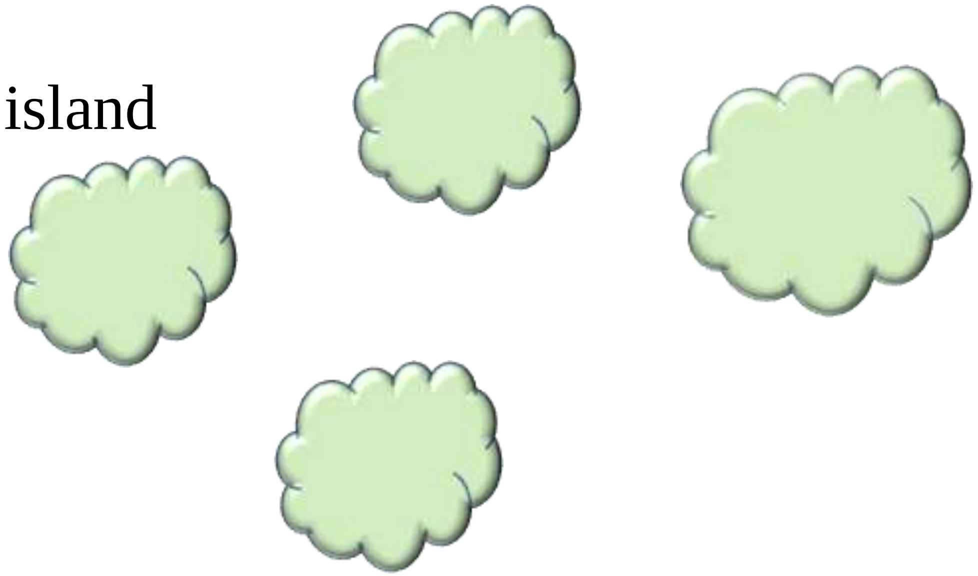
Graphs

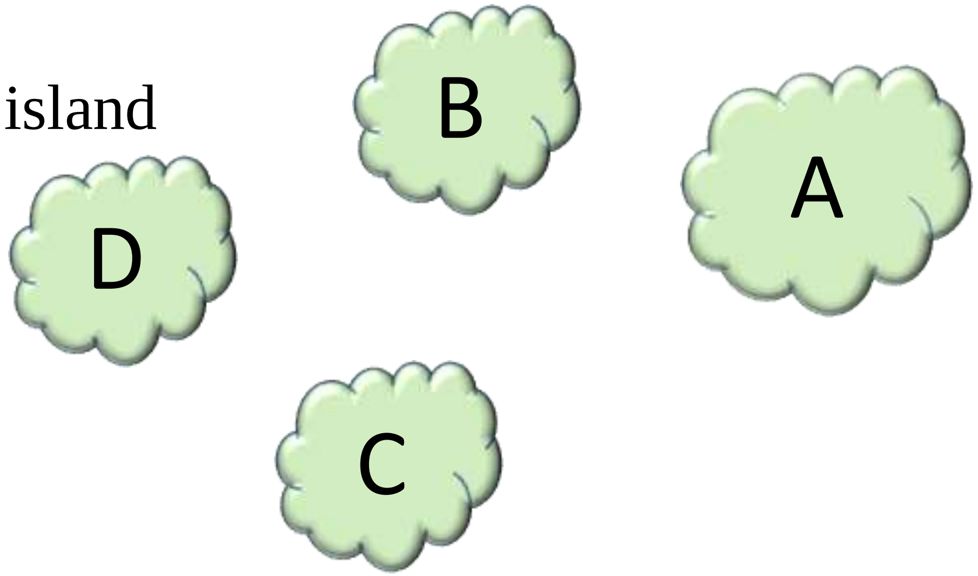
黃世強 (Sai-Keung Wong)

College of Computer Science
National Yang Ming Chiao Tung University
Taiwan

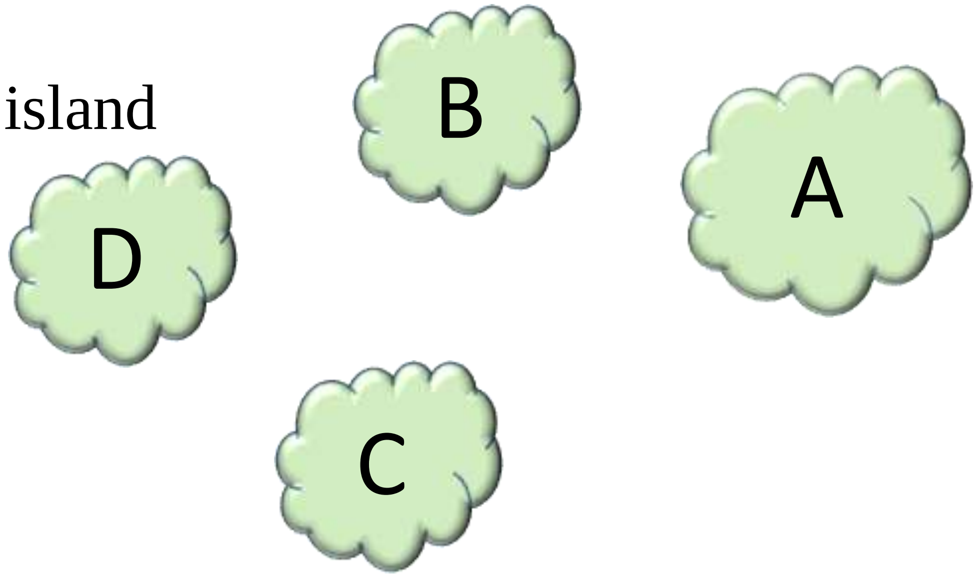
Graphs



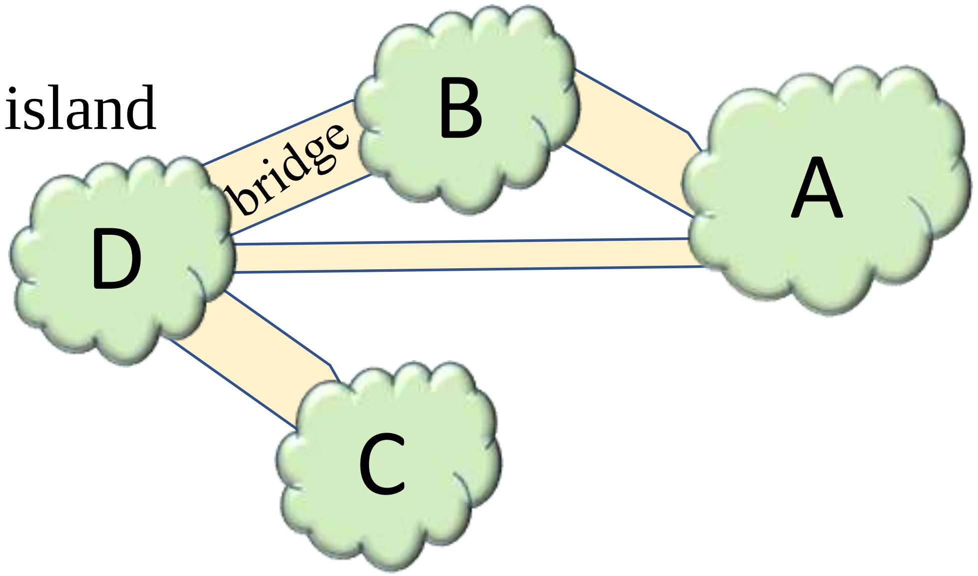
Graphs



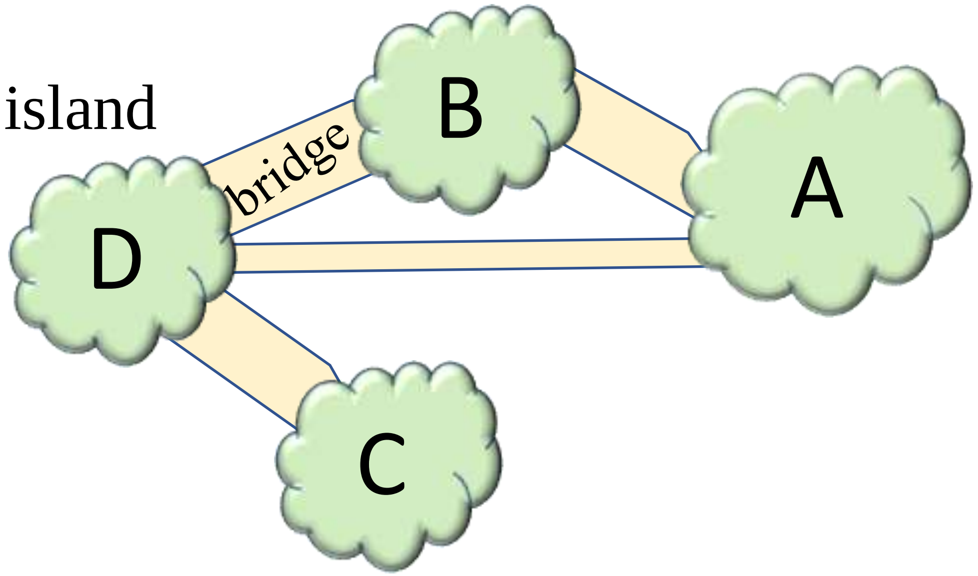
Graphs



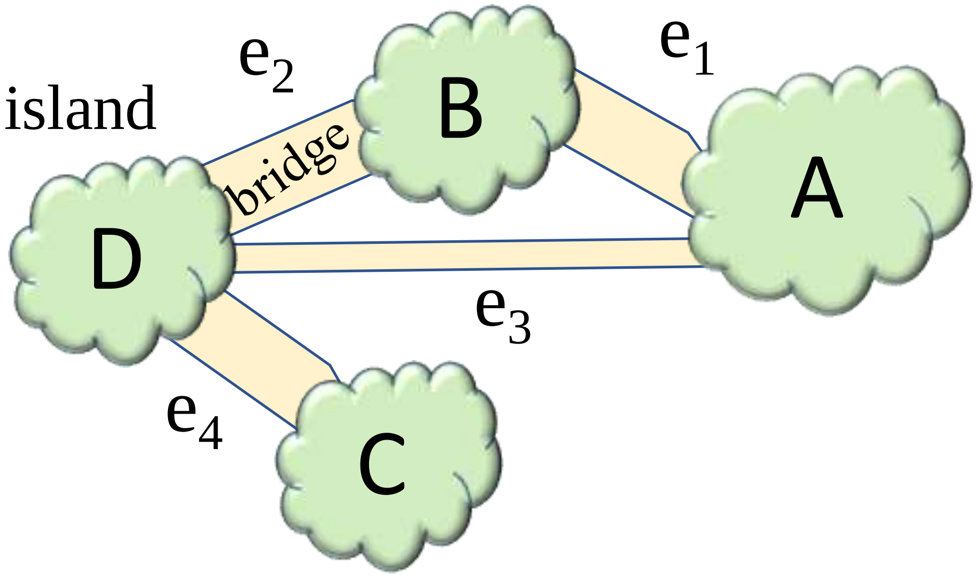
Graphs



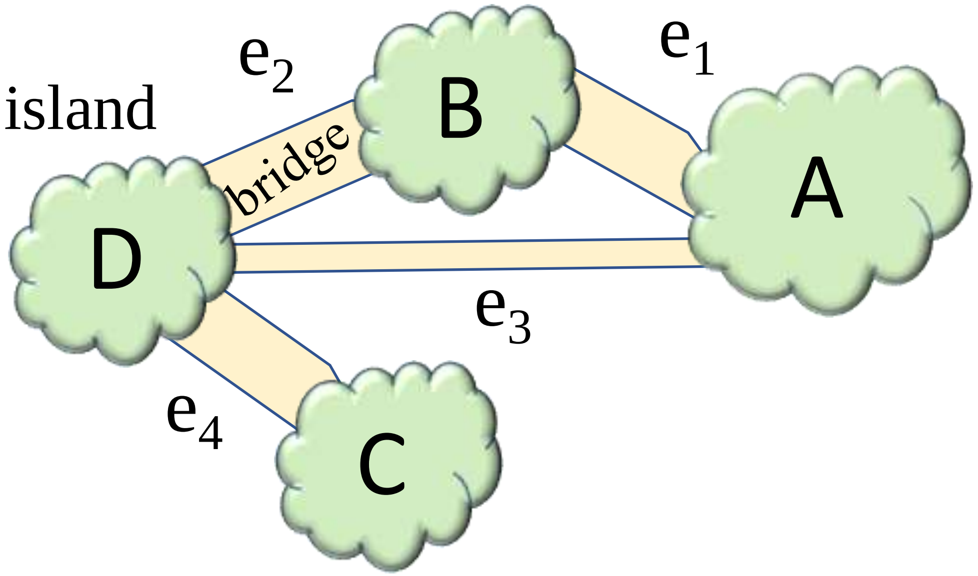
Graphs



Graphs

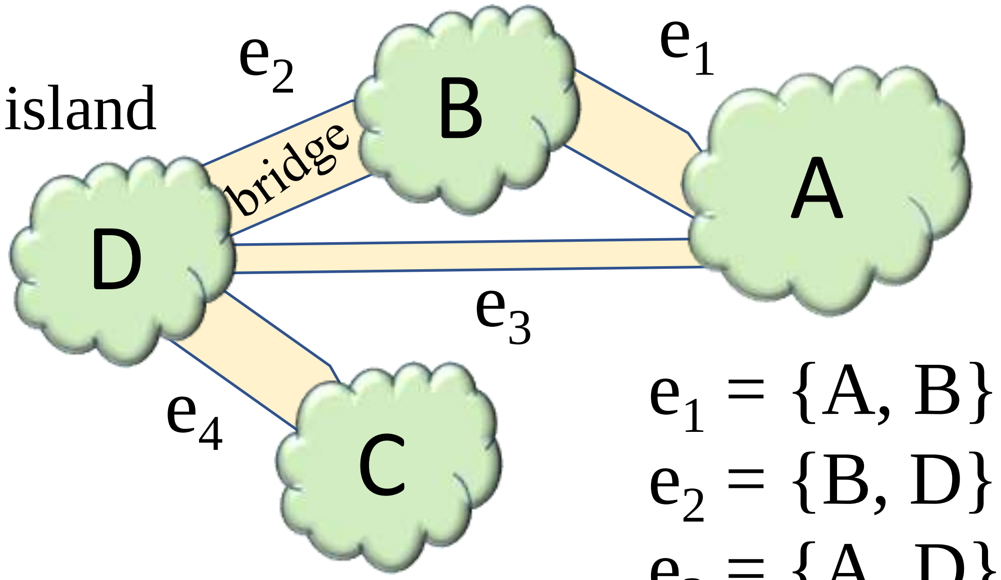


Graphs



An edge e connects two nodes.

