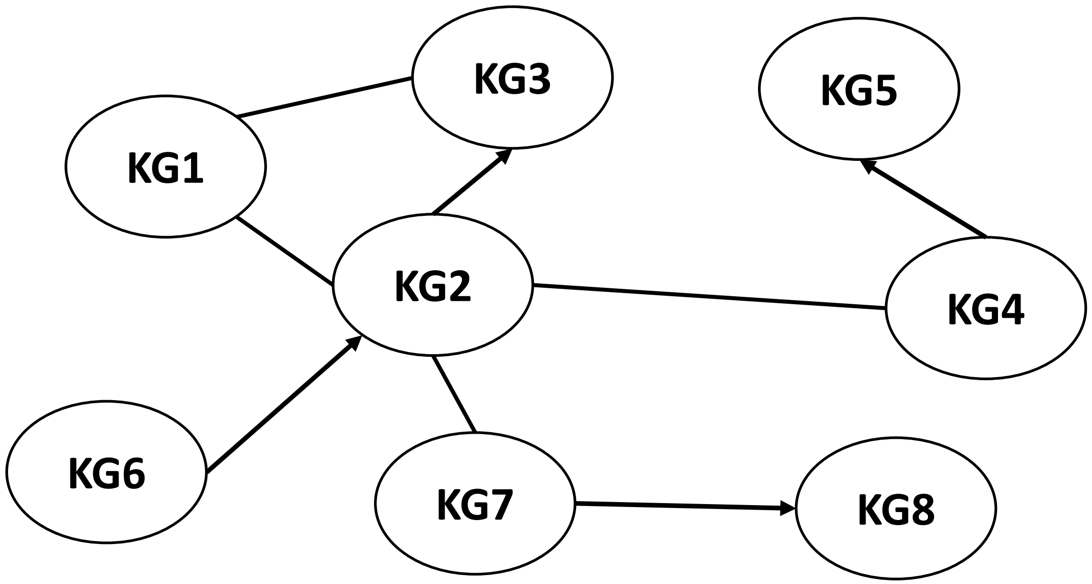
E={{A,B},{A,D},{B,D},{D,C},{D,E}}, set of all 2-element subsets of V

1. It is also possible to have a ***mixed graph,*** with both directed and undirected edges.
2. Both undirected and mixed graphs can be <u>converted</u> *(if necessary)* to a directed graph.



An undirected graph converted to a directed graph.

Graphs: mixed graph example



Exercise: 1) Roads in Kigali? What kind of scenario(s) in city road network design are represented by the graph?
2) Give another example of a real-world problem that can be modelled with a mixed graph.

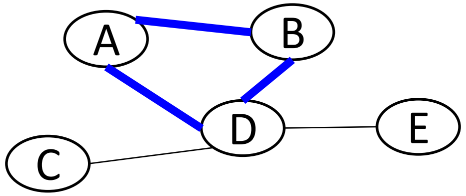
Graphs: definitions

- ***Adjacent vertices***: Vertices u and v are adjacent if there is an edge with u and v as vertices, i.e. $(u,v) \in E$.
- ***Degree*** of a ***vertex***: number of adjacent vertices.
- An edge ***e*** is ***incident*** on a vertex if the vertex is one of the edges endpoints.
- ***Path***: sequence of vertices v_1, v_2,….v_n, such that v_{k+1} is adjacent to v_k for k=1…n-1.

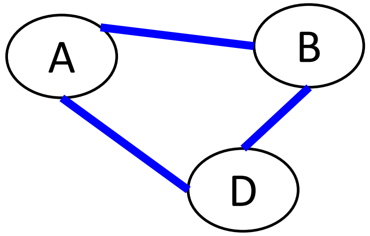
Graphs: definitions

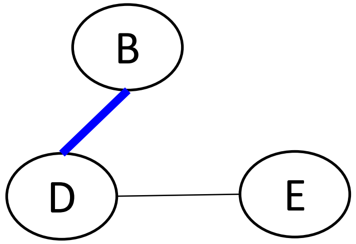
Simple path: a path with no repeated vertices, e.g. A, B, D.

Cycle: a simple path where the first and the last vertices are the same, e.g. A,B,D,A.



Connected graph: any two vertices are connected by some path.





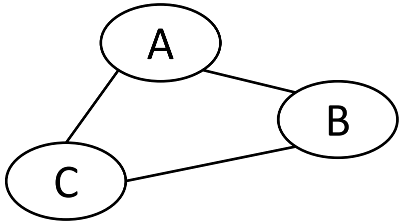
Subgraph: a subset of vertices and edges forming a graph, i.e., **G'=(V',E')** is a subgraph of **G(V,E)** if **V'** $\subseteq$ **V, E'** $\subseteq$ **E** and **G'** is a graph.

Connected component: A connected component of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices.

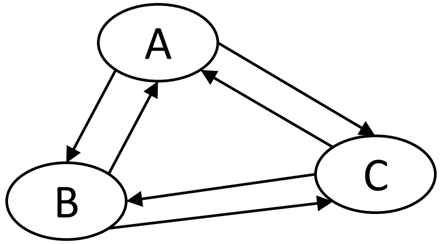
Tree: connected graph without cycles.

Graphs: definitions

Complete graph: every vertex is connected to every other vertex.

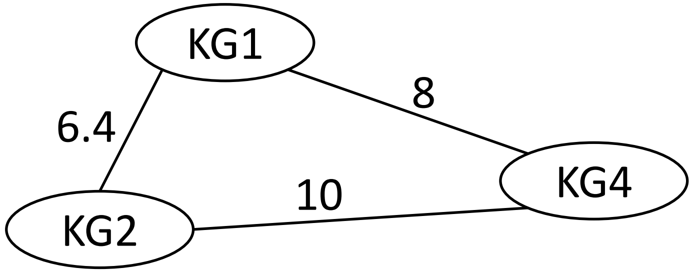


Complete undirected graph



Complete directed graph

Weighted graph: a graph in which every edge has an associated value or weight.



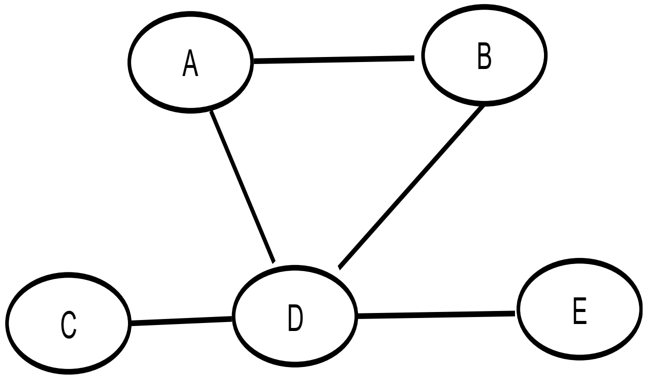
Graph representation:

A graph G=(V,E) with **n vertices** and **m edges** can be represented using two data structures:

- ***Adjacency matrix:*** an **n × n** matrix M (**an array**), where each element
  - M[i,j]=1 if (i,j) is an edge of G, and
  - M[i,j]= 0 if it is not.
  - For a weighted graph M[i,i]=w, the weight of the edge if (I,j) is an edge and M[i,i]=∞ if it is not.
- ***Adjacency list:*** uses a **linked list** that stores the sequence of vertices that are adjacent to each vertex.

Graph representation: adjacency matrix

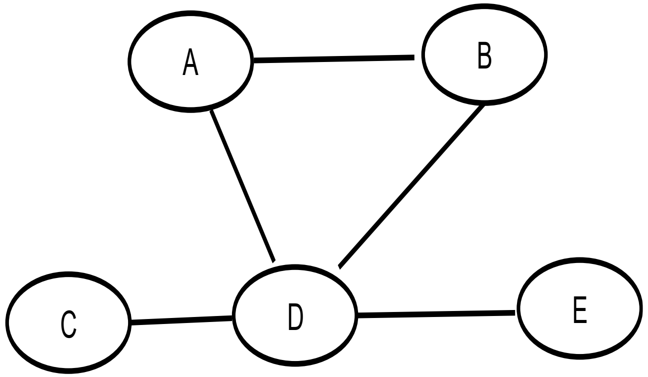
- Space requirement: $\Theta(|V|^2)$



	A	**B**	**C**	**D**	**E**
A	**0**	**1**	**0**	**1**	**0**
B	**1**	**0**	**0**	**1**	**0**
C	**0**	**0**	**0**	**1**	**0**
D	**1**	**1**	**1**	**0**	**1**
E	**0**	**0**	**0**	**1**	**0**

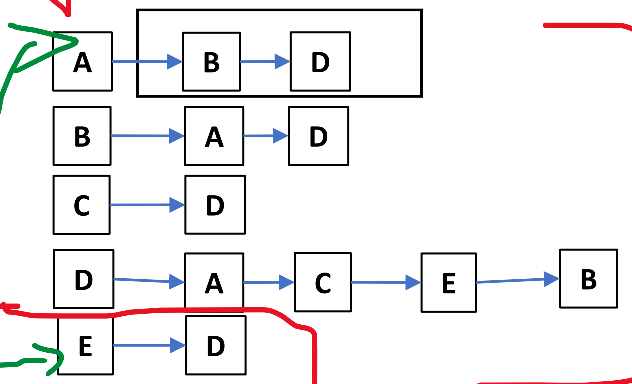
Graph representation: adjacency lists

- Space requirement: $\Theta(|V|+|E|)$



Can be held in array: *adjacency lists*

List of vertices



Each vertex is also the head of a list to all adjacent vertices.

Adjacency list for vertex E

Code Implementation: Adjacency List

```
struct node
{
int dest;
node *next;//next node in list
};

struct adjList
{
node *head;
};
```

```
class graph
{
private:
        int nvertices;
        adjList *adjLists;//adjacency info: list of edges
public:
        graph(int nvertices);//constructor
        node *createListNode(int dest);
        void addEdge(int srcData,int destData);
        void printGraph();
};
graph::graph(int nvertices)
{
        this->nvertices=nvertices;//set up the number of vertices
        adjLists=new adjList[nvertices];//create an adjacency list for each vertex
        for(int i=0;i<nvertices;i++)
                adjLists[i].head=NULL;//set each head of the list to NULL.
}
```

nvertices=5

adjacency list-3

head=NULL	head=NULL	head=NULL	head=NULL	head=NULL
0	1	2	3	4

Code Implementation: Adjacency List

```
/*
Utility method that creates nodes
*/
node *graph::createListNode(int dest)
{
	node * newNode=new node;
	newNode->dest=dest;
	newNode->next=NULL;
	return newNode;
}
```

Code Implementation: Adjacency List

```
/*This should create two nodes - the source and destination nodes.*/
void graph::addEdge(int src, int dest)
{
        //Adding a new node
        node * newNode=createListNode(dest);
        newNode->next=adjLists[src].head;//make the current head of the list to be the next node for the new node
        adjLists[src].head=newNode;//make the new node the head of the adjacency list.

        //
        newNode=createListNode(src);
        newNode->next=adjLists[dest].head;
        adjLists[dest].head=newNode;
}
```

Code Implementation: Adjacency List

```
void graph::printGraph()
{
        int v;
        cout<<"Adjacency lists\n";
        for(v=0;v<nvertices;v++)//step through each adjacency list
        {
                node * aNode=adjLists[v].head;
                cout<<"\n["<<v<<"]";
                while(aNode)
                {
                        cout<<"->"<<aNode->dest;
                        aNode=aNode->next;
                }
                cout<<endl;
        }
}
```

Graph Traversal

- ***Traversal***: systematic procedure for exploring a graph through examining all its vertices and edges.
  - Maintain information about the state of a vertex to keep track of the traversal.
  - States:
    - ***Undiscovered***: the vertex is in the initial untouched state.,
    - ***Discovered***: vertex has been found but its edges have not been processed.
    - ***Processed***: the state of the vertex after we have visited all its edges.
- Traversal algorithms:
  - Breadth-first search (BFS)
  - Depth-first search (DFS)

Graph traversal

- Begins with a starting vertex, $s \in V$
- Coloring vertices:
  - Undiscovered vertices are colored ***white***.
  - Color a vertex ***gray*** if it has been discovered but all its edges have not been explored.
  - Color a vertex ***black*** after all its adjacent vertices have been discovered.