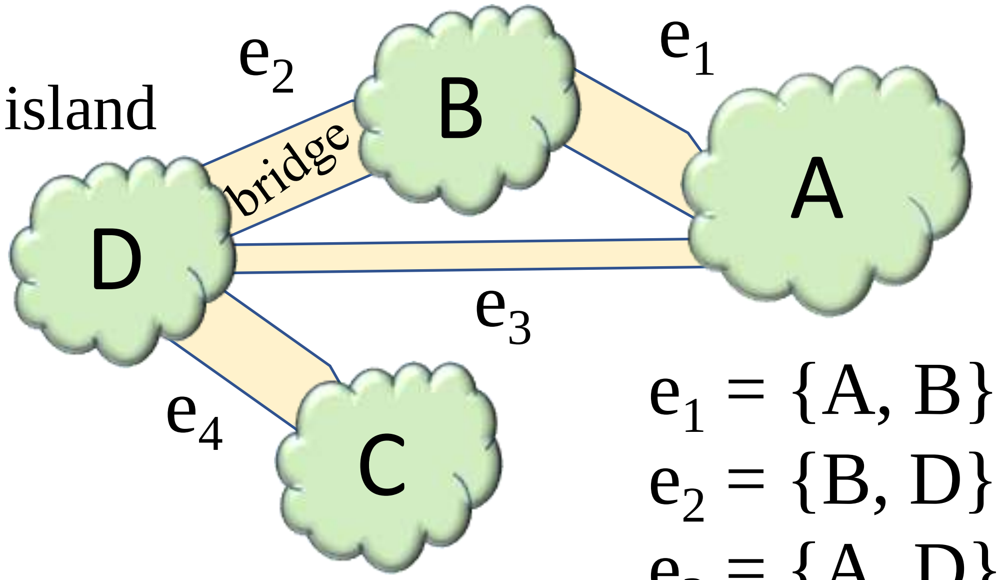
Graphs



$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.

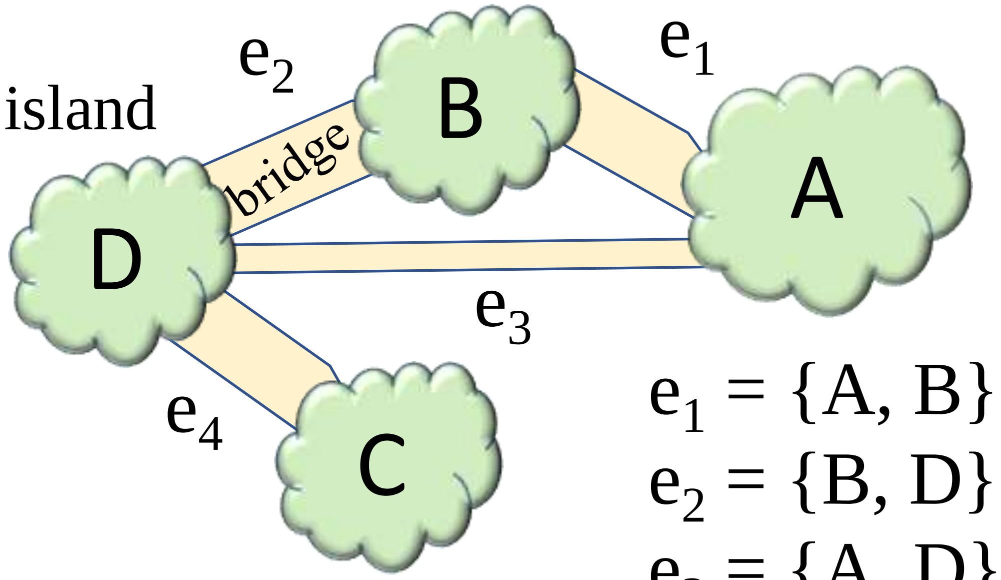
Graphs



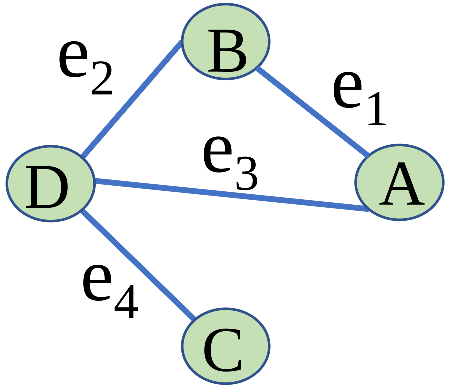
$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.

Graphs



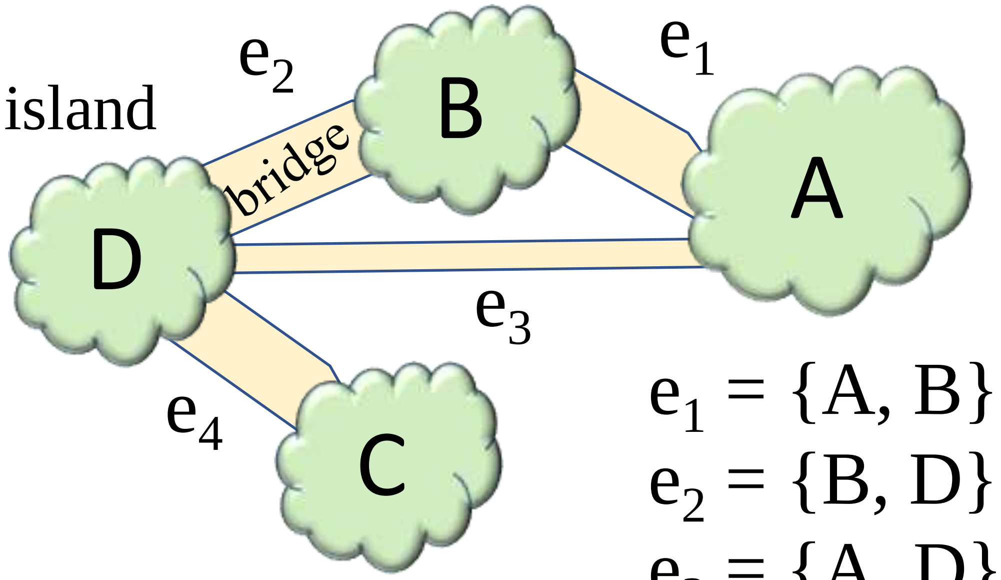
$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$



Spatial relation

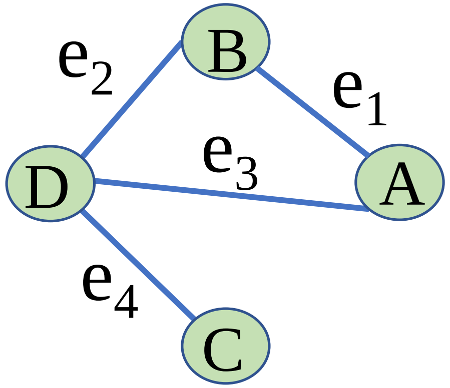
An edge e connects two nodes.

Graphs

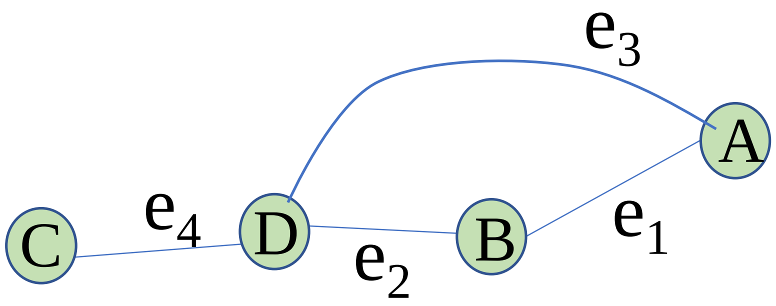


$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.



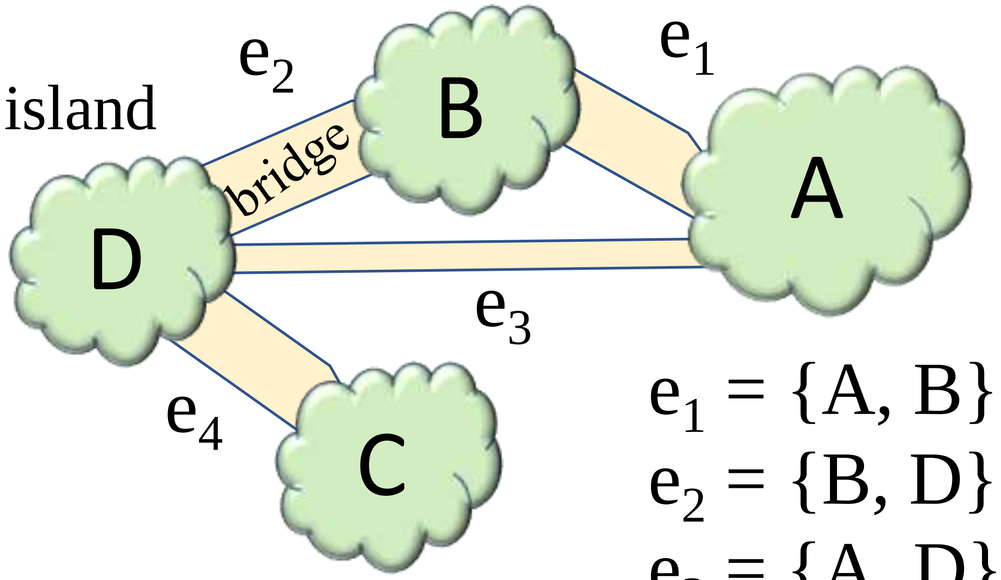
Spatial relation



Connectivity information

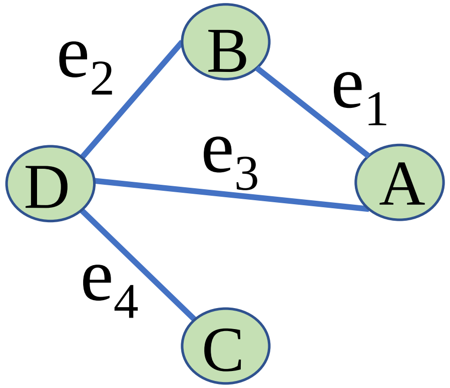
Graphs

- We can use a graph to represent an abstract structure of entities in real life.
- For example, there are bridges connecting islands. We can use **edges** to represent **bridges** and use **nodes** to represent **islands**.

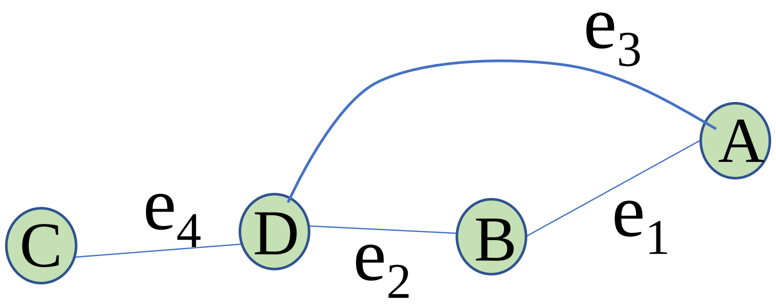


$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.