Breadth first search (BFS)

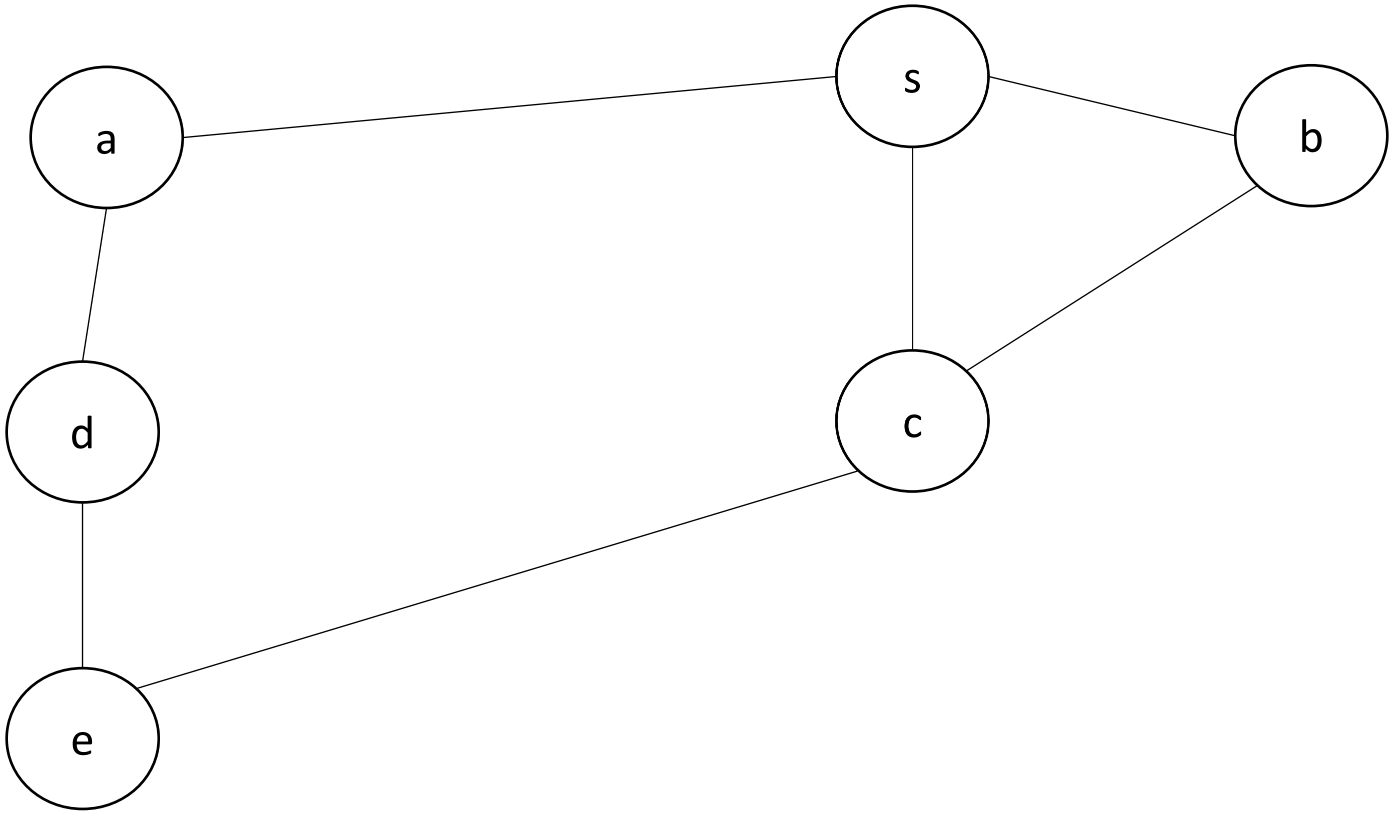
- Uses a FIFO queue to ensure oldest unexplored vertices are processed first.
- Process:
  - The starting vertex s is assigned the distance 0.
  - All vertices that are 1 edge away from s are visited and assigned a distance 1.
  - Search proceeds to vertices 2 edges away, which are assigned distance 2.
  - Process continues until each vertex has been assigned a level/distance.
- The level of each vertex v corresponds to the length of the shortest path from s to v.

BFS algorithm

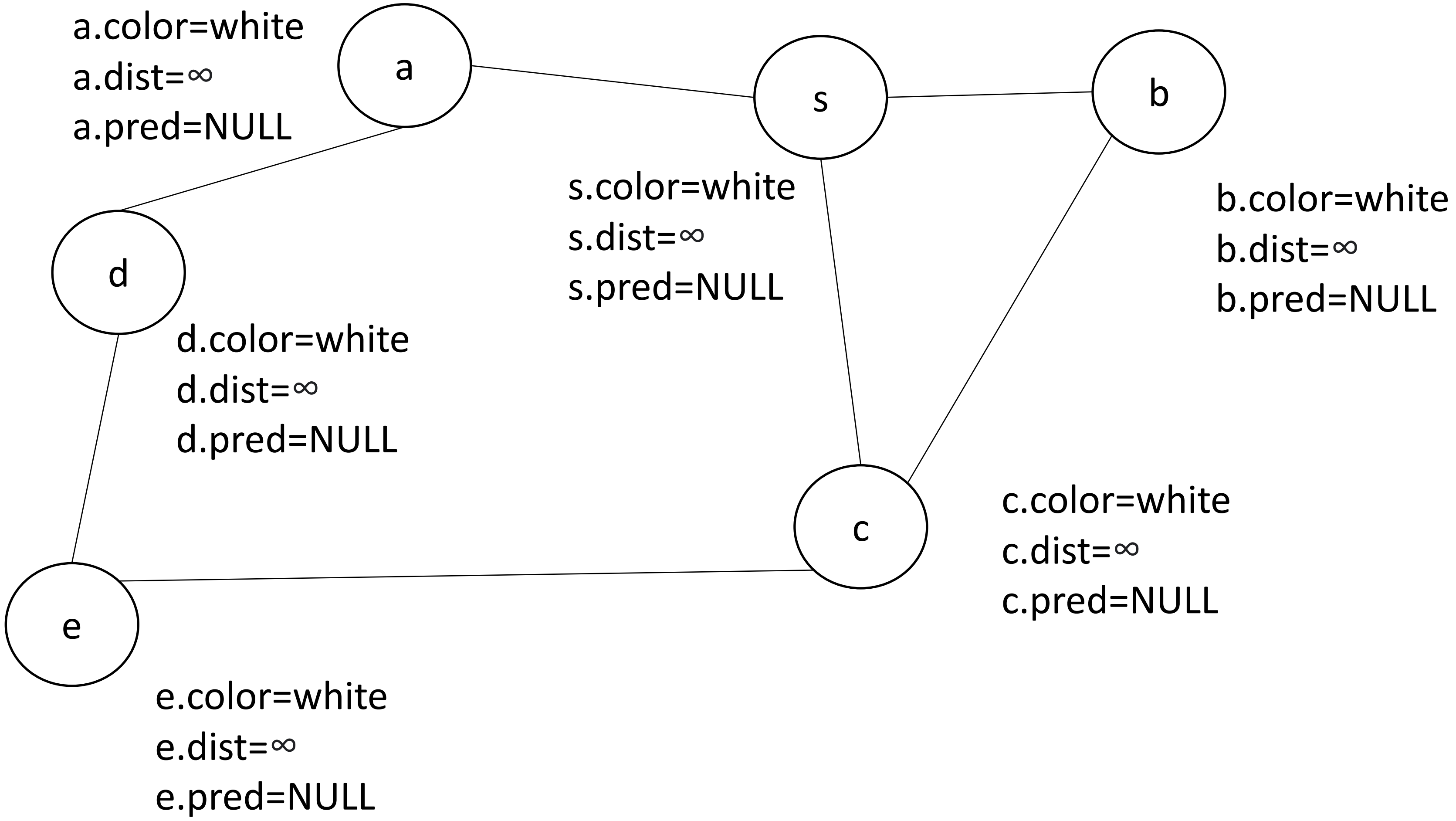
```
BFS(G,s):
##Initialize all vertices
For u ∈ G.V Do
        u.color=white
        u.dist= ∞
        u.pred=NULL
##Initialize BFS
s.color=gray
s.dist=0
Q={s}##a FIFO queue

## handle all u's children
While Q is not empty Do
        dequeue(Q,u)
        For v ∈ u.adj Do
                If v.color=white Then
                        v.color=gray
                        v.dist=u.dist+1
                        v.pred=u
                        enqueue(Q,v)
        u.color=black
```

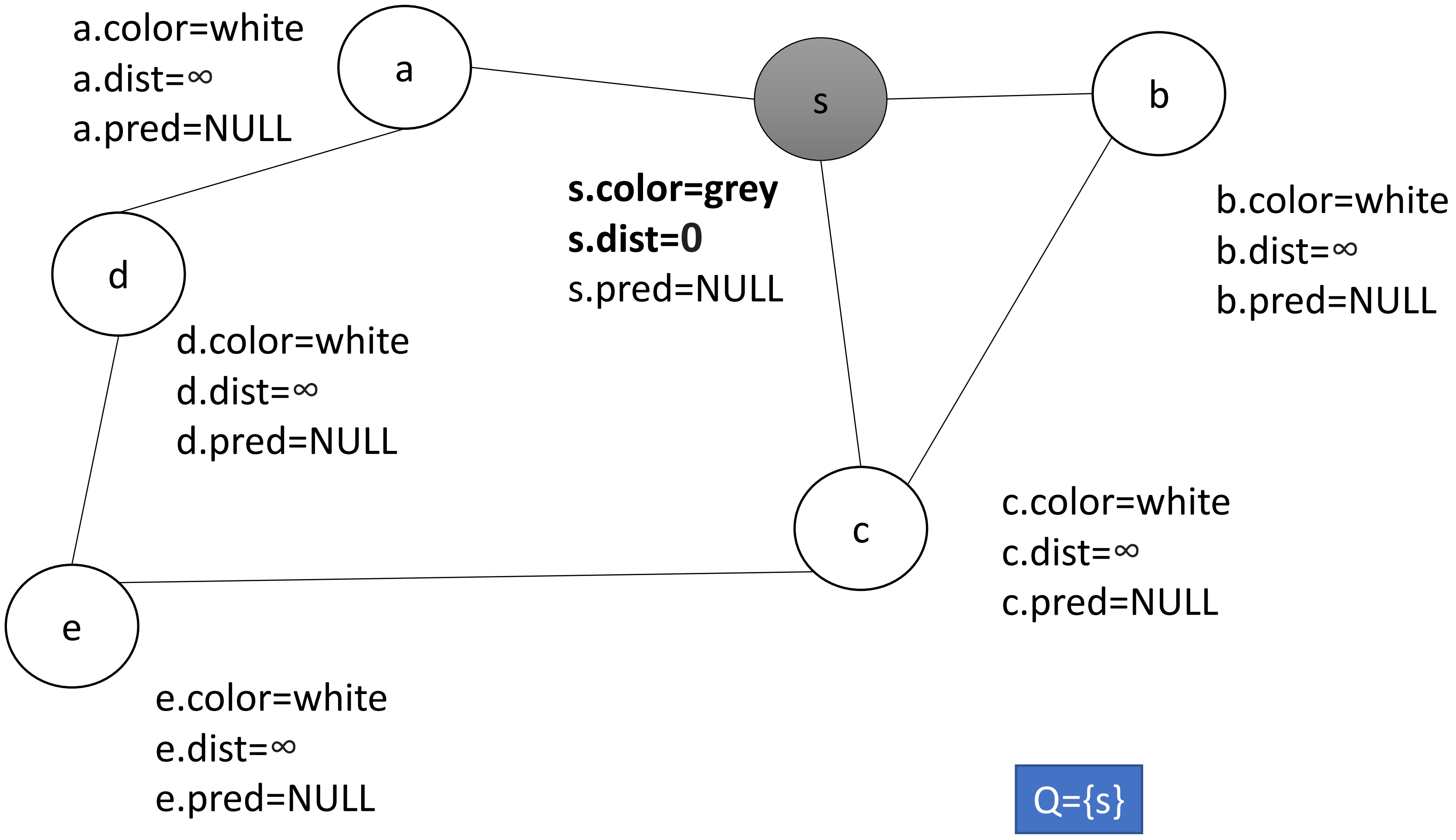
BFS Example



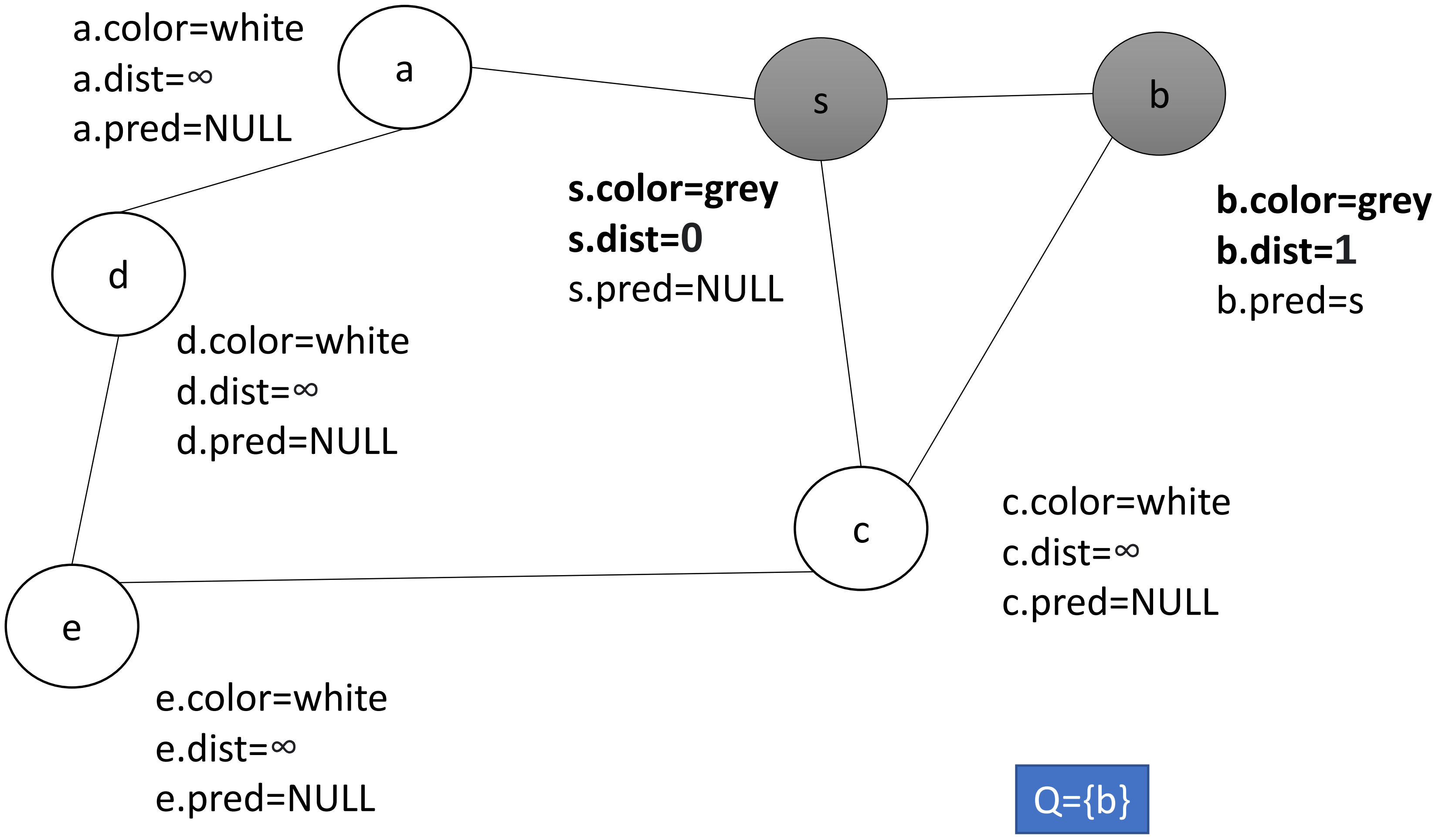
BFS Example


a.color=white
a.dist=∞
a.pred=NULL
a
s
b
s.color=white
s.dist=∞
s.pred=NULL
b.color=white
b.dist=∞
b.pred=NULL
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=white
c.dist=∞
c.pred=NULL
e
e.color=white
e.dist=∞
e.pred=NULL

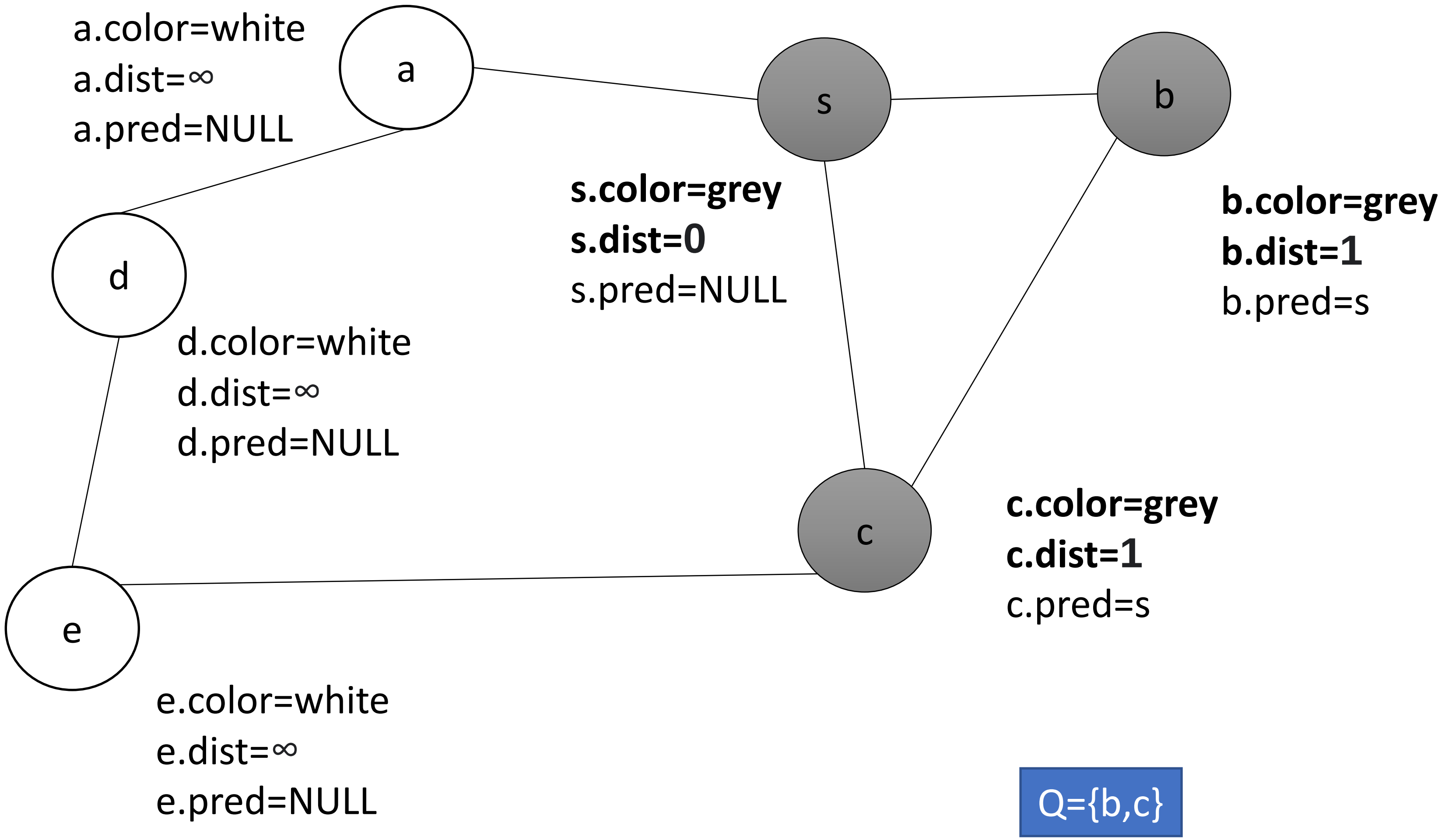
BFS Example


a.color=white
a.dist=∞
a.pred=NULL
a
s
b
s.color=grey
s.dist=0
s.pred=NULL
b.color=white
b.dist=∞
b.pred=NULL
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=white
c.dist=∞
c.pred=NULL
e
e.color=white
e.dist=∞
e.pred=NULL
Q={s}

BFS Example

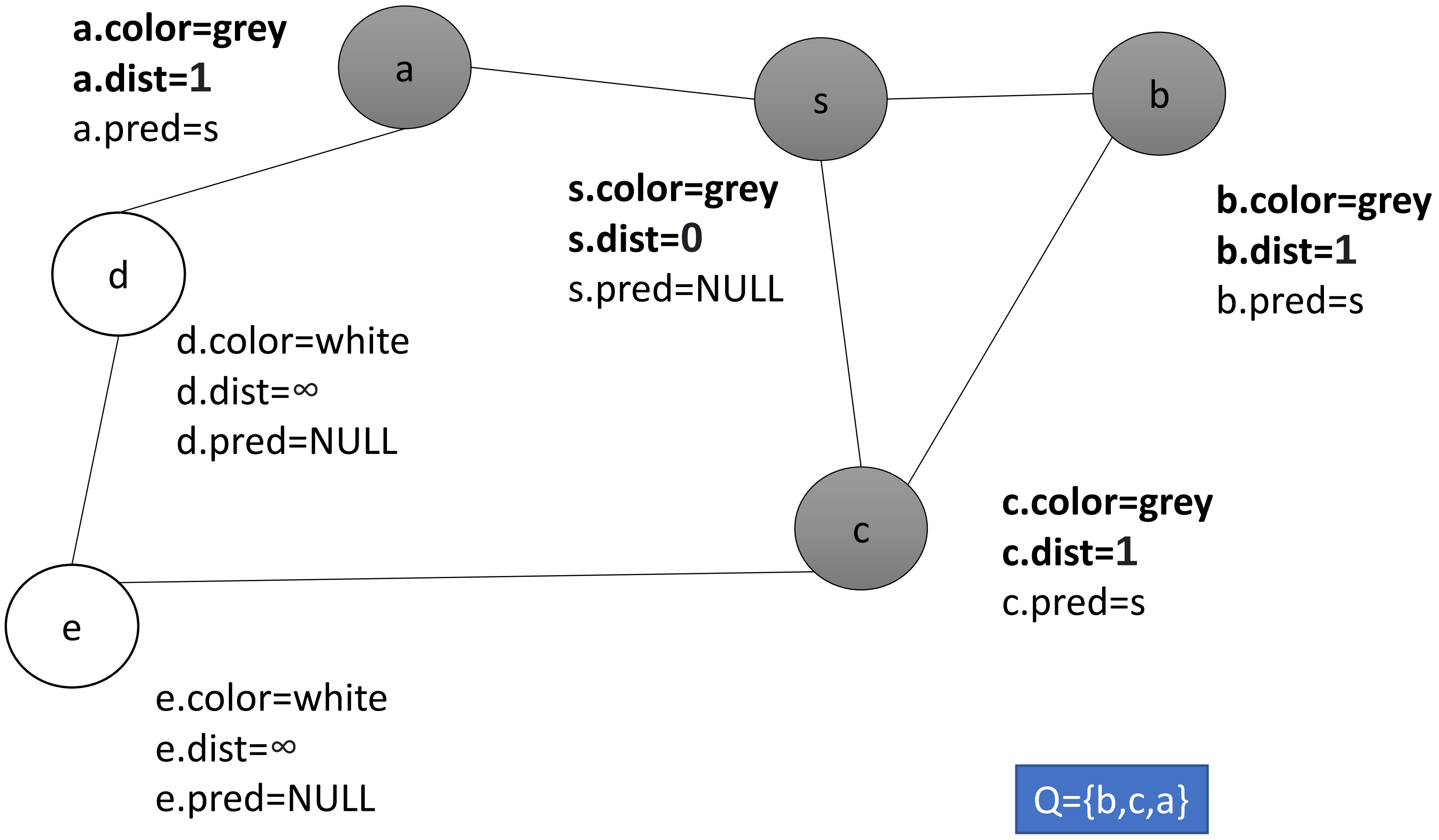

a.color=white
a.dist=∞
a.pred=NULL
a
s
b
s.color=grey
s.dist=0
s.pred=NULL
b.color=grey
b.dist=1
b.pred=s
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=white
c.dist=∞
c.pred=NULL
e
e.color=white
e.dist=∞
e.pred=NULL
Q={b}