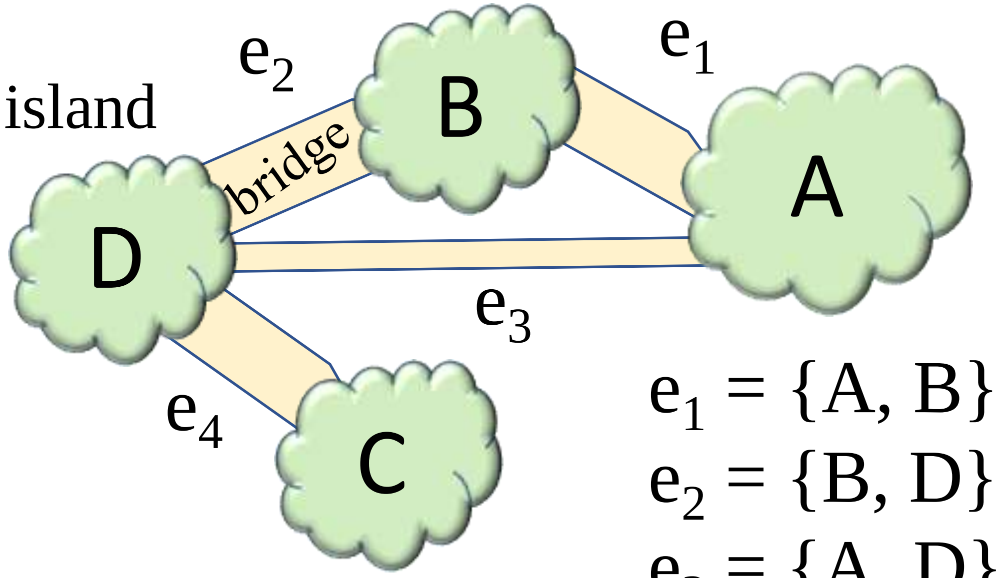
Spatial relation



Connectivity information

Graphs

- We can use a graph to represent an abstract structure of entities in real life.
- For example, there are bridges connecting islands. We can use edges to represent bridges and use nodes to represent islands.



$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.

A set of nodes

$V = \{ A, B, C, D\}$

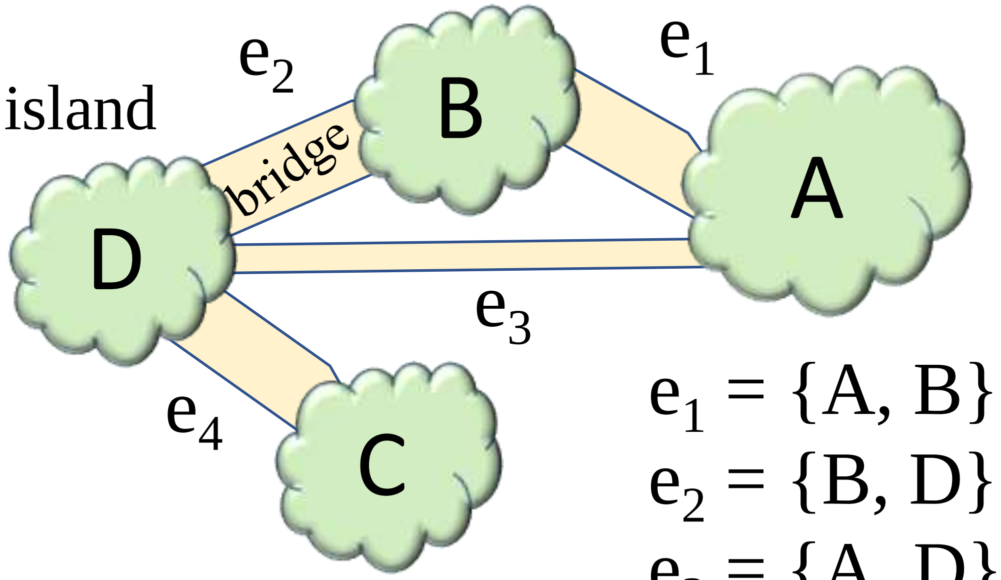
A set of edges

$E = \{e_1, e_2, e_3, e_4 \}$

Graph $G = (V, E)$

Graphs

- We can use a graph to represent an abstract structure of entities in real life.
- For example, there are bridges connecting islands. We can use edges to represent bridges and use nodes to represent islands.



$e_1 = \{A, B\}$
$e_2 = \{B, D\}$
$e_3 = \{A, D\}$
$e_4 = \{D, C\}$

An edge e connects two nodes.

A set of nodes

$V = \{ A, B, C, D\}$

A set of edges

$E = \{e_1, e_2, e_3, e_4 \}$

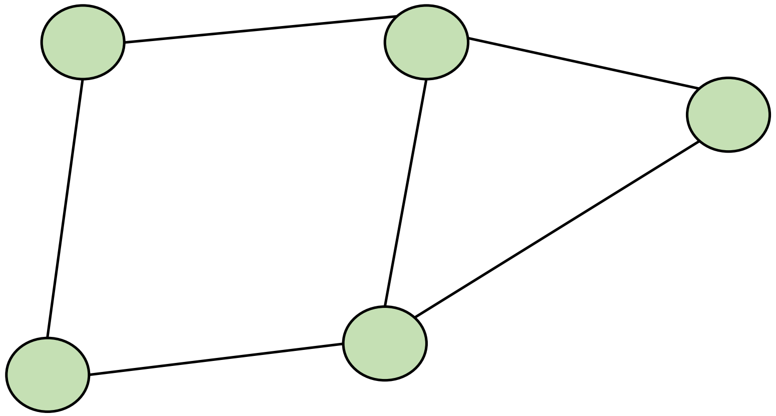
Graph $G = (V, E)$

Basic concepts about graphs

- Directed vs. undirected graphs
- Weighted vs. unweighted graphs
- Adjacent vertices
- Incident
- Degree
- Neighbor
- Loops
- Parallel edge
- Simple graph
- Complete graph: every two nodes are connected by an edge
- Spanning tree

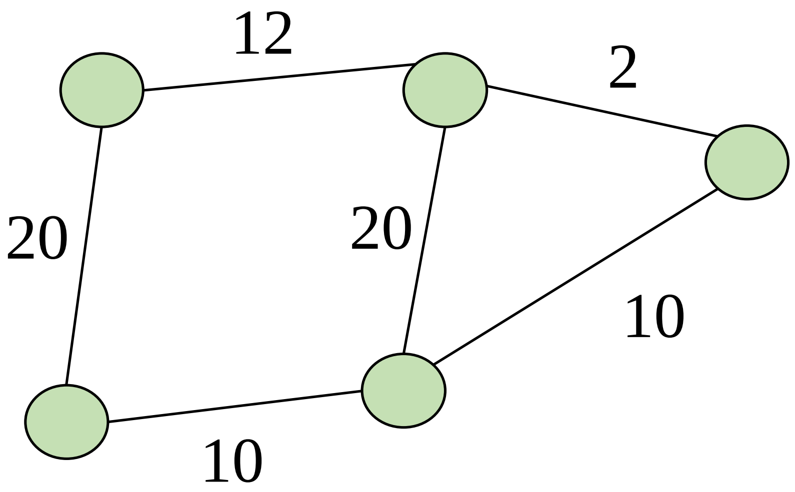
Basic concepts about graphs

- Weight: A vertex or an edge can be assigned a value, called weight.
- Incident: an edge is incident to a vertex if it connects the vertex
- Degree (or valence) of a vertex: the number of edges attached at it
- Neighbor of a vertex: vertices can be accessed by the edges incident to the vertex.
- Loop: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.



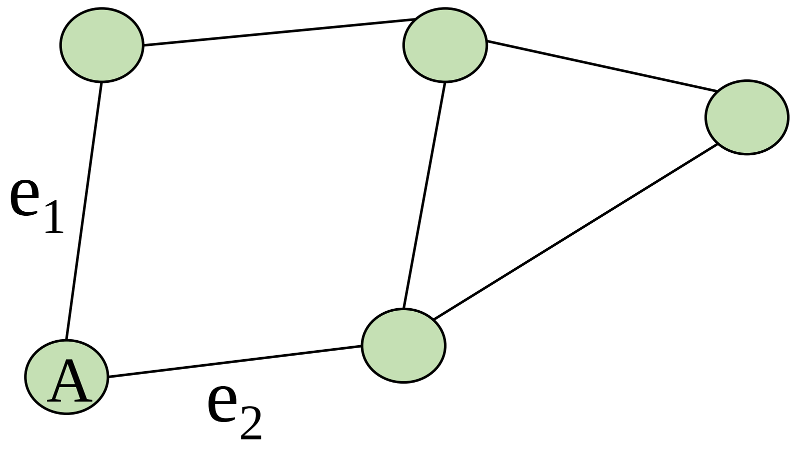
Basic concepts about graphs

- **Weight**: A vertex or an edge can be assigned a value, called weight.
- Incident: an edge is incident to a vertex if it connects the vertex
- Degree (or valence) of a vertex: the number of edges attached at it
- Neighbor of a vertex: vertices can be accessed by the edges incident to the vertex.
- Loop: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.



Basic concepts about graphs

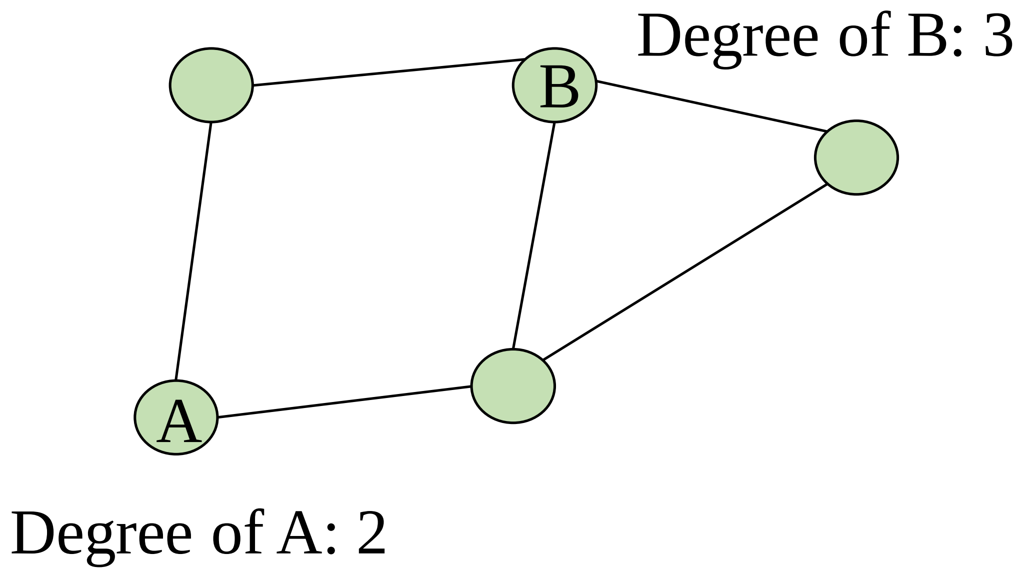
- Weight: A vertex or an edge can be assigned a value, called weight.
- **Incident**: an edge is incident to a vertex if it connects the vertex
- Degree (or valence) of a vertex: the number of edges attached at it
- Neighbor of a vertex: vertices can be accessed by the edges incident to the vertex.
- Loop: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.



e_1 and e_2 are incident to A

Basic concepts about graphs

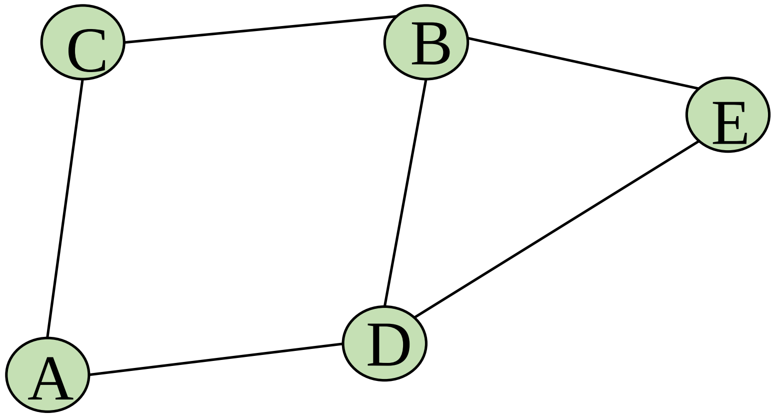
- Weight: A vertex or an edge can be assigned a value, called weight.
- Incident: an edge is incident to a vertex if it connects the vertex.
- **Degree (or valence) of a vertex**: the number of edges attached at it
- Neighbor of a vertex: vertices can be accessed by the edges incident to the vertex.
- Loop: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.



Basic concepts about graphs

- Weight: A vertex or an edge can be assigned a value, called weight.
- Incident: an edge is incident to a vertex if it connects the vertex.
- Degree (or valence) of a vertex: the number of edges attached at it.
- **Neighbor of a vertex**: vertices can be accessed by the edges incident to the vertex.
- Loop: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.

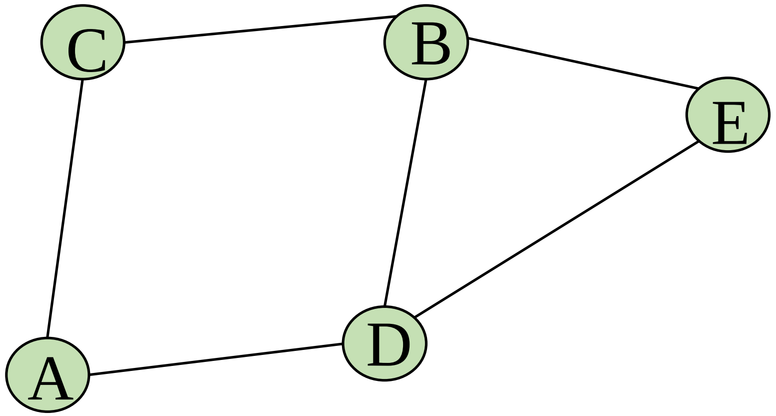
Neighbor of B: {C, D, E}

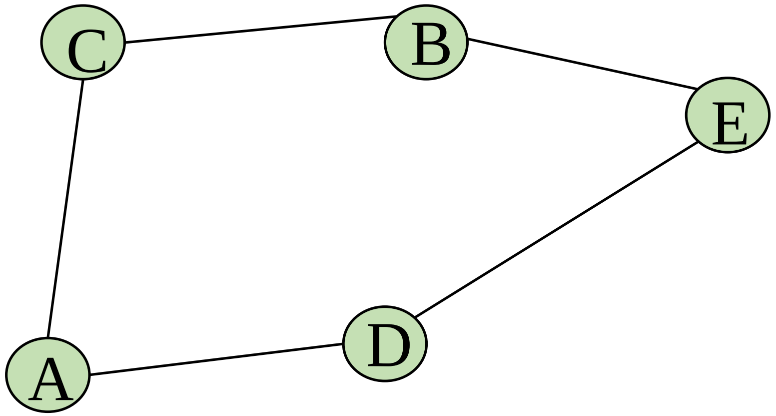


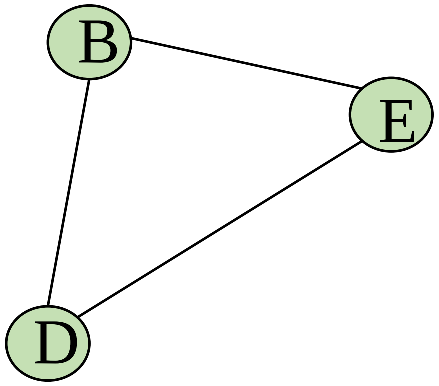
Neighbor of A: {C, D}

Basic concepts about graphs

- Weight: A vertex or an edge can be assigned a value, called weight.
- Incident: an edge is incident to a vertex if it connects the vertex
- Degree (or valence) of a vertex: the number of edges attached at it
- Neighbor of a vertex: vertices can be accessed by the edges incident to the vertex.
- **Loop**: starting at a vertex and then traversing along one or mode edges and finally getting back to the vertex.

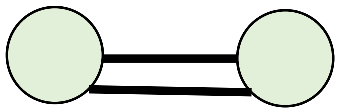


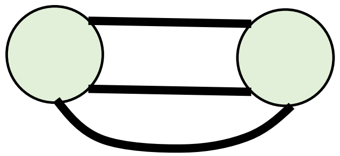




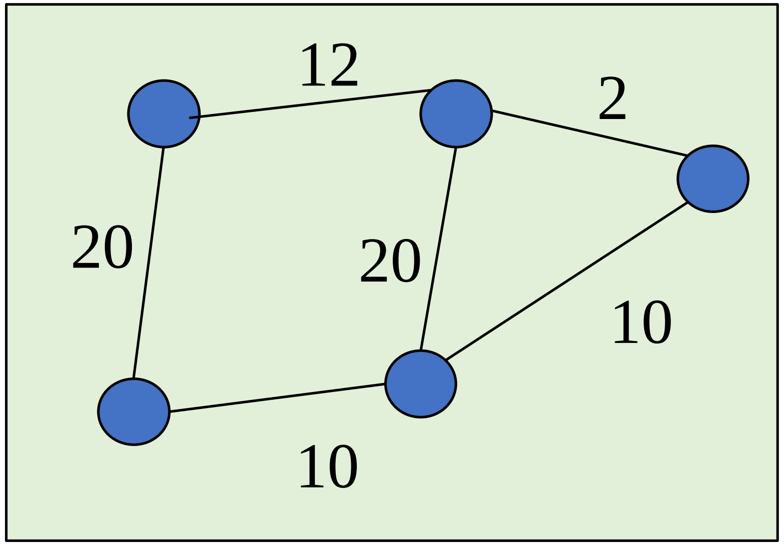
Basic concepts about graphs

- **Parallel edges:** Two or more edges are incident to the same two vertices.

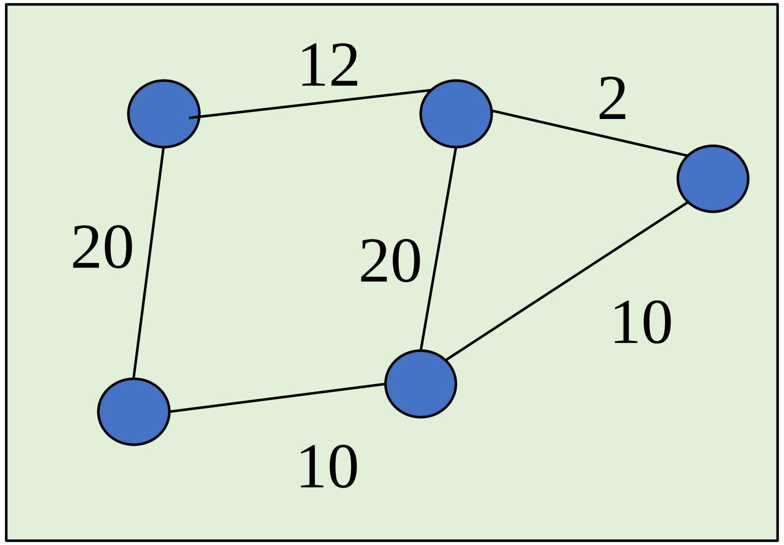


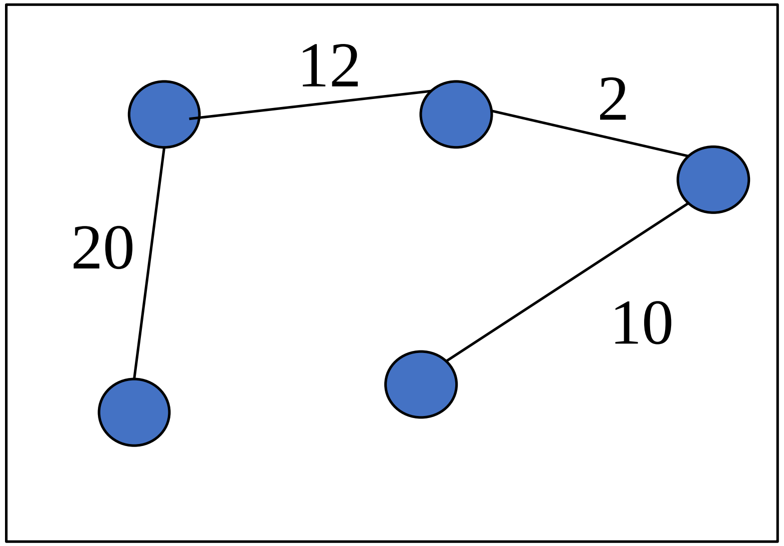


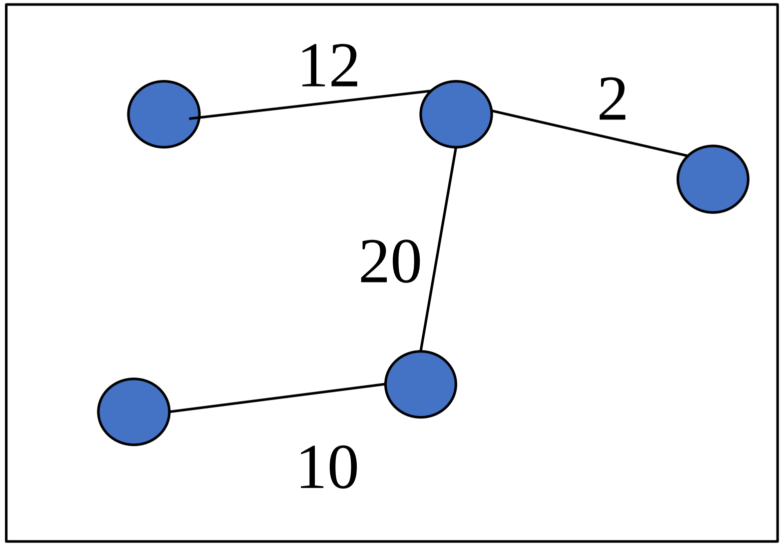
Spanning trees: No loop. All nodes are connected.

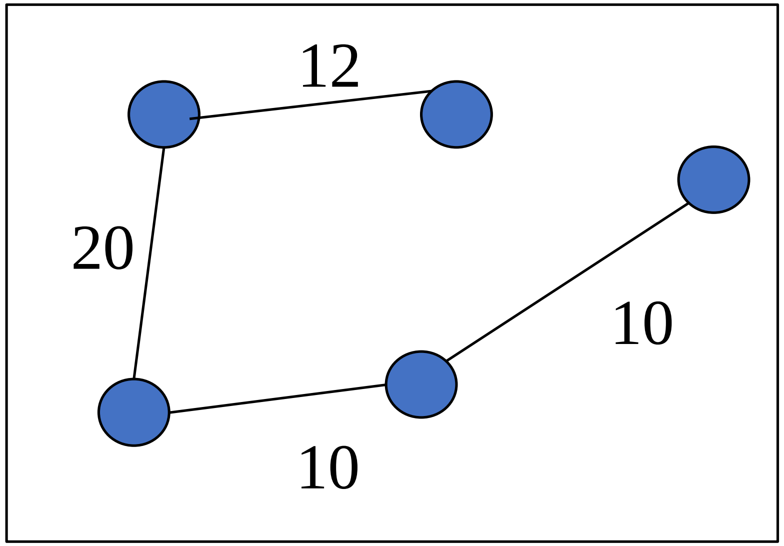


Spanning trees: No loop. All nodes are connected.