BFS Example

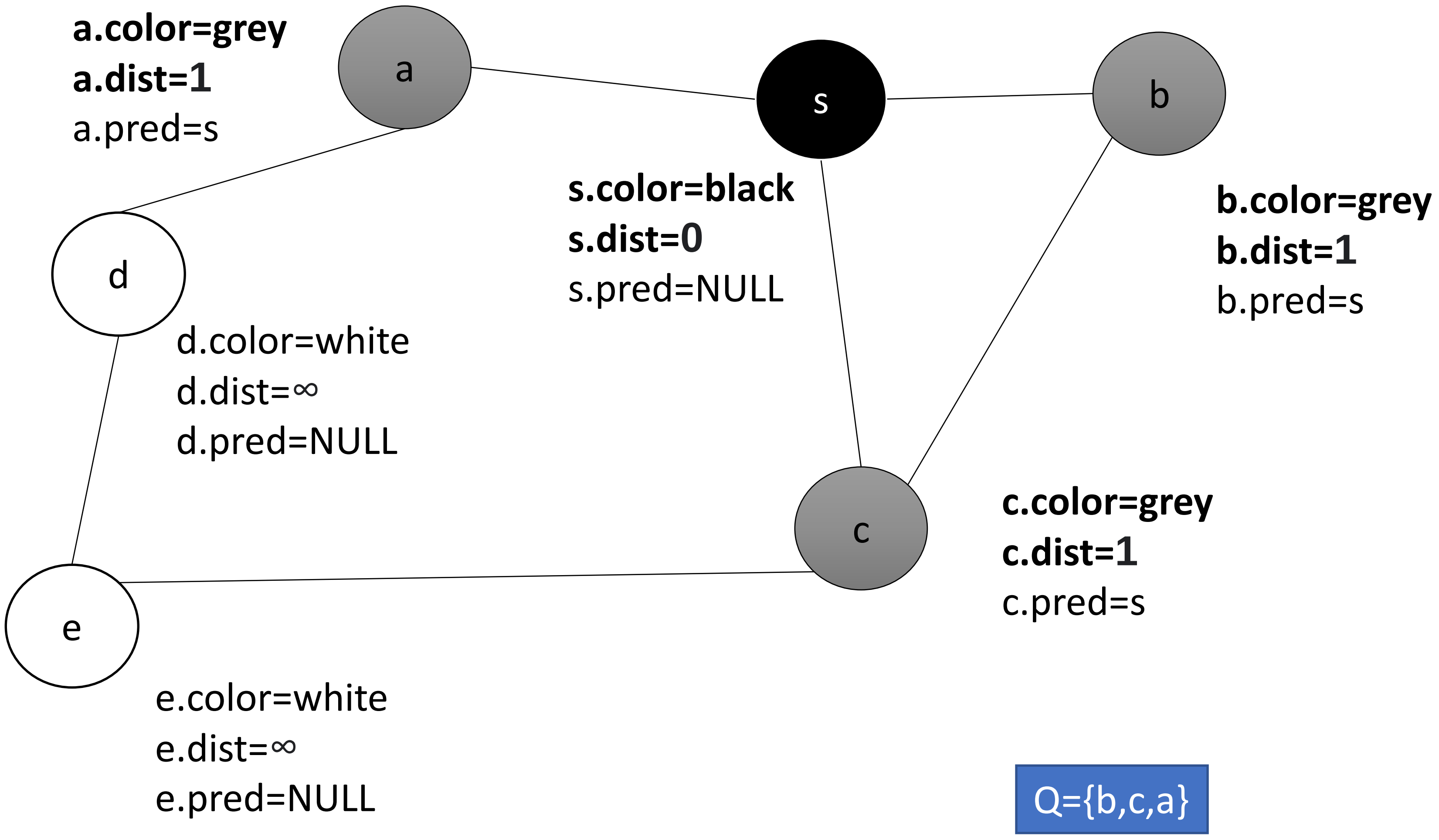


Q={b,c}

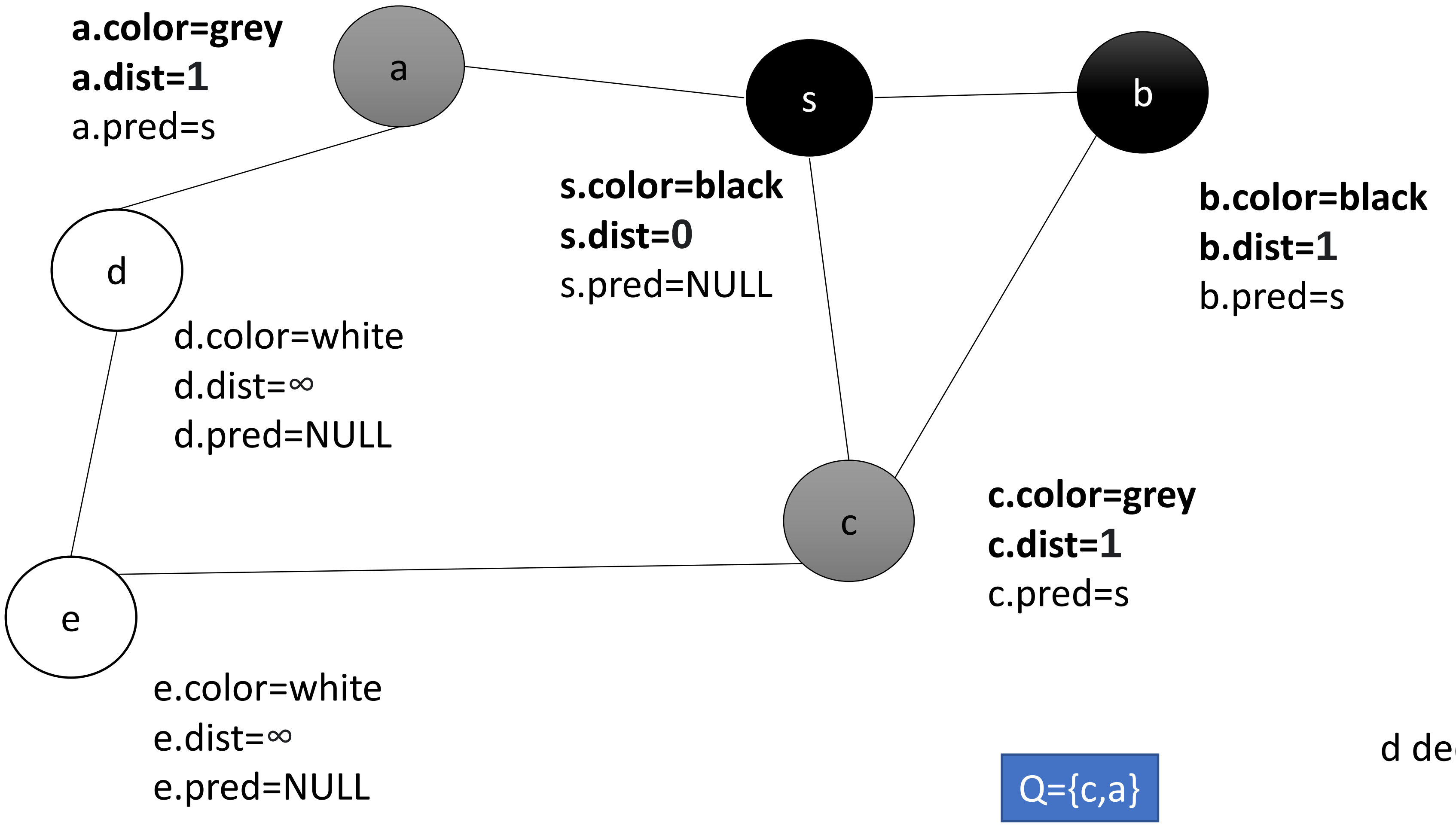
BFS Example


a
s
b
d
c
e
a.color=grey
a.dist=1
a.pred=s
s.color=grey
s.dist=0
s.pred=NULL
b.color=grey
b.dist=1
b.pred=s
d.color=white
d.dist=∞
d.pred=NULL
c.color=grey
c.dist=1
c.pred=s
e.color=white
e.dist=∞
e.pred=NULL
Q={b,c,a}

BFS Example


a.color=grey
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=grey
b.dist=1
b.pred=s
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=grey
c.dist=1
c.pred=s
e
e.color=white
e.dist=∞
e.pred=NULL
Q={b,c,a}