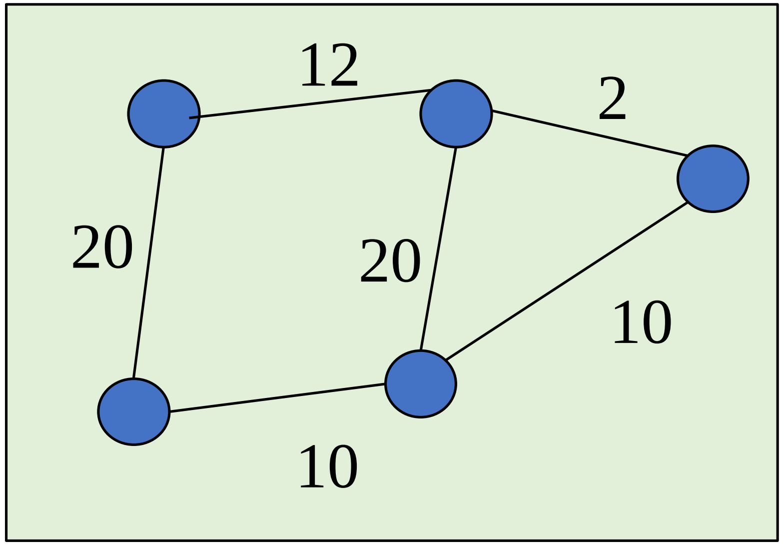


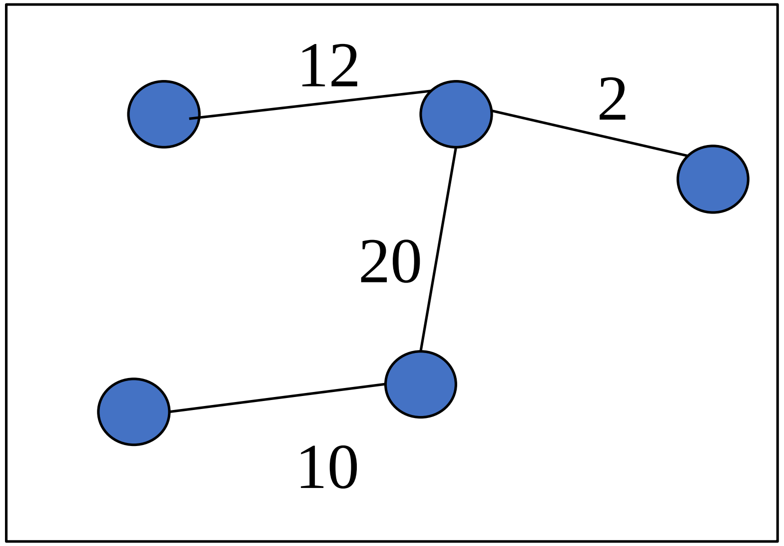




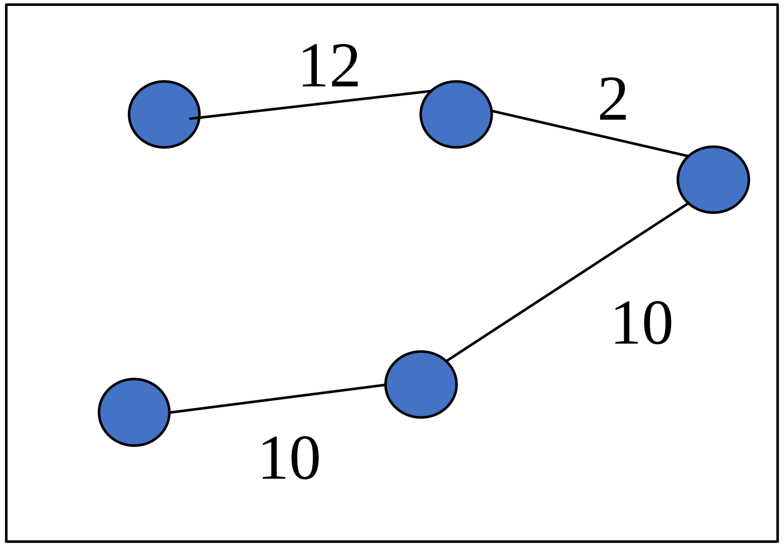


Minimum spanning trees: Spanning trees with the lowest weight



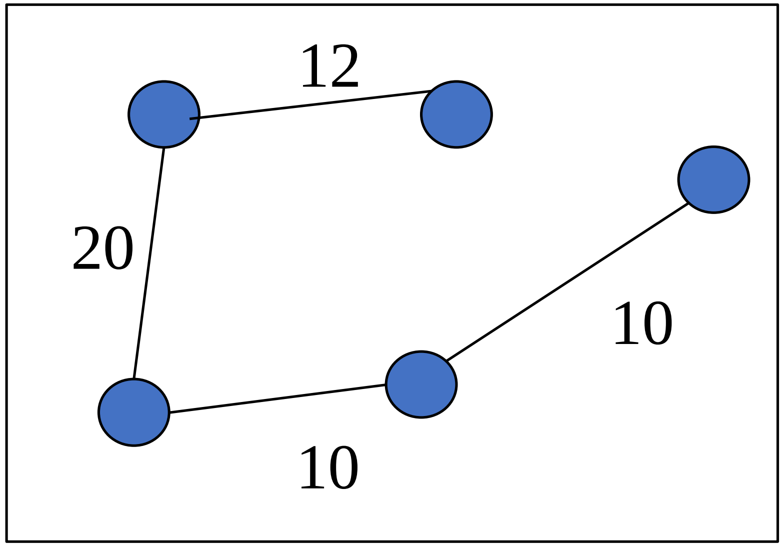


Total weight 44



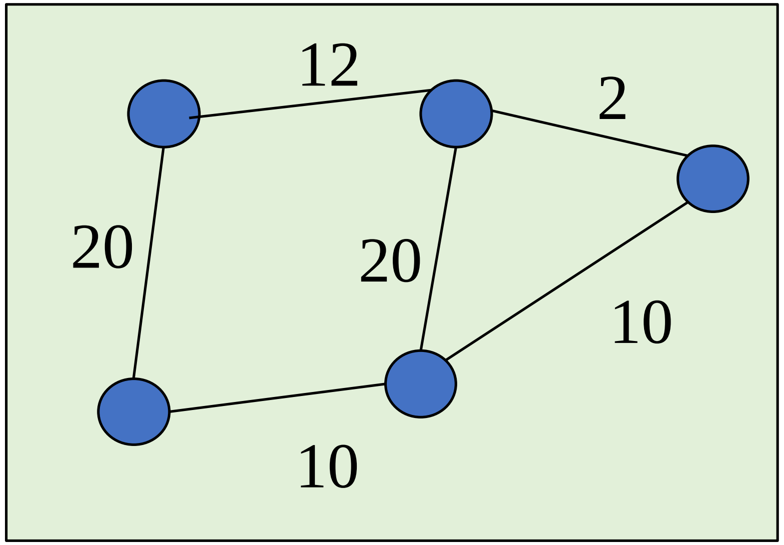
Minimum Spanning tree

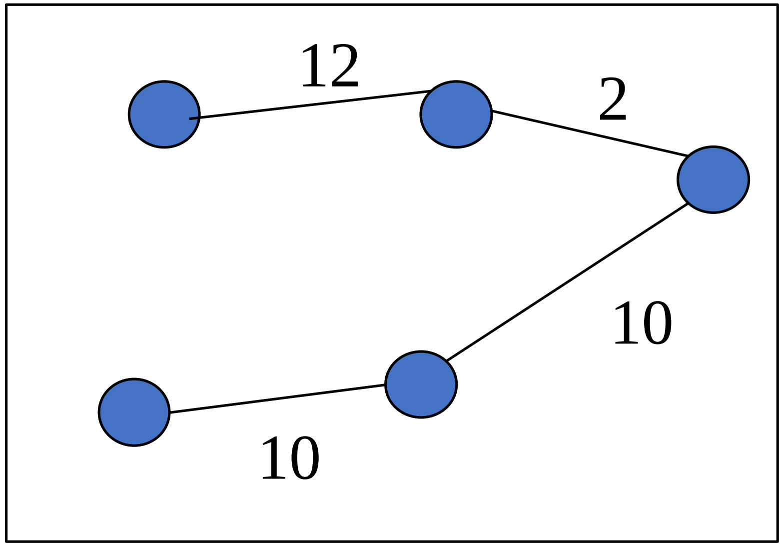
Total weight 34



Total weight 52

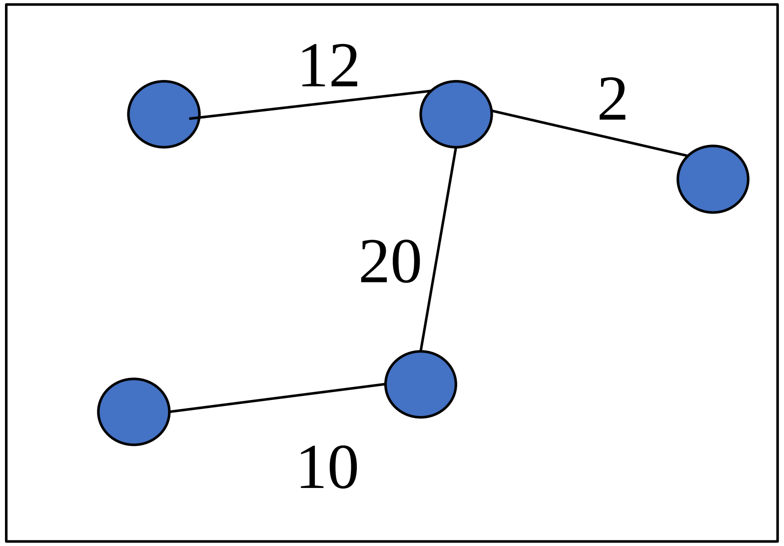
Minimum spanning trees: Spanning trees with the lowest weight



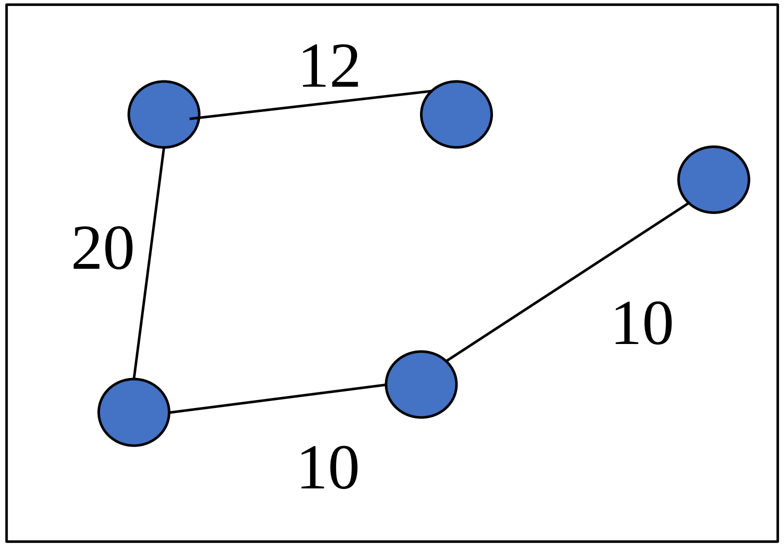


Minimum Spanning tree

Total weight 34



Total weight 44

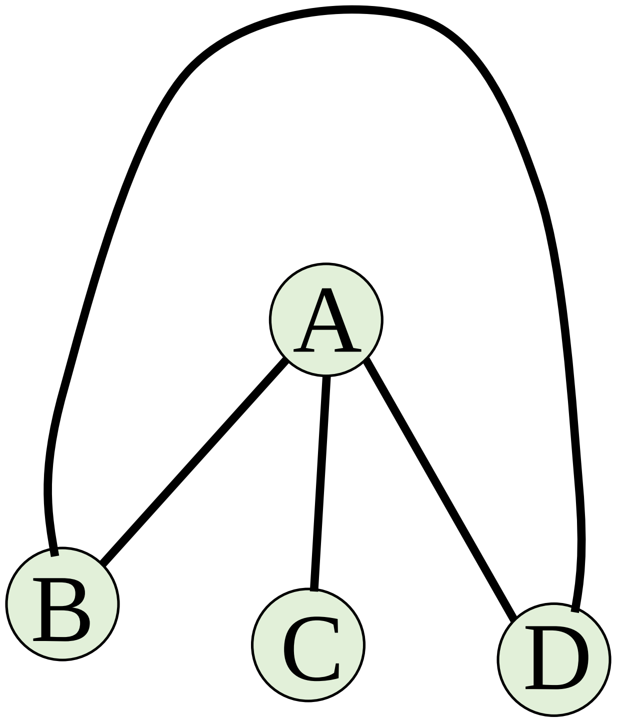


Total weight 52

Undirected graphs

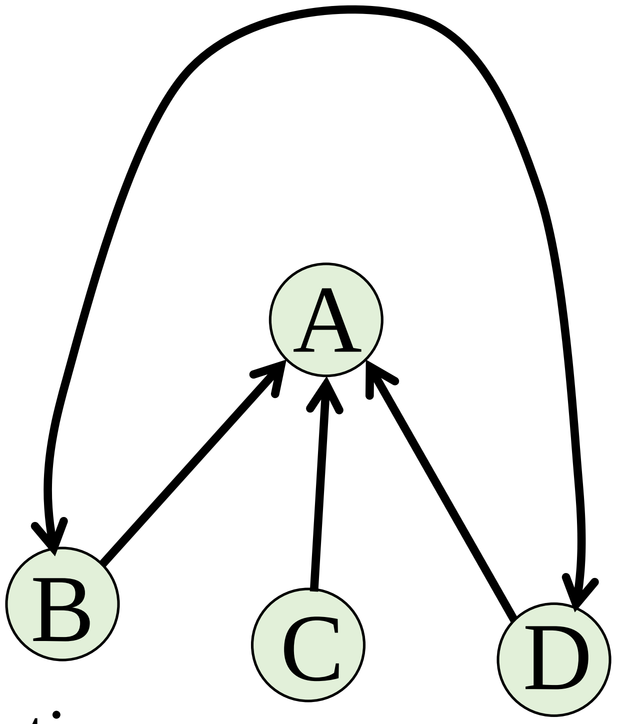
All the edges have no directions.

In an undirected graph:
edge {B, A} is the same as edge {A, B}.



Directed graphs

All the edges have directions.



In a directed graph:
edge {B, A} is not the same as edge {A, B}.
One node is the source and another is the destination.

e.g., for edge {B, A}, B is the source and A is the destination.

To traverse an edge, start from the source node to the destination node.

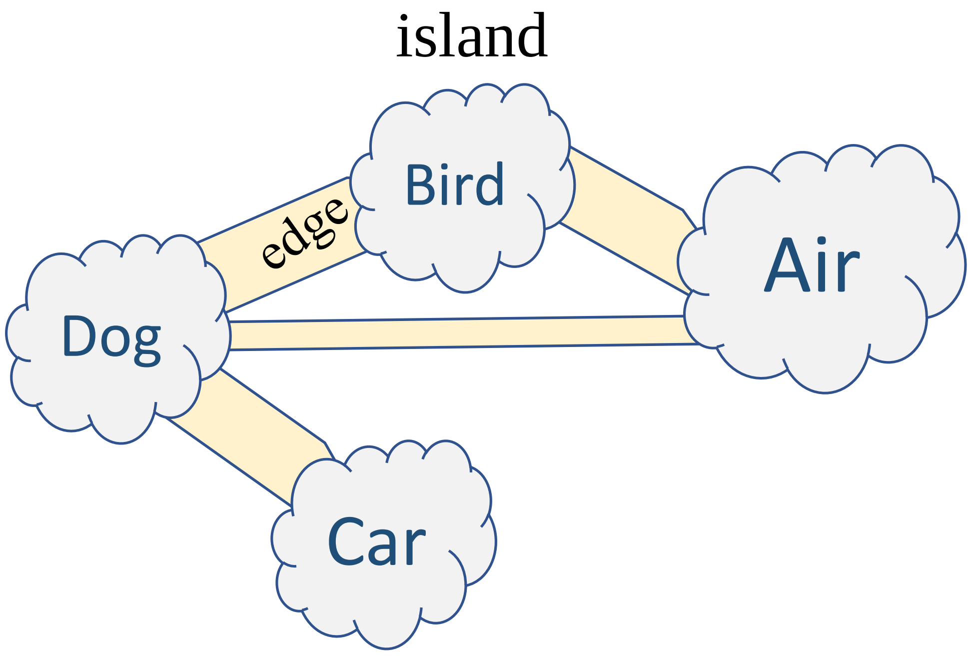
Representation of graphs

Vertex representation:

- ➢ Vertex objects class Vertex { …… }
- ➢ Vertex array vector<Vertex*> vertices;

Edge representation:

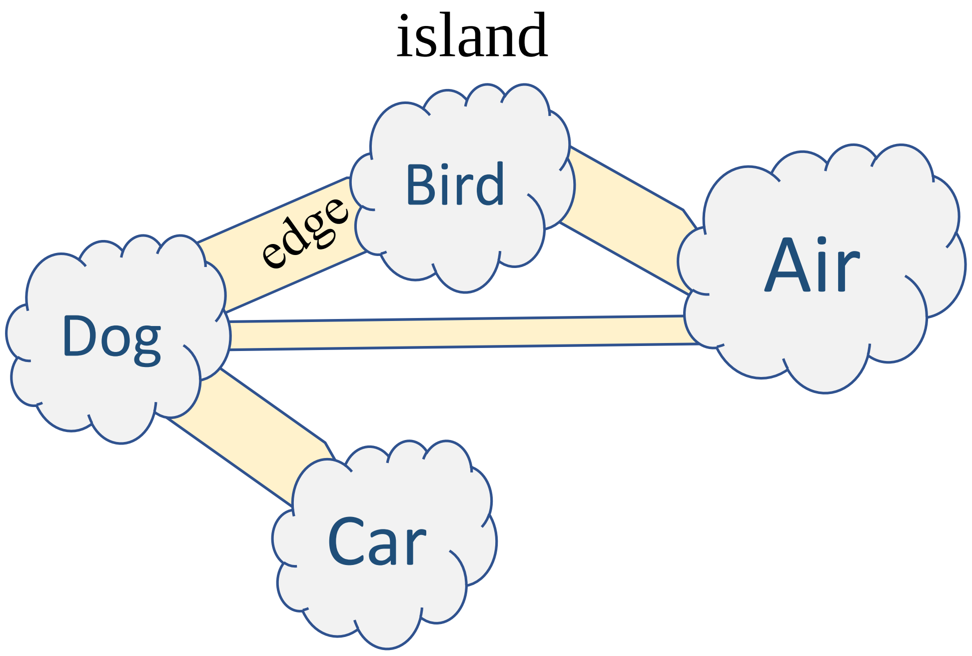
- ➢ Edge objects class Edge { Vertex v1,v2; }
- ➢ Edge array vector<Edge*> edges;
- ➢ Adjacency matrices bool adjMatrix[NumVertices][NumVertices];
- ➢ Adjacency lists vector<vector<int>> adjList;



Representation of vertices

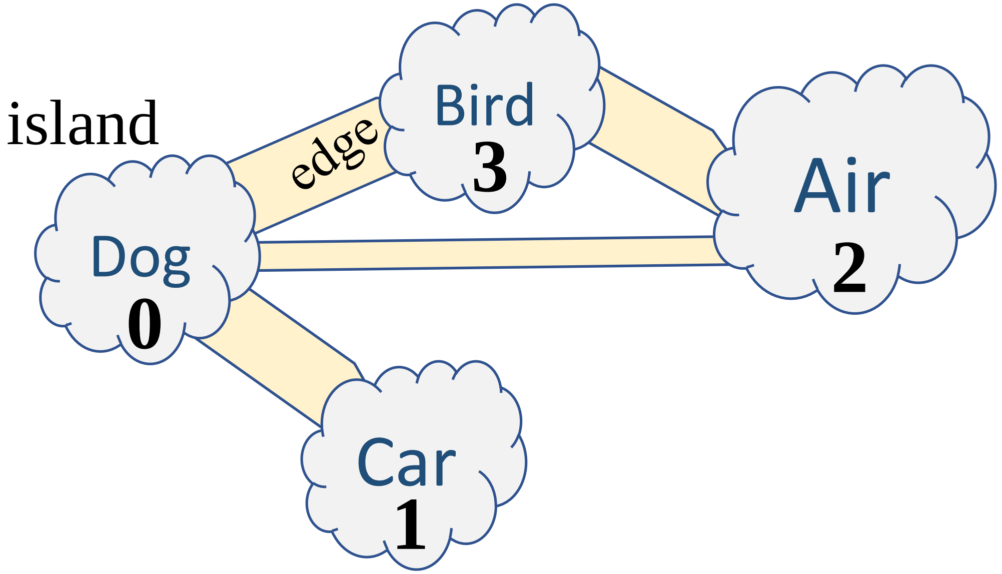
```
class Vertex {
public:
   int vertex_id;
   string name;
   vector3 p;                // position
   double weight;            // cost
   Color color;              // color
   …… // other attributes
public:
   Vertex( ) { }
   Vertex(const string &s, int id) {
      name =s; vertex_id = id;
   }
   Vertex( const vector3 &p);
   Vertex( double x, double y, double z );
};
```

attributes



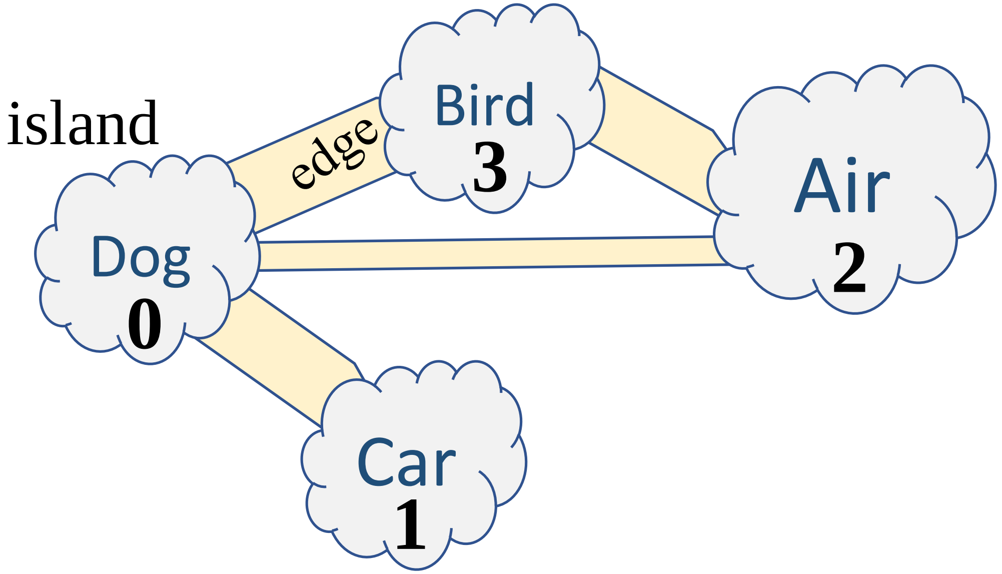
Representation of edges

```
class Edge
{
public:
  int edge_id;
  int u;   // 1st vertex id
  int v;   // 2nd vertex id
  double weight;        ⎫ attributes
  ...... // other attributes ⎭
  Edge(int u, int v)
  {
    this->u = u;
    this->v = v;
  }
};
```



Representation of edges: Edge array

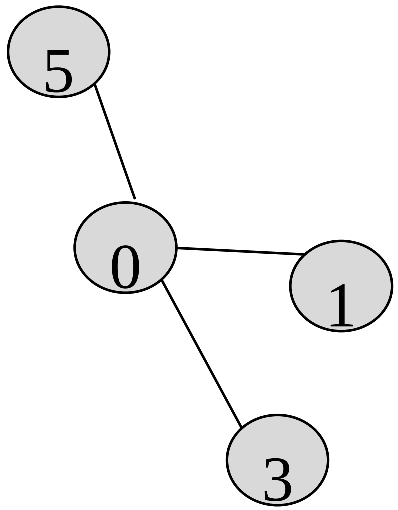
```
class Edge
{
public:
  int edge_id;
  int u;   // 1st vertex id
  int v;   // 2nd vertex id
  double weight;        } attributes
  ...... // other attributes
  Edge(int u, int v)
  {
    this->u = u;
    this->v = v;
  }
};
```




```
vector<Edge*> edgeArr;
Edge *e = new Edge(0, 1);
edgeArr.push_back( e );
edgeArr.push_back( new Edge(0, 3) );
edgeArr.push_back( new Edge(0, 2) );
......
```

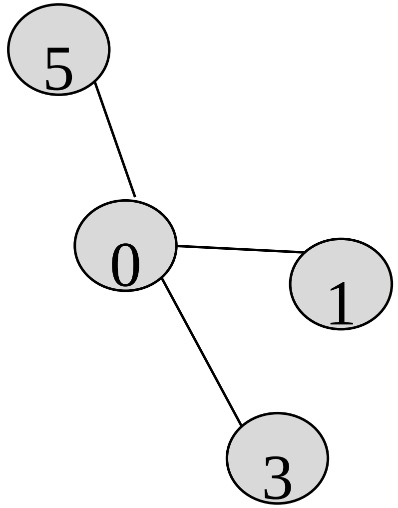
Representation of edges: Edge adjacency matrix

```
int adjacencyMatrix[6][6] = {
   {0, 1, 0, 1, 0, 1},
   {1, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0},
   {1, 0, 0, 0, 0, 0},
   {0, 0, 0, 0, 0, 0},
   {1, 0, 0, 0, 0, 0},
};
```



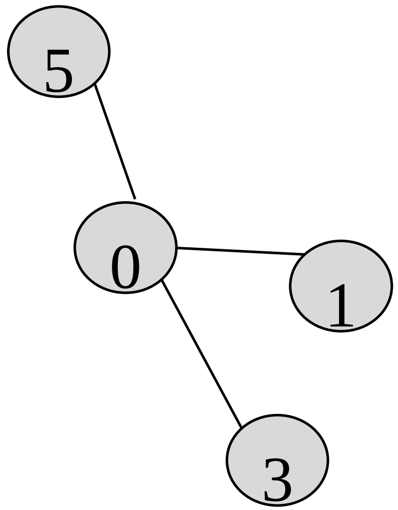
Representation of edges: Edge adjacency matrix

```
int adjacencyMatrix[6][6] = {
  {0, 1, 0, 1, 0, 1}, // vertex 0
  {1, 0, 0, 0, 0, 0}, // vertex 1
  {0, 0, 0, 0, 0, 0}, // ...
  {1, 0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0, 0},
  {1, 0, 0, 0, 0, 0},
};
```



Representation of adjacency edge list: Array list

```
vector<vector<Edge*>> neighbors(6);
neighbors.clear( );
neighbors.resize(6);
neighbors[0].push_back(new Edge(0, 1));
neighbors[0].push_back(new Edge(0, 3));
neighbors[0].push_back(new Edge(0, 5));
neighbors[1].push_back(new Edge(1, 0));
neighbors[3].push_back(new Edge(3, 0));
neighbors[5].push_back(new Edge(5, 0));
...

// each element of neighbors is a vector<Edge*> structure.
```



Graph class

```
template<typename Vertex, typename Edge>
class Graph {
public:
  Graph( )
  Graph( const vector<Vertex> &, const vector<vector<Edge>> &);
  void clear( );
  int addVertex( const Vertex &);  // return vertex id
  int addEdge( int vertex0_id, int vertex1_id); // return edge id
  vector<int> getNeighbors( int v_id );
  vector<int> getIncidentEdges( int v_id );
  int getNumberOfVertices( ) const;
  int getNumberOfEdges( ) const;
  const vector<Vertex> &getVertices( ) const;
  const vector<vector<Edge>> &getEdges( ) const;
};
```

Graph class

```
template<typename Vertex, typename Edge>
class Graph {
public:
  Graph( )
  Graph( const vector<Vertex*> &, const vector<vector<Edge*>> &);
  void clear( );
  int addVertex( const Vertex &);  // return vertex id
  int addEdge( int vertex0_id, int vertex1_id); // return edge id
  vector<int> getNeighbors( int v_id );
  vector<int> getIncidentEdges( int v_id );
  int getNumberOfVertices( ) const;
  int getNumberOfEdges( ) const;
  const vector<Vertex> &getVertices( ) const;
  const vector<vector<Edge>> &getEdges( ) const;
};
```

Graph class

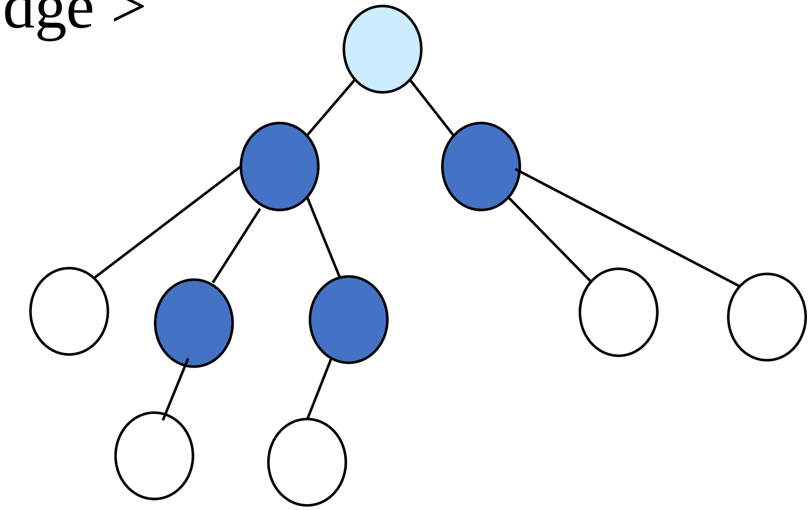
```
template<typename Vertex, typename Edge>
class Graph {
public:
  Graph( )
  Graph( const Graph &);
  void clear( );
  int addVertex( const Vertex &);  // return vertex id
  int addEdge( int vertex0_id, int vertex1_id); // return edge id
  vector<int> getNeighbors( int v_id );
  vector<int> getIncidentEdges( int v_id );
  int getNumberOfVertices( ) const;
  int getNumberOfEdges( ) const;
  const vector<Vertex> &getVertices( ) const;
  const vector<vector<Edge>> &getEdges( ) const;
};
```