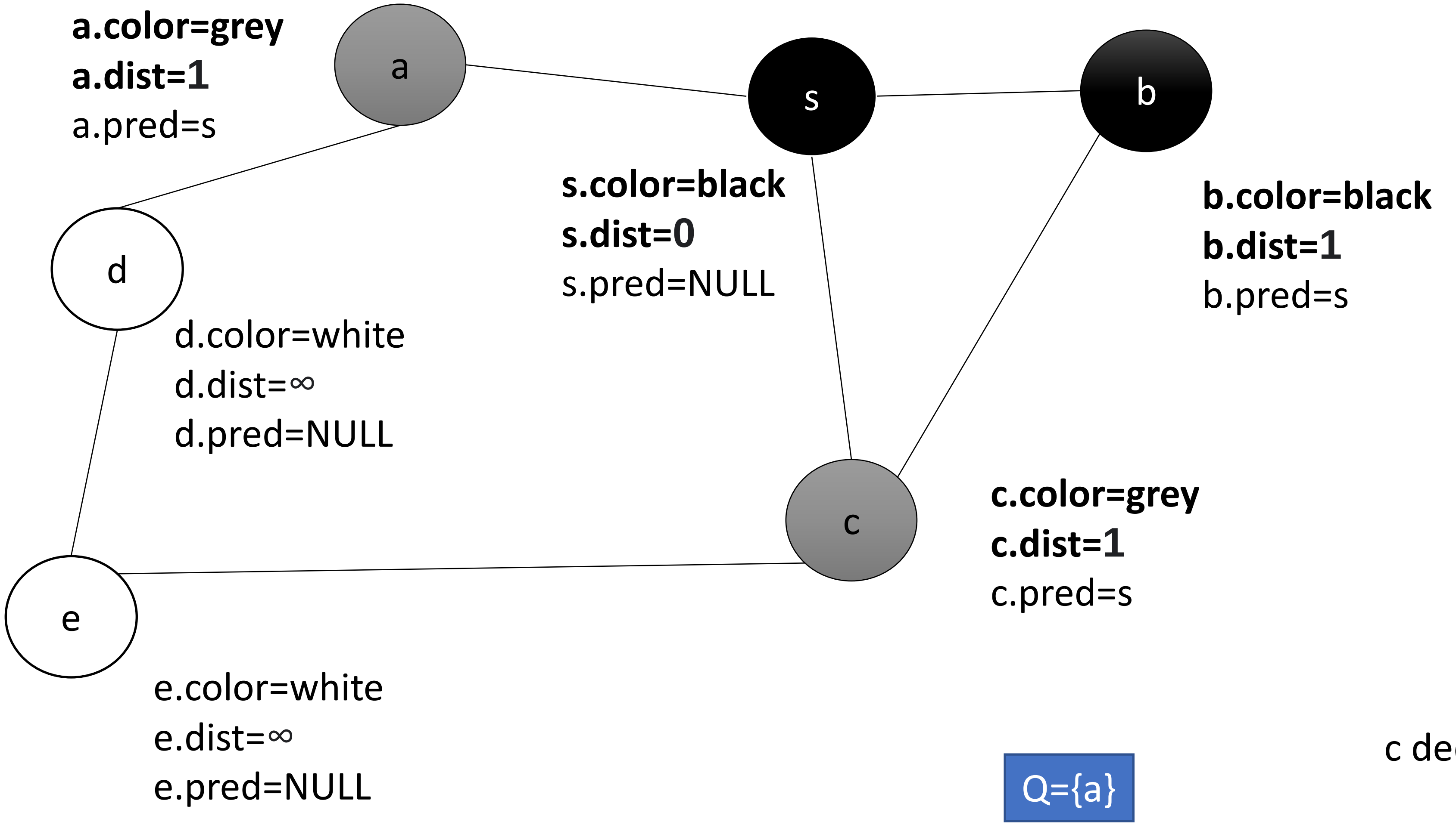
BFS Example



Q={c,a}

d dequeued

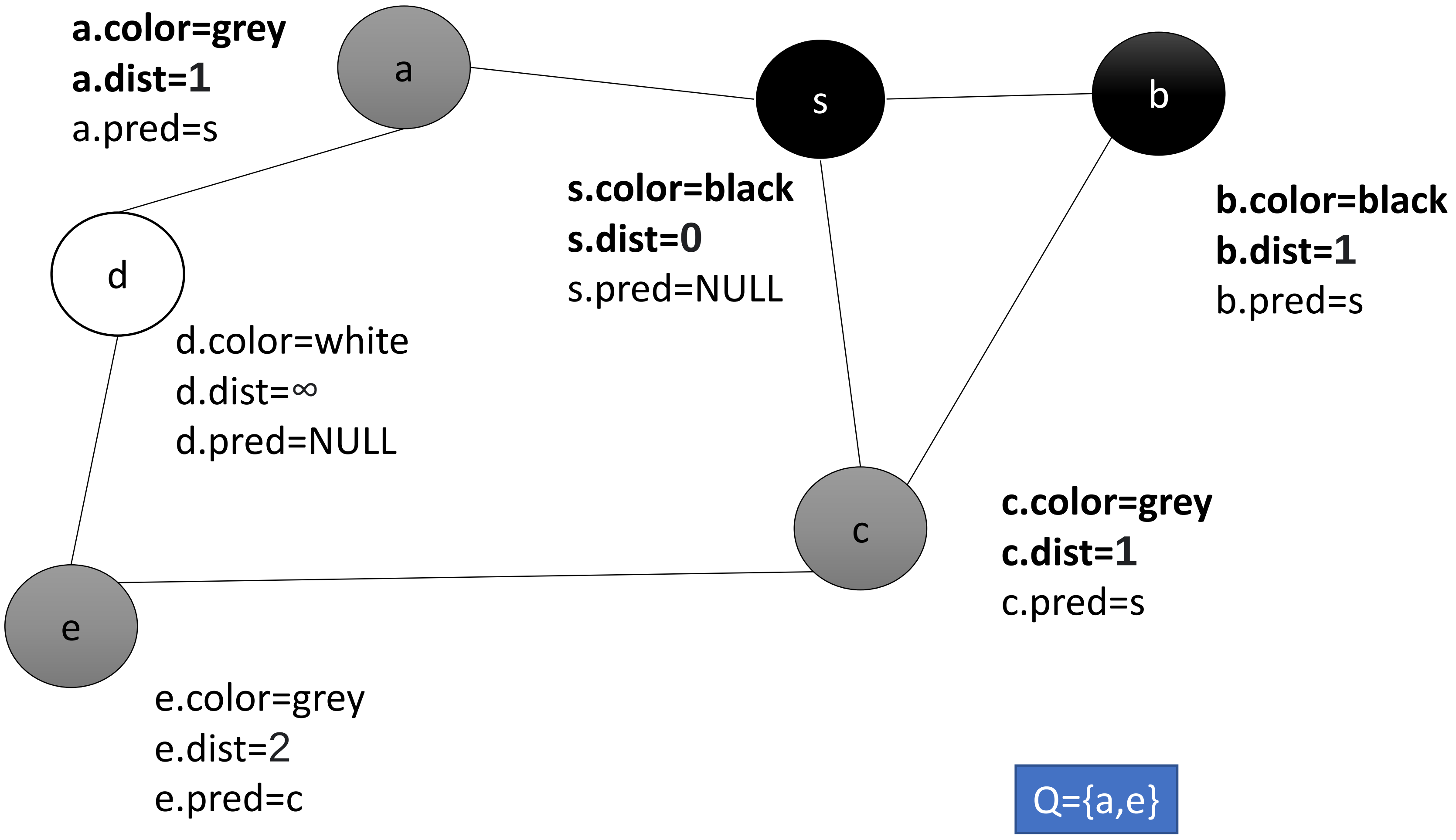
BFS Example



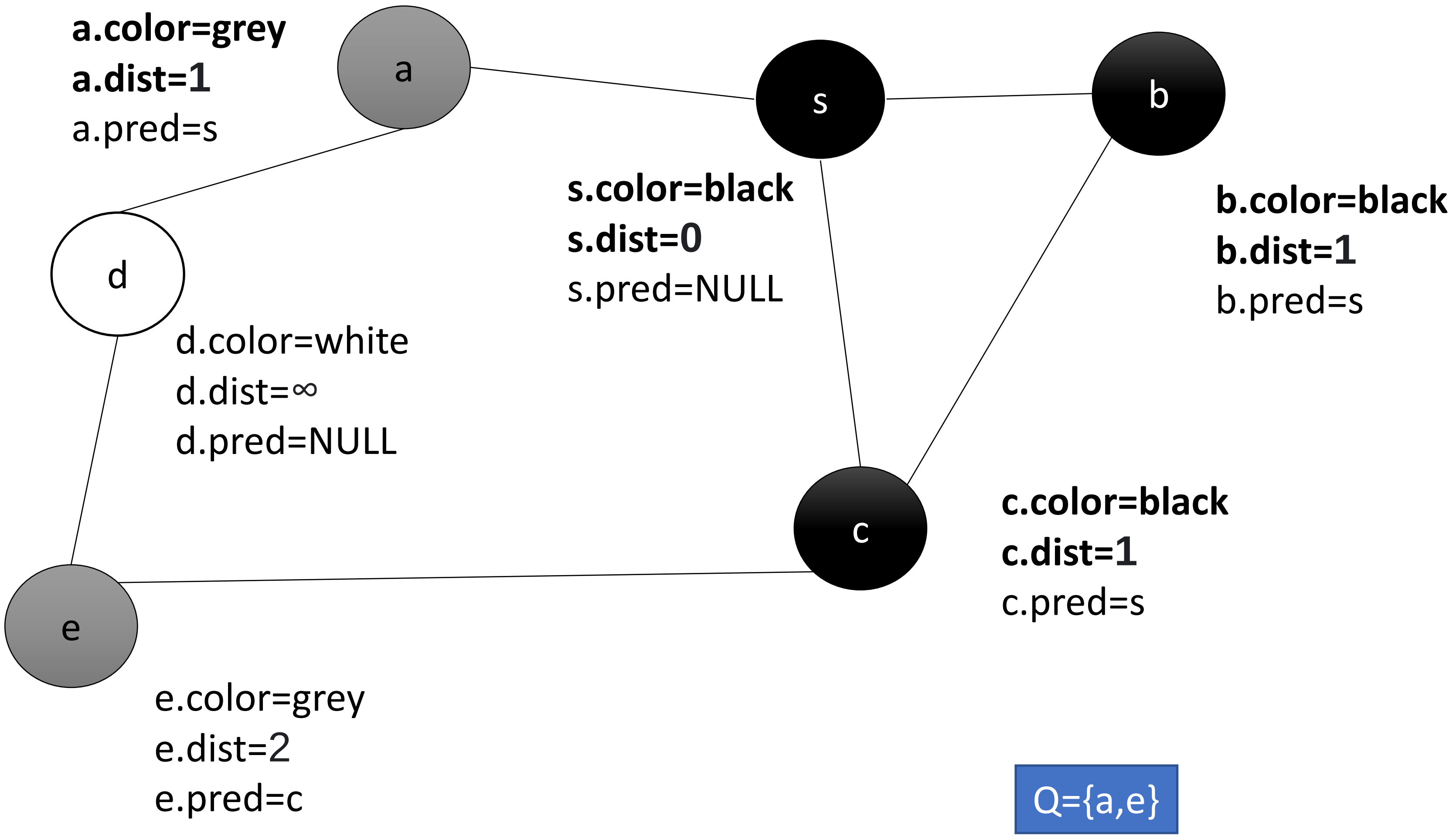
Q={a}

c dequeued

BFS Example

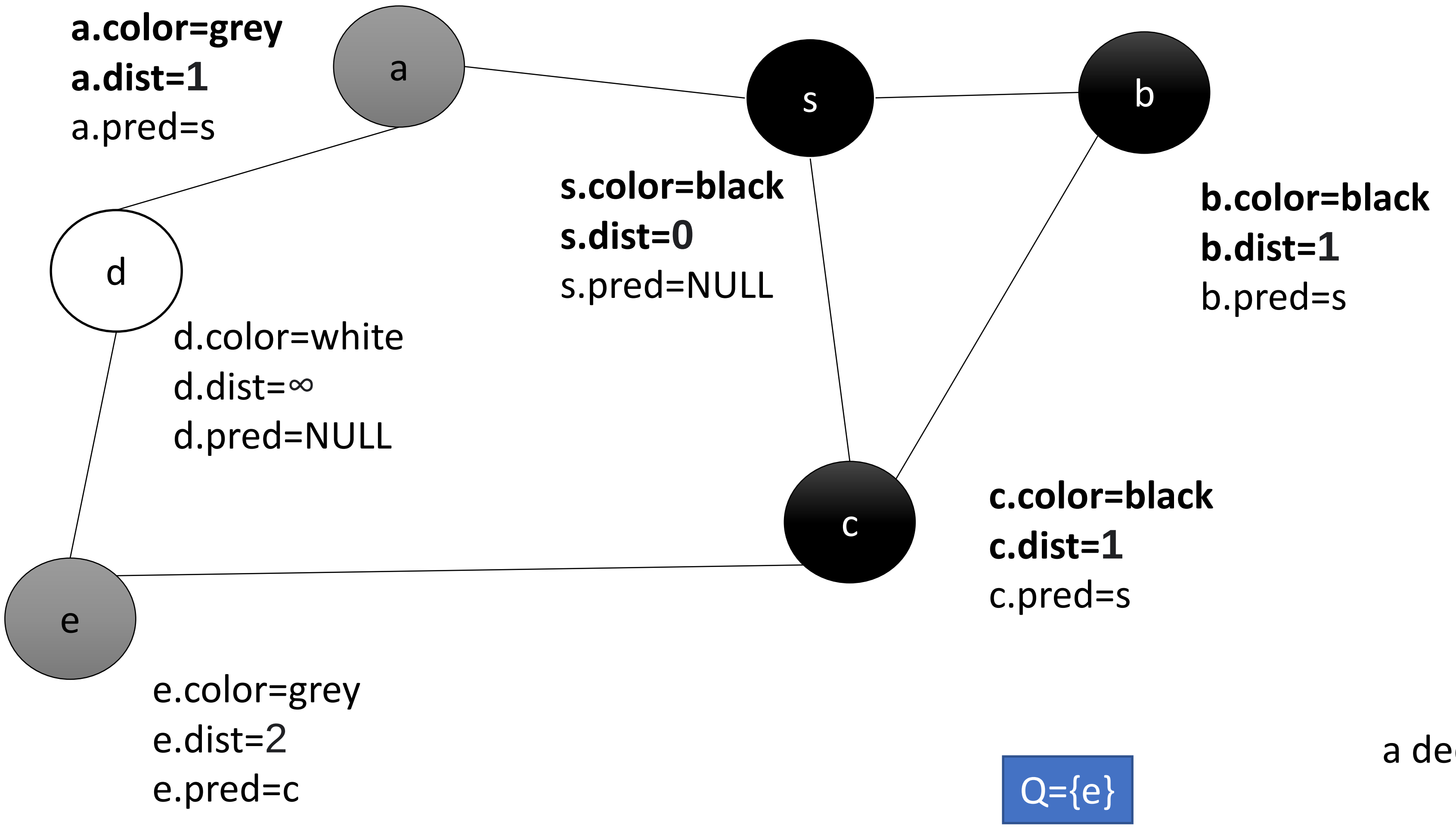

a.color=grey
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=grey
c.dist=1
c.pred=s
e
e.color=grey
e.dist=2
e.pred=c
Q={a,e}

BFS Example



a.color=grey
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=white
d.dist=∞
d.pred=NULL
c
c.color=black
c.dist=1
c.pred=s
e
e.color=grey
e.dist=2
e.pred=c
Q={a,e}