Graph class

```
template<typename Vertex, typename Edge >
class Graph {
public:
  Vertex *getVertex(int vertex_id) const;
  Edge *getEdge(int edge_id) const;

  void printf_edges_vertices( ) const;
  void printf_vertices( ) const;
  void printf_edges( ) const;
  void printf_AdjacencyMatrix( ) const;

  Tree dfs(int vertex_id) const; // return a depth-first search tree
  Tree bfs(int vertex_id) const; // return a breadth-first search tree
  Trees mst( ) const; // return a set of minimum spanning trees
};
```

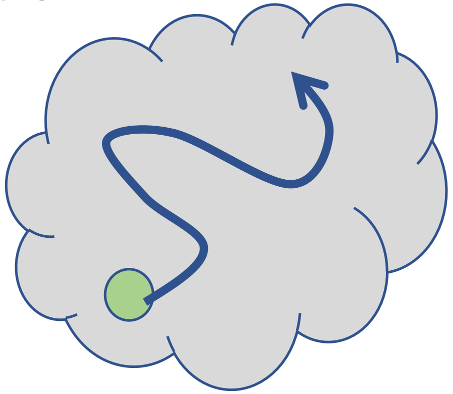
Tree

```
template<typename Vertex, typename Edge >
class Tree {
public:
  Tree( );
  Tree( const Tree & );
  int getRoot( ) const;
  Vertex *getVertex( ) const;
  vector<int> getPath( int vertex_id) const;

};
```



Depth-First Search

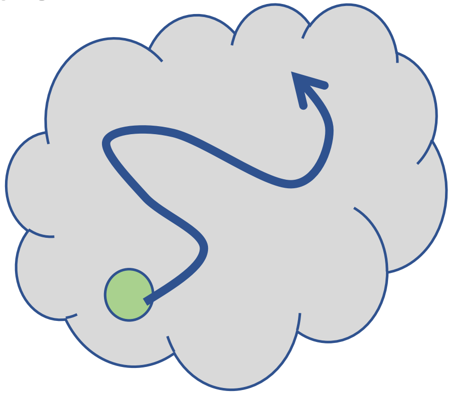
Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.



```
dfs(vertex v)
   visit v;
   for each neighbor w of v
      if (w is not visited)
         dfs(w);
```

Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

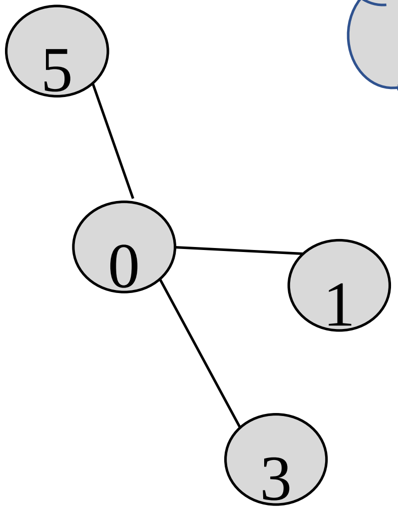


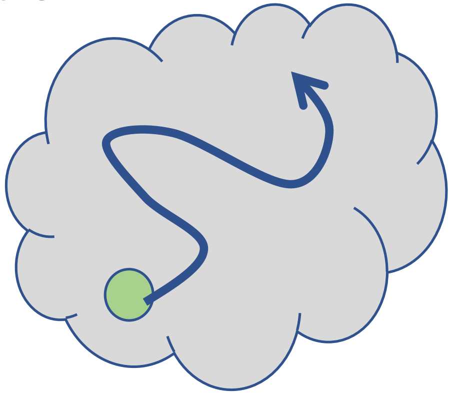
```
dfs(vertex v)
  visit v;
  for each neighbor w of v
    if (w is not visited)
      dfs(w);
```

Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

```
dfs(vertex v)
  visit v;
  for each neighbor w of v
    if (w is not visited)
      dfs(w);
```

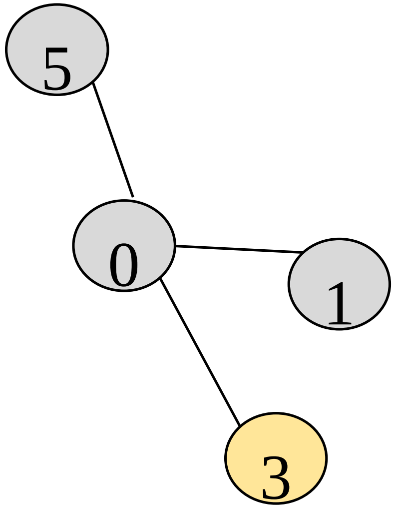




Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

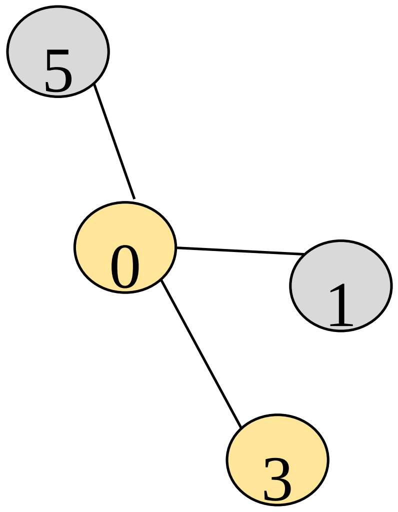
```
dfs(vertex v)
  visit v;
  for each neighbor w of v
    if (w is not visited)
      dfs(w);
```



Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

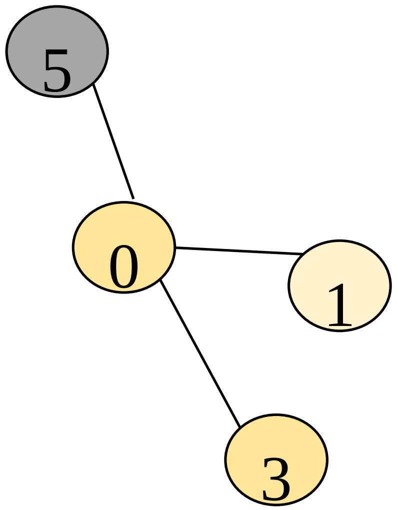
```
dfs(vertex v)
  visit v;
  for each neighbor w of v
    if (w is not visited)
      dfs(w);
```

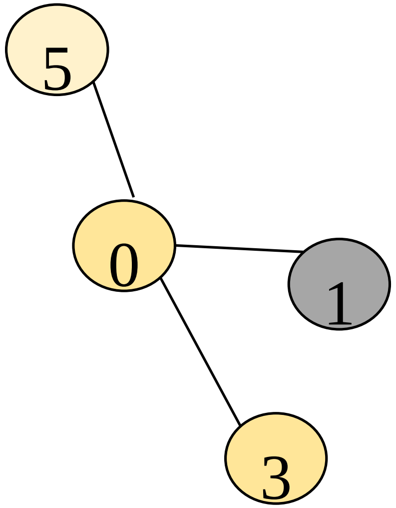


Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

```
dfs(vertex v)
   visit v;
   for each neighbor w of v
      if (w is not visited)
         dfs(w);
```

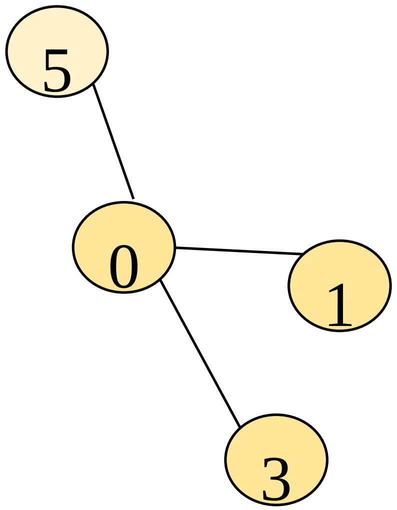


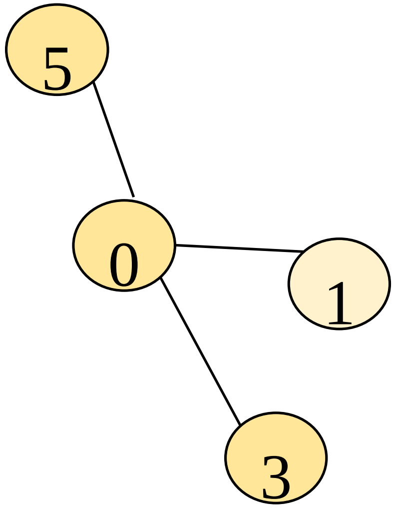


Depth-First Search

Traverse a graph or a tree in a depth-first manner. That's that, a node is explored if it has not been visited.

```
dfs(vertex v)
  visit v;
  for each neighbor w of v
    if (w is not visited)
      dfs(w);
```





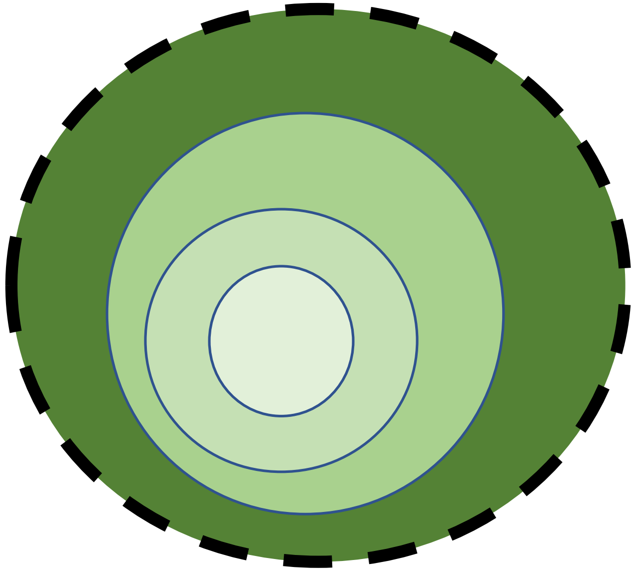

```
Possible result:
3,0,1,5
3,0,5,1
```

Breadth-first search

Traverse a graph or a tree in a breadth-first manner. That's that, the neighbors of a node are visited first.

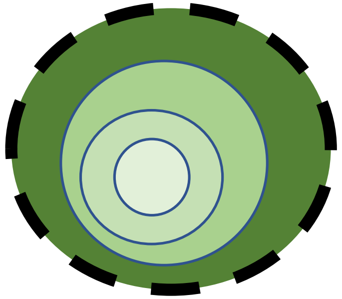
Breadth-first search

Traverse a graph or a tree in a breadth-first manner. That’s that, the neighbors of a node are visited first.



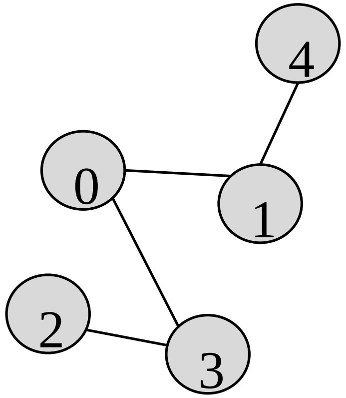
Breadth-first search

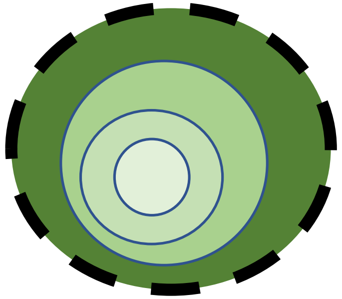
Traverse a graph or a tree in a breadth-first manner. That's that, the neighbors of a node are visited first.



Breadth-first search

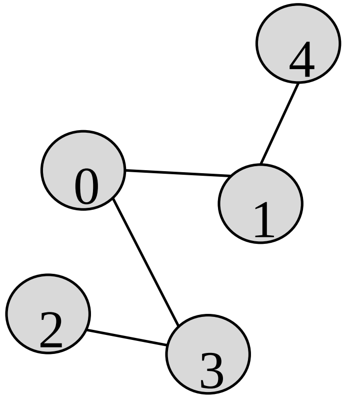
Traverse a graph or a tree in a breadth-first manner. That’s that, the neighbors of a node are visited first.



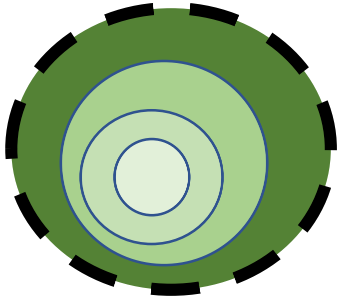


Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}    // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
//need a structure
//to store the result
```

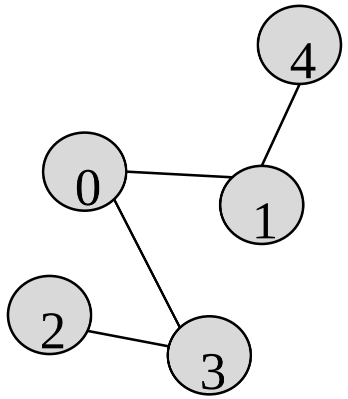


bfs(0)

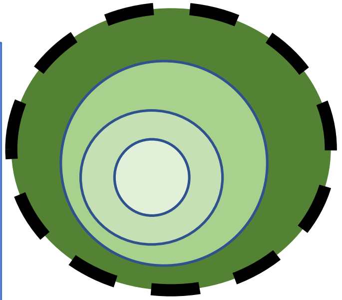


Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}    // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
//need a structure
//to store the result
```



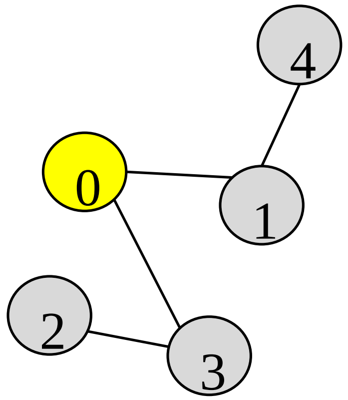
bfs(0)
q = { 0 }



Breadth-first search

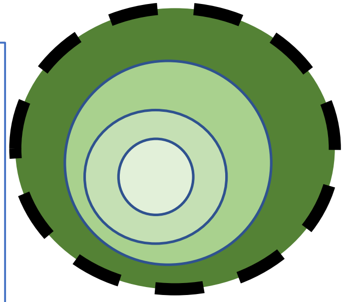
```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

N(x) : neighbors of x



bfs(0)
q = { 0 }
q = { } dequeue 0
q = {1, 3} add N(0)

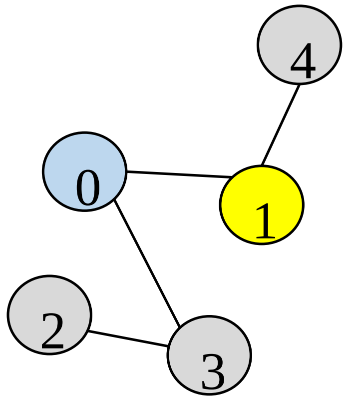
0



Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

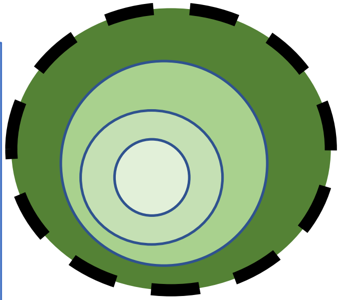
N(x) : neighbors of x



bfs(0)

q = { 0 }	
q = { }	dequeue 0
q = {1, 3}	add N(0)
q = {3}	dequeue 1
q = {3, 4}	add N(1)

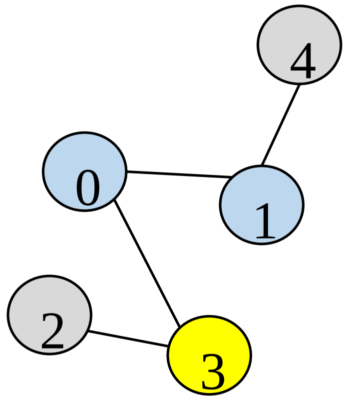
0
1



Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

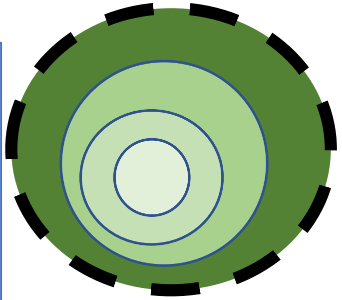
N(x) : neighbors of x



bfs(0)

q = { 0 }	
q = { }	dequeue 0
q = {1, 3}	add N(0)
q = {3}	dequeue 1
q = {3, 4}	add N(1)
q = {4}	dequeue 3
q = {4, 2}	add N(3)

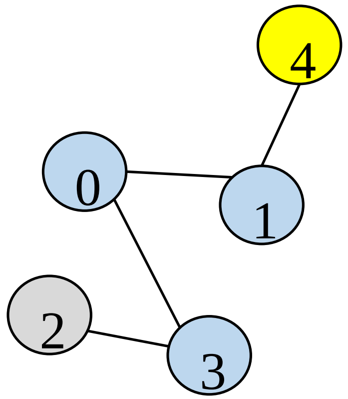
0
1
3



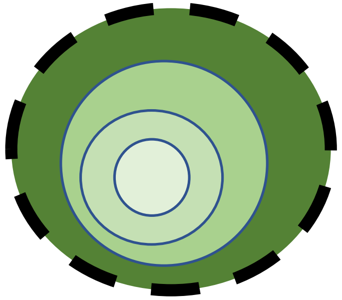
Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

N(x) : neighbors of x



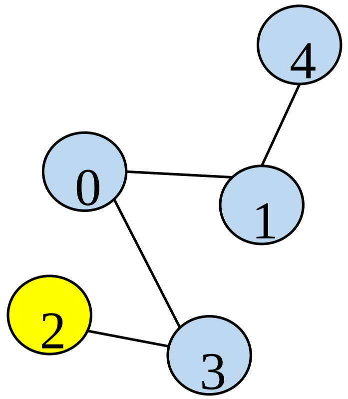
bfs(0)		0
q = { 0 }		1
q = { }	dequeue 0	3
q = {1, 3}	add N(0)	4
q = {3}	dequeue 1	2
q = {3, 4}	add N(1)	
q = {4}	dequeue 3	
q = {4, 2}	add N(3)	
q = {2}	dequeue 4, add N(4)	



Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

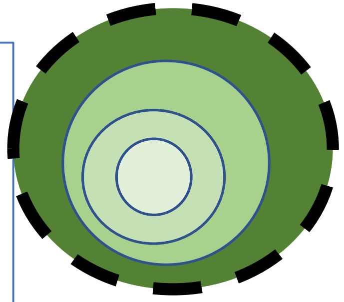
N(x) : neighbors of x



bfs(0)

q = { 0 }	
q = { }	dequeue 0
q = {1, 3}	add N(0)
q = {3}	dequeue 1
q = {3, 4}	add N(1)
q = {4}	dequeue 3
q = {4, 2}	add N(3)
q = {2}	dequeue 4, add N(4)
q = { }	dequeue 2, add N(2)

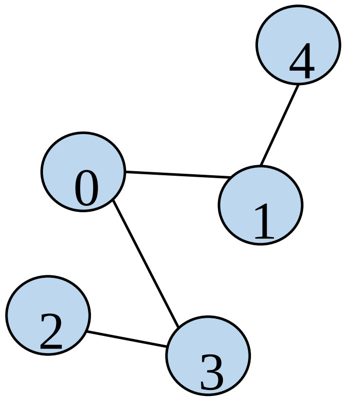
0
1
3
4
2



Breadth-first search

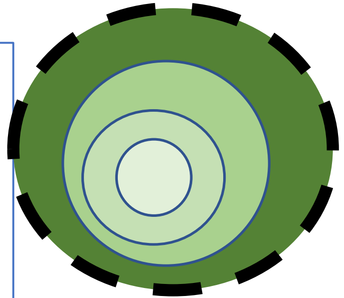
```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}    // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

N(x) : neighbors of x



bfs(0)	
q = { 0 }	
q = { }	dequeue 0
q = {1, 3}	add N(0)
q = {3}	dequeue 1
q = {3, 4}	add N(1)
q = {4}	dequeue 3
q = {4, 2}	add N(3)
q = {2}	dequeue 4, add N(4)
q = { }	dequeue 2, add N(2)

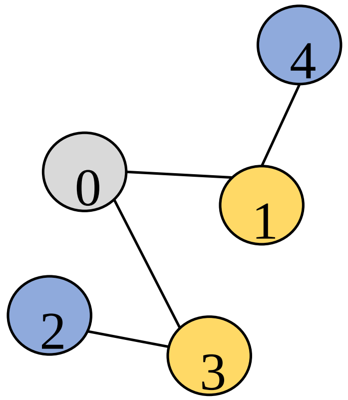
0
1
3
4
2



Breadth-first search

```
bfs(vertex v) {
  use a queue, q = { }, for storing vertices to be visited;
  q ← q + {v}     // add v into the queue;
  while q != { } {   // while q is non-empty
    dequeue a vertex, say u, from q
    visit u;
    for each neighbor w of u
      if w has not been visited {
        q ← q + {w}
      }
  }
}
```

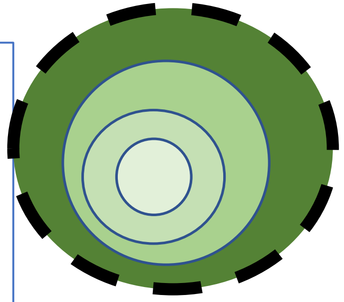
N(x) : neighbors of x



bfs(0)

q = { 0 }	
q = { }	dequeue 0
q = {1, 3}	add N(0)
q = {3}	dequeue 1
q = {3, 4}	add N(1)
q = {4}	dequeue 3
q = {4, 2}	add N(3)
q = {2}	dequeue 4, add N(4)
q = { }	dequeue 2, add N(2)

0
1
3
4
2



Exercises on the depth-first search

- Detect a loop in a graph
- Find a loop in a graph
- Detect whether there is a path connecting two vertices
- Determining whether a path connects two vertices

Breadth-first search

- Path finding
- Detect whether there is a cycle in the graph.
- Find a cycle in the graph
- Determine whether a graph is bipartite.
- GPS navigation systems: find positions of all neighbors

Supplemental Materials

Graphs

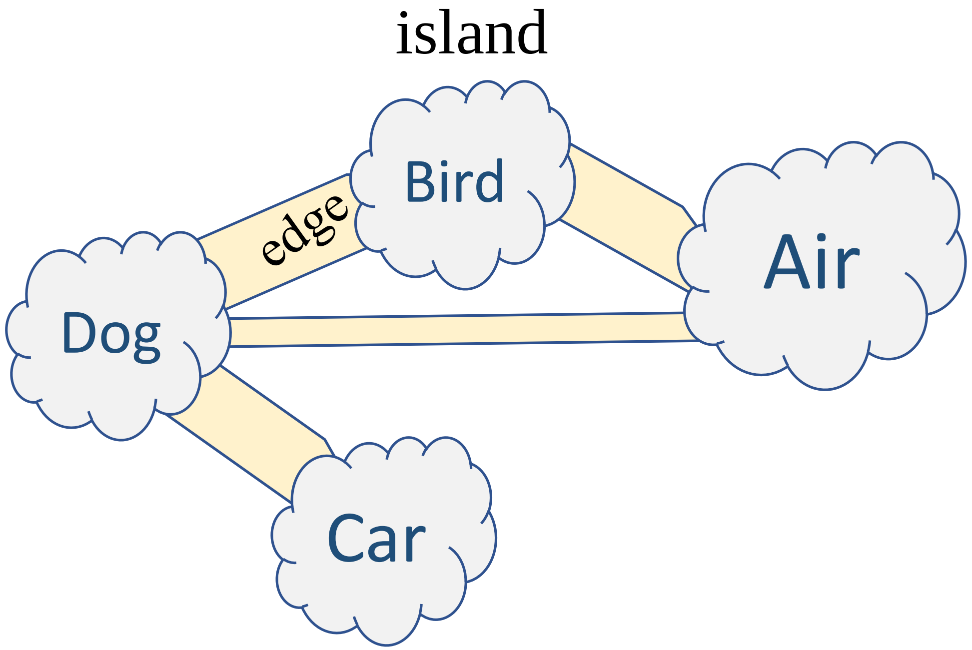
- A graph, denoted as G = (V, E), consists two components V and E, where V is a set of nodes (or vertices) and E is a set of edges.
- $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$, where n is the number of nodes and m is the number of edges.
- An edge e connects two nodes in V. For example, if v1 and v2 are connected, we have e = {v1, v2}.

Representation of vertices

string vertices[ ] = {“Air”, “Bird”, “Dog”, “Car”};

vector<string> vertices;
vertices.emplace_back(“Air”);
…

class Vertex {
public:
  string name;
  int vertex_id;
  …… // other attributes
  Vertex( ) { }
  Vertex(const string &s, int id) { name =s; vertex_id = id;}
};

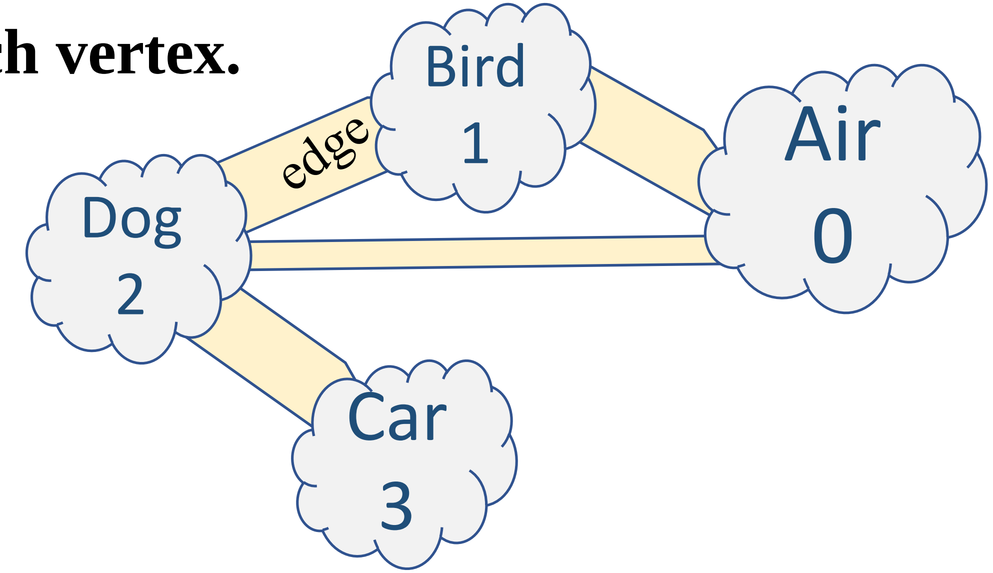