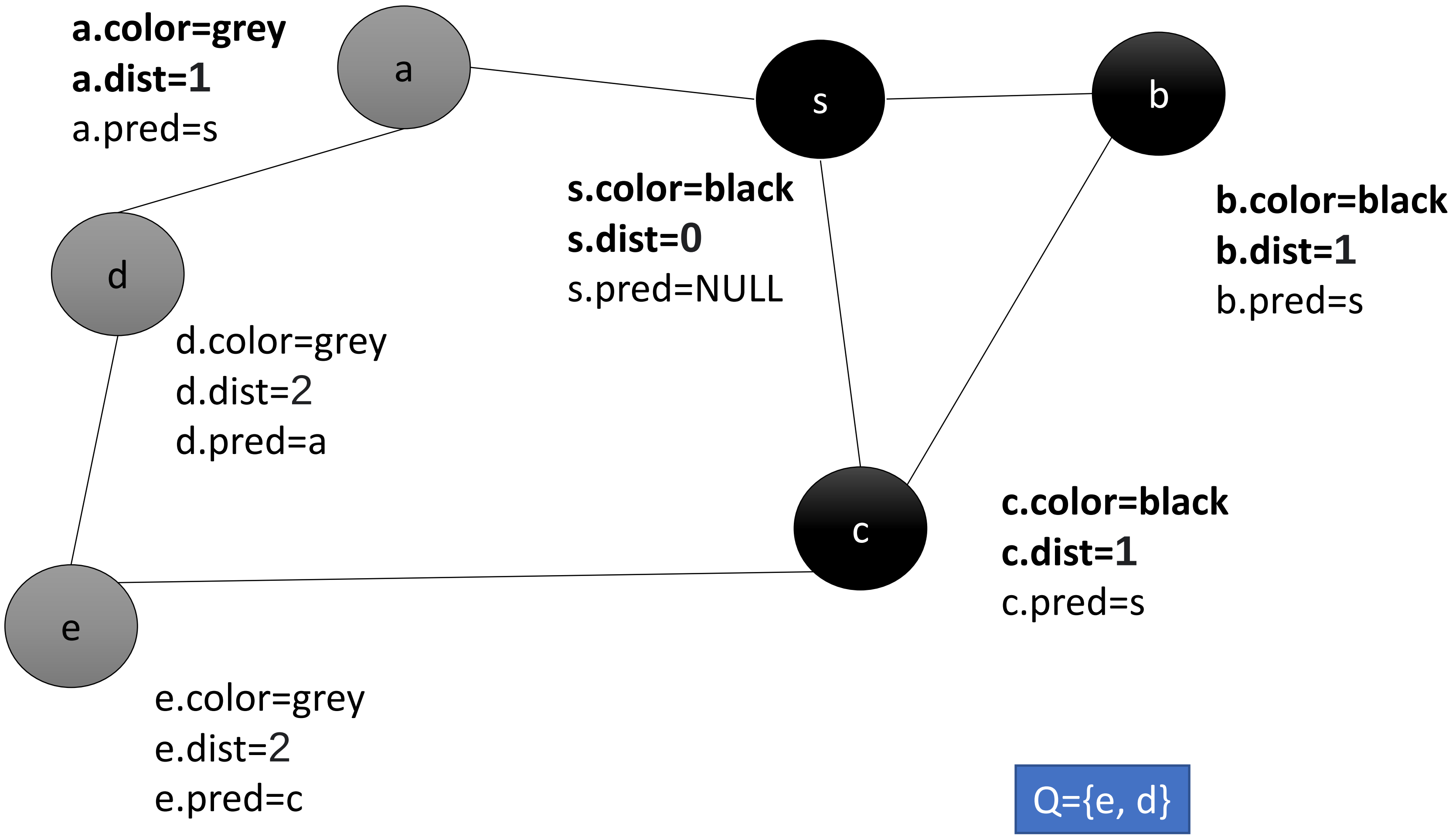
BFS Example



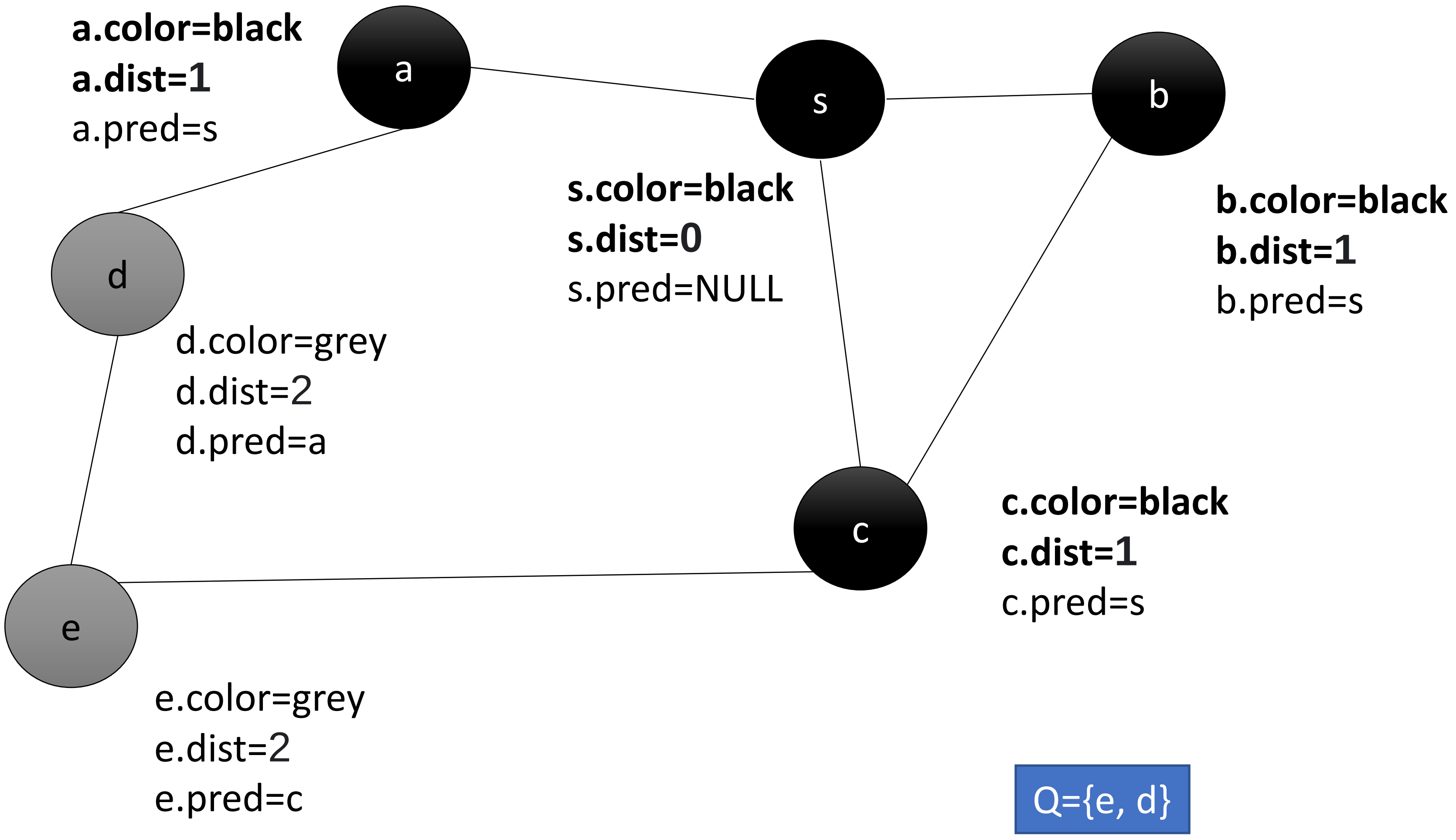
Q={e}

a dequeued

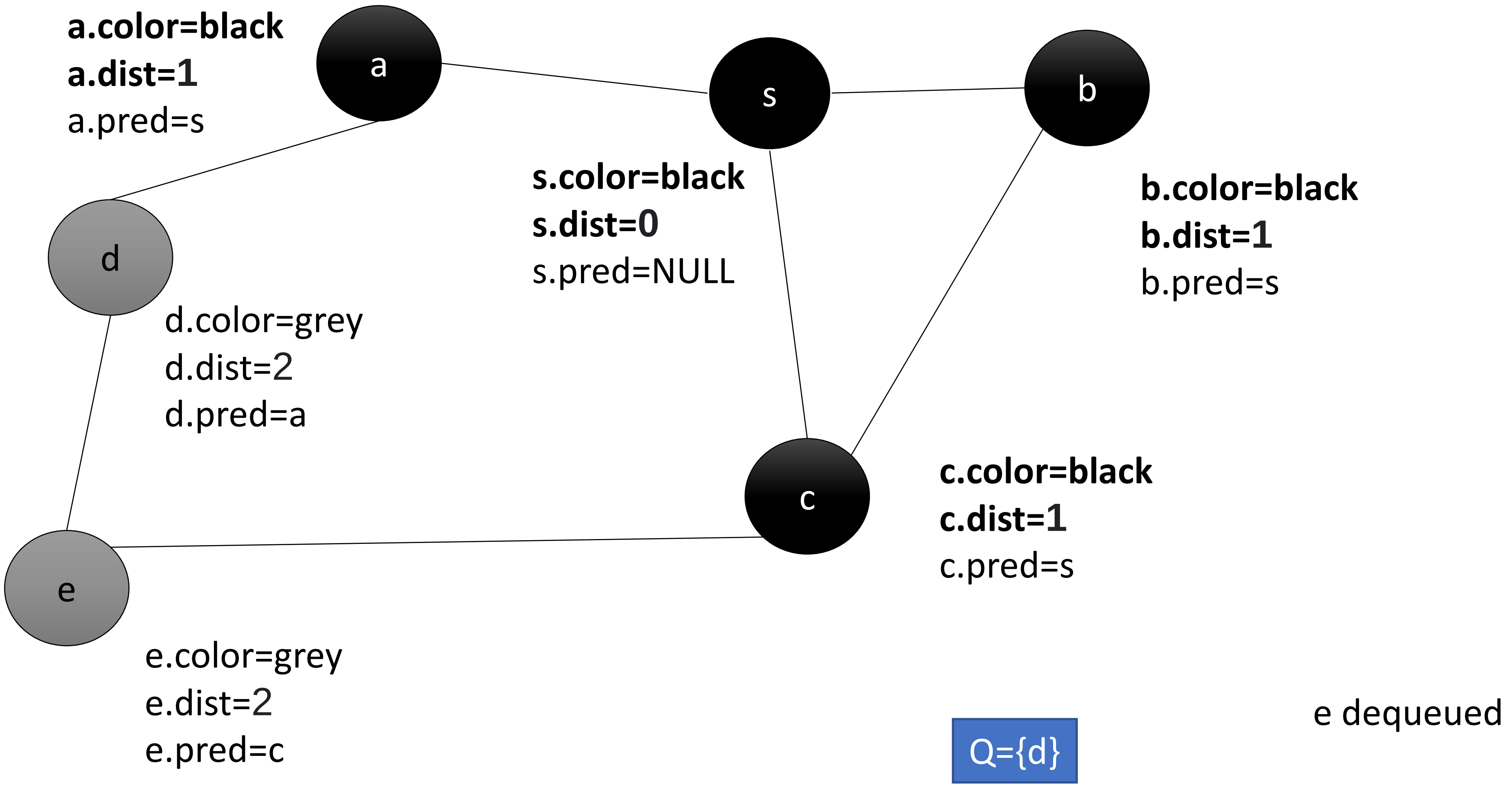
BFS Example


a.color=grey
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=grey
d.dist=2
d.pred=a
c
c.color=black
c.dist=1
c.pred=s
e
e.color=grey
e.dist=2
e.pred=c
Q={e, d}

BFS Example

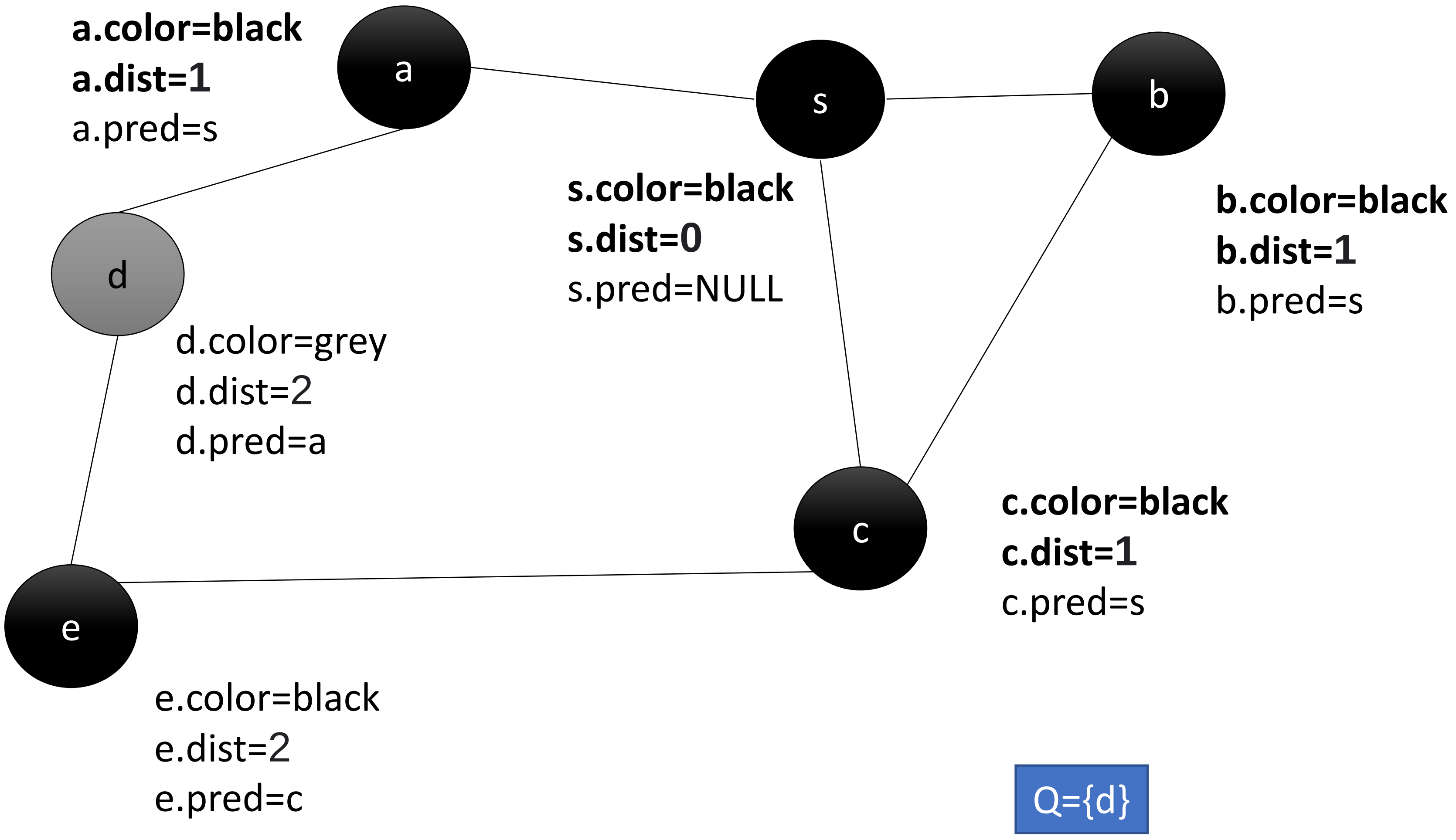

a
s
b
d
c
e
a.color=black
a.dist=1
a.pred=s
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d.color=grey
d.dist=2
d.pred=a
c.color=black
c.dist=1
c.pred=s
e.color=grey
e.dist=2
e.pred=c
Q={e, d}

BFS Example

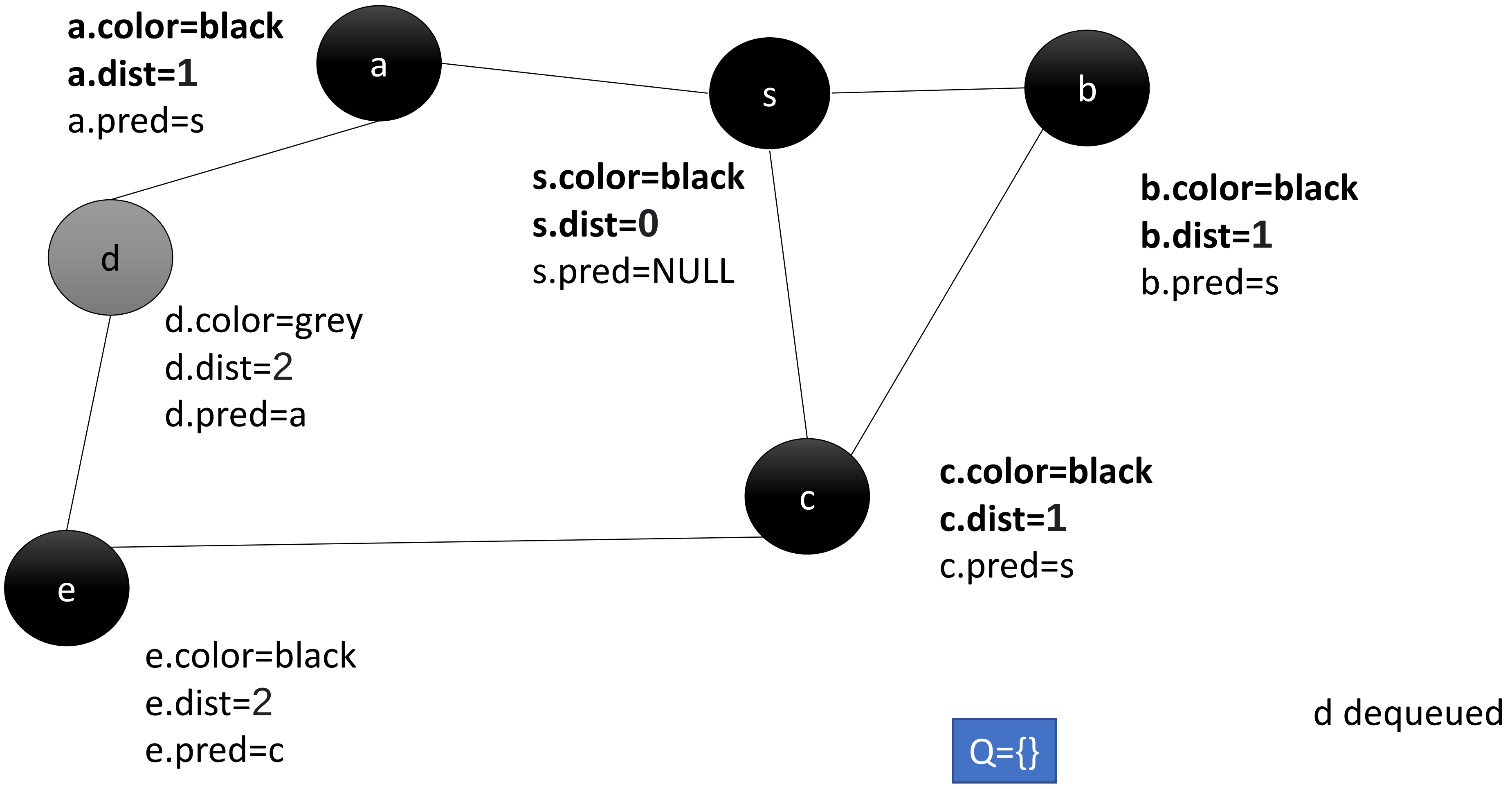


a.color=black
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=grey
d.dist=2
d.pred=a
c
c.color=black
c.dist=1
c.pred=s
e
e.color=grey
e.dist=2
e.pred=c
Q={d}
e dequeued

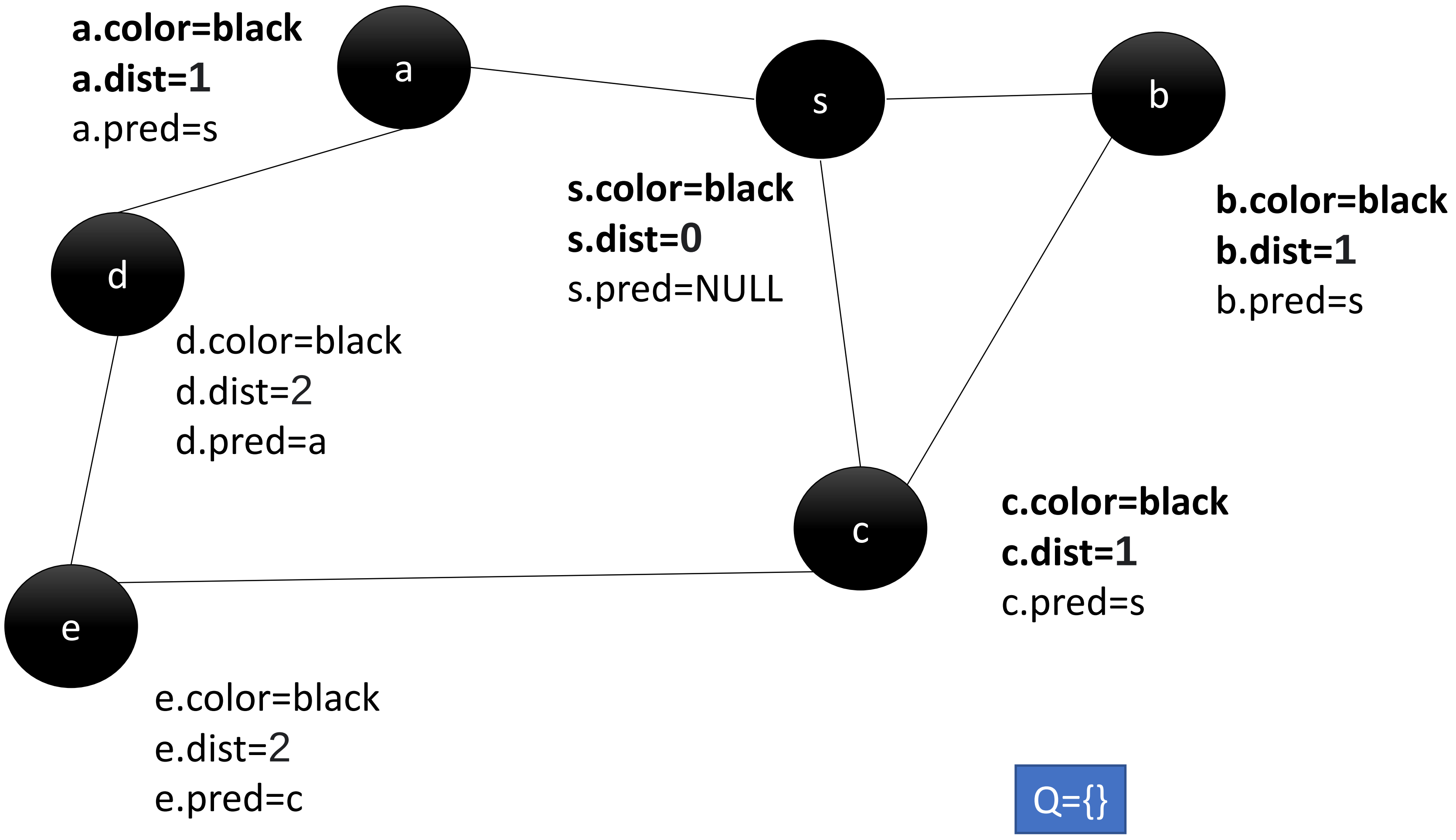
BFS Example


a.color=black
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=grey
d.dist=2
d.pred=a
c
c.color=black
c.dist=1
c.pred=s
e
e.color=black
e.dist=2
e.pred=c
Q={d}

BFS Example


a
s
b
d
c
e
a.color=black
a.dist=1
a.pred=s
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d.color=grey
d.dist=2
d.pred=a
c.color=black
c.dist=1
c.pred=s
e.color=black
e.dist=2
e.pred=c
Q={}
d dequeued

BFS Example



a.color=black
a.dist=1
a.pred=s
a
s
b
s.color=black
s.dist=0
s.pred=NULL
b.color=black
b.dist=1
b.pred=s
d
d.color=black
d.dist=2
d.pred=a
c
c.color=black
c.dist=1
c.pred=s
e
e.color=black
e.dist=2
e.pred=c
Q={}

BFS Running time: Exercise

- What is the running time of BFS given a graph represented by an adjacency list?

Applications of BFS

- Find a ***shortest path***,where length is measured in number of edges.

- Find ***connected components*** of an undirected graph

- ***Garbage collection***: traverse the graph of objects reachable from the stack in BFS manner.
  - Enables garbage collector to remove objects that are not reachable.

Depth-first search (DFS)

- Uses notion of *traversal time- starttime and endtime recorded for each vertex.*
- Start at vertex *s*, mark *s as* "visited (discovered)", and label *s* as current vertex called *u*
- Travel along an ***arbitrary*** edge (*u*,*v*).
  - If edge (*u*,*v*) leads to an already visited vertex *v,* return to *u*
  - If vertex *v* is unvisited, move to *v*, paint *v* "visited", set *v* as our current vertex, and repeat the previous steps

Depth-first search (DFS)

- Eventually, you get to a point where all edges from *u* lead to visited vertices
- Backtrack until we get back to a previously visited vertex *v*
- *v* becomes our current vertex, and we repeat the previous steps
- Uses a ***stack***.

DFS

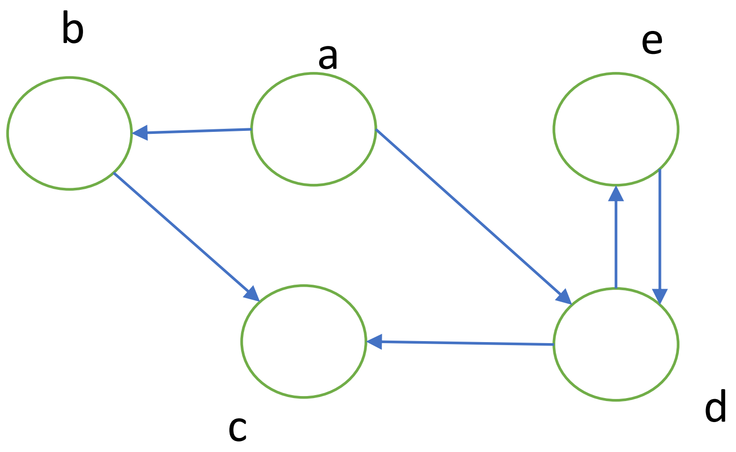
- *Uses backtracking*: advance if possible, backtrack if not possible to advance further.
- *Organizes* vertices by *start/end* times.
- Classifies edges as either ***tree*** or ***back edges***.

DFS algorithm

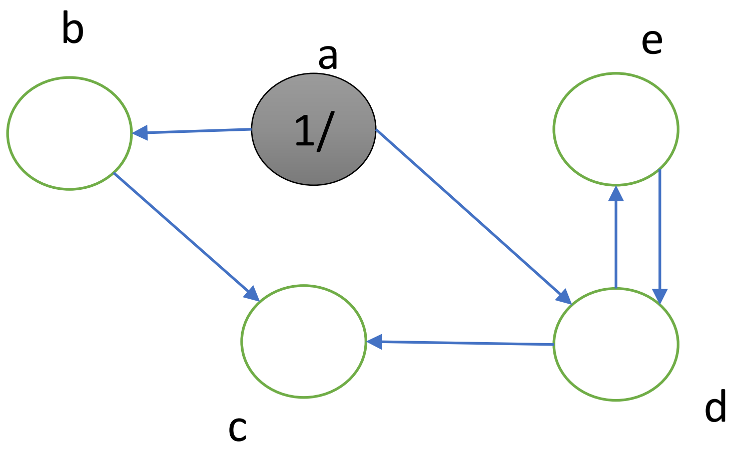
```
DFS(G):
##Initialize all vertices
For u ∈ G.V Do #undiscovered
        u.color=white
        u.pred=NULL
time=0 #reset time counter
##Visit all vertices
For u ∈ G.V Do
        If u.color=white Then
                DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u):
##Initialize DFS
u.color=gray #discovered
time=time+1
u.startttime=time #discovery time
##Visit all children-recursively
For v ∈ u.adj Do
        If v.color=white Then
                v.pred=u
                DFS-VISIT(G,v)
u.color=black #finished/processed
time=time+1
u.endtime=time #finishing time
```

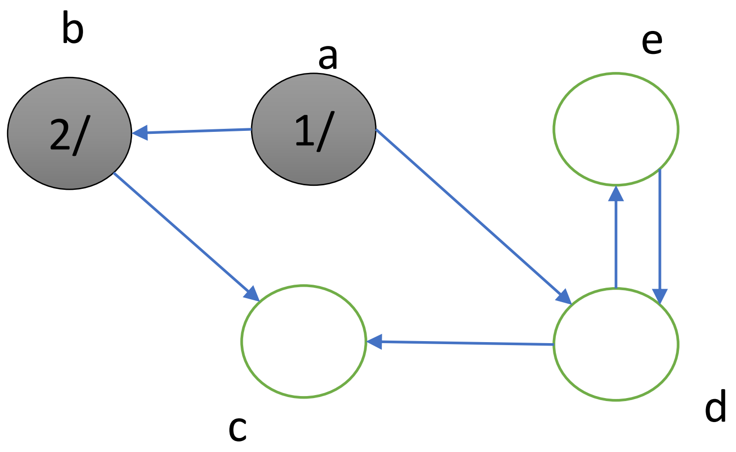
DFS Example



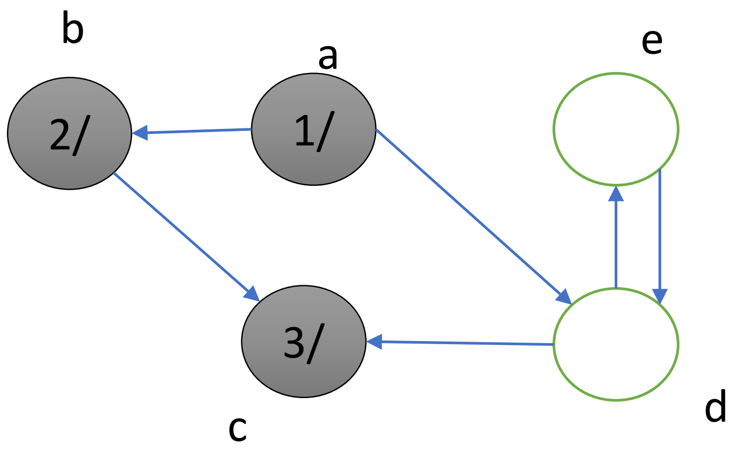
DFS Example



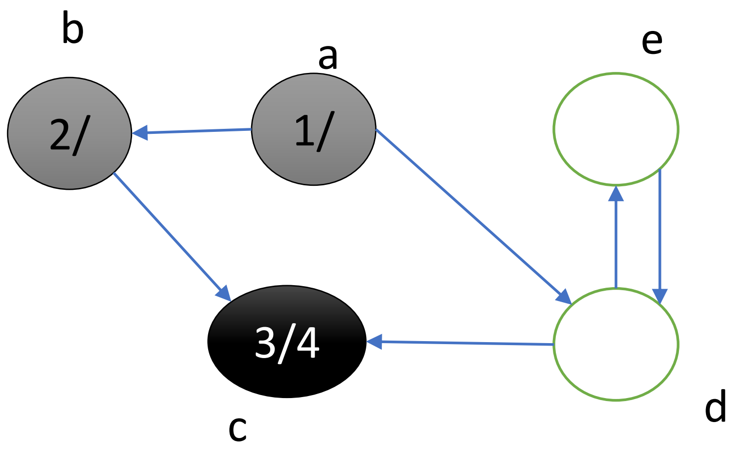
DFS Example



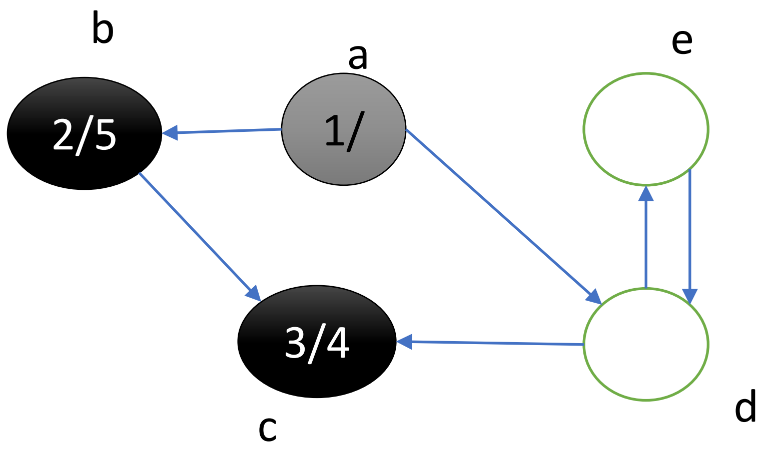
DFS Example



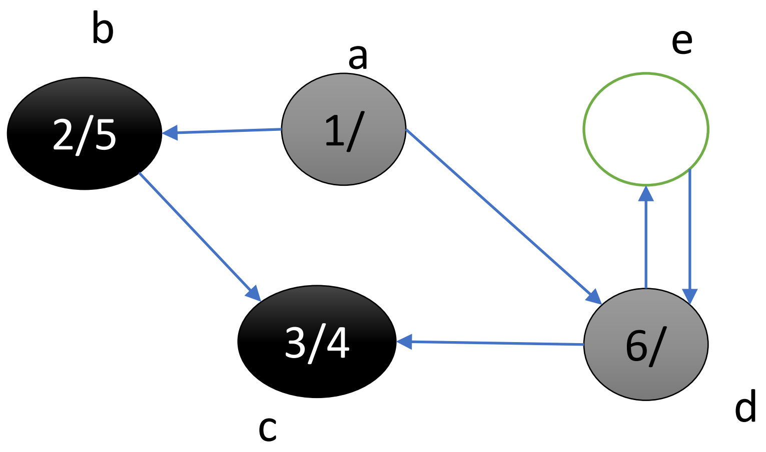
DFS Example



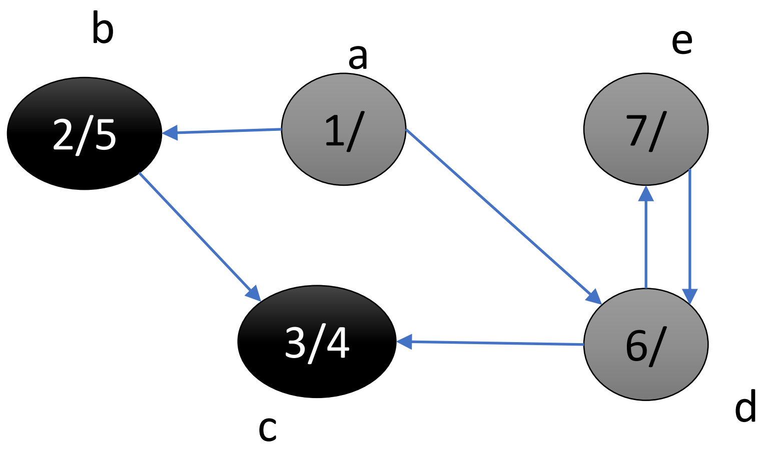
DFS Example



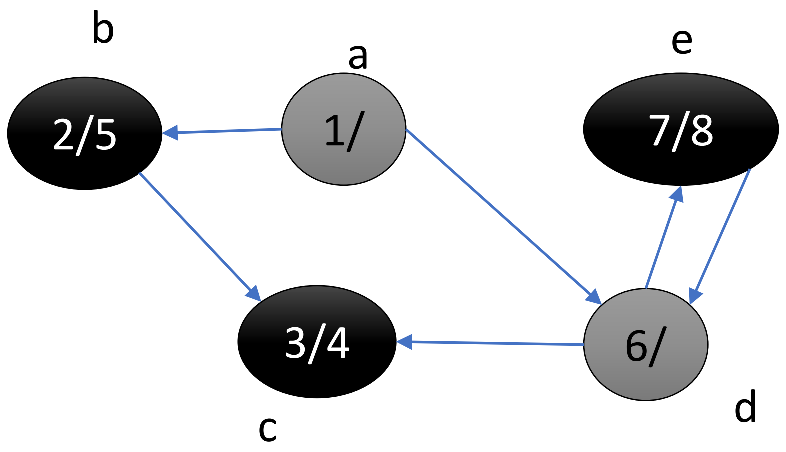
DFS Example



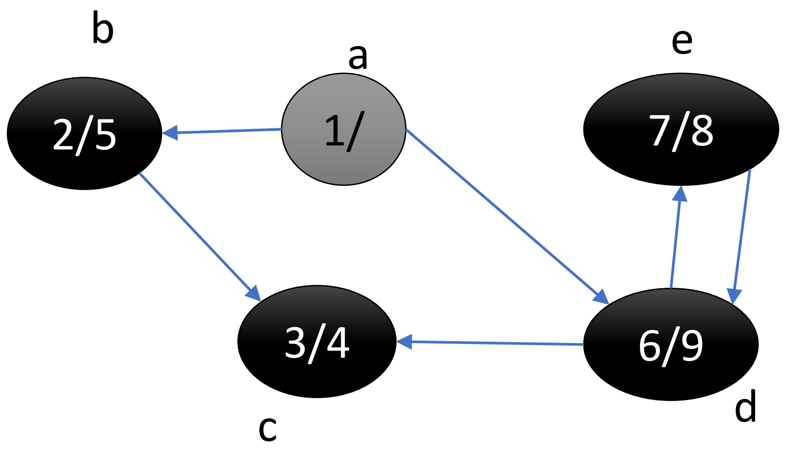
DFS Example



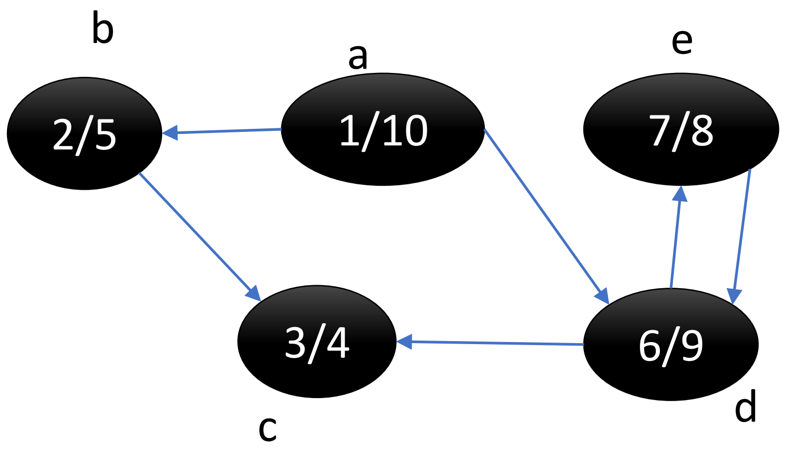
DFS Example



DFS Example



DFS Example

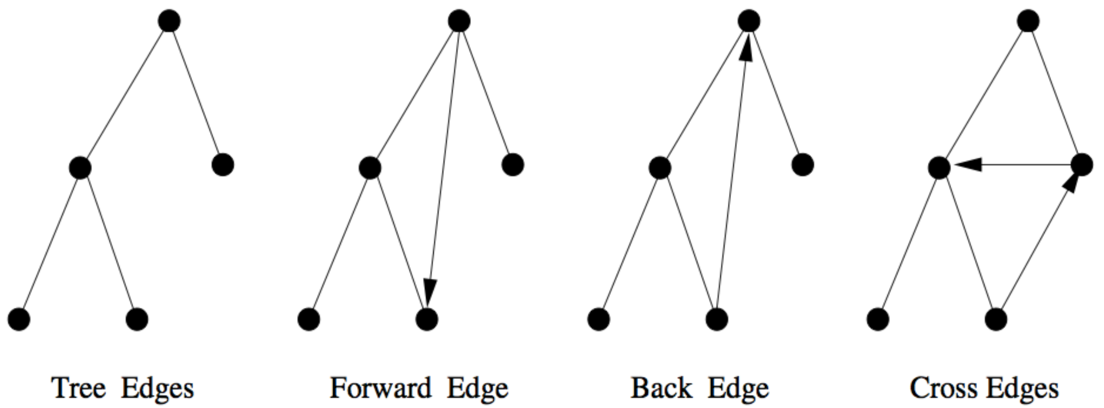


DFS: edge classification

- Traversing an undirected graph in DFS reveals only ***tree and back edges***.
- Traversing a directed graph reveals ***tree, back, forward, and cross*** edges.

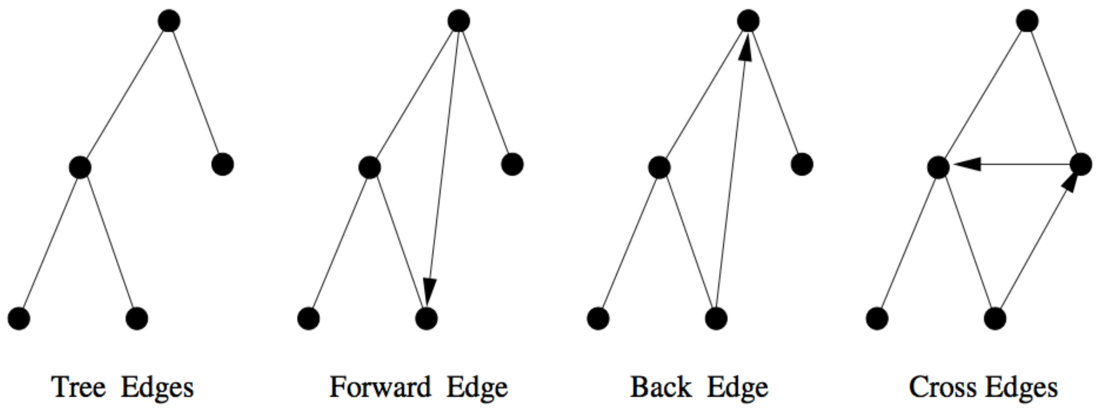
DFS: edge classification

- Tree edge: gray to white
  - Edges in depth first forest.
  - (u,v) is a tree edge if v is visited for the first time from u.
  - Form a tree containing each vertex visited in G.
- Back edge: gray to gray
  - Non-tree edge from descendant to ancestor.
  - (u,v) is a back edge if v appears before u based on start-time and there is a path from v to u.



DFS: edge classification

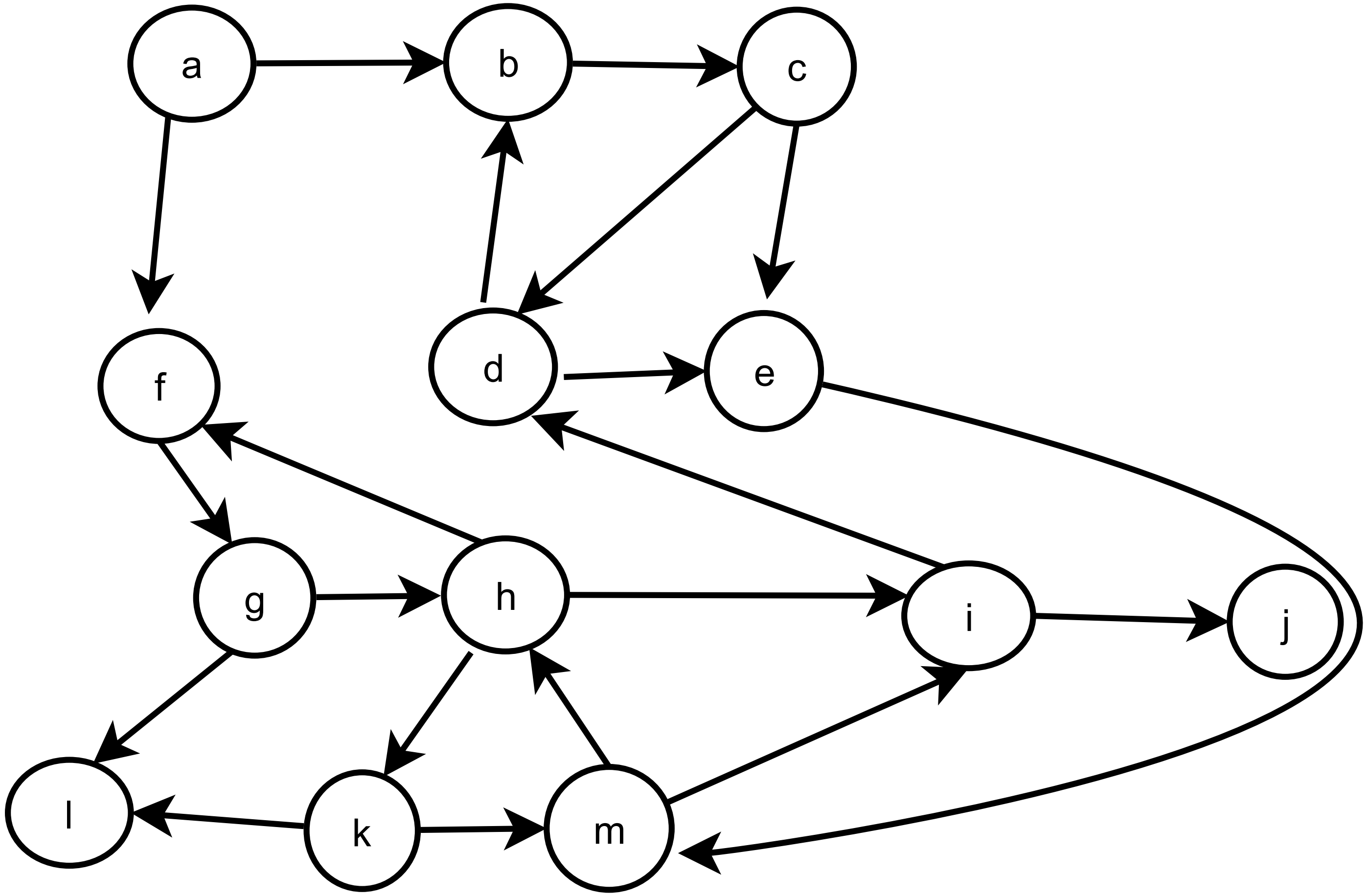
- Forward edge: gray to black
  - Non-tree edge from ancestor to descendant.
  - (u,v) is a forward edge if v appears after u based on start-time and there is a path from u to v.
- Cross edge: gray to black
  - Between trees or subtrees.
  - (u,v) is a cross edge if v is neither an ancestor nor a descendant of u.



Most algorithms use ***tree and back edges***.

Examples? Later.

Exercise: Edge Classification



Identify each of the following, if any:

a) Tree edges
b) Back edges
c) Forward edges
d) Cross edges

DFS Applications

- Find connected components (in an undirected graph)
- Check digraph for cycles.
  - A graph, G, has a cycle it has at least one back edge.
- Graph mining and social network analysis e.g. relationship discovery, categorization, recommendation(e.g. people you may know, videos you may like), etc.
  - Example: [link]
  - Chakrabarti, D. (2011). Graph Mining. In: Sammut, C., Webb, G.I. (eds) Encyclopedia of Machine Learning. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-30164-8_350
- Topological sorting: Find a linear order that refines a given partial order (e.g. organizing courses by prerequisites, grooming and dressing up, implementing a recipe)

BFS vs. DFS

- BFS
  - BFS visits all vertices that are reachable from the start node s and ***returns one search tree***.
- DFS
  - Visits all vertices in the graph and ***may return multiple search trees***.
  - DFS classifies edges as *tree, forward, back, and cross* edges.

Summary

- Definitions: graphs, vertices, edges, cycles, subgraphs, etc.
- Two graph representations: ***adjacency list and adjacency matrix***
- Graph traversal: ***BFS and DFS***
- Applications of graph search algorithms

Next

- DAGs and topological sorting,
- Min-Spanning tree: Prims/Kruskals,
- Shortest Path algorithms: Dijkstras, Floyds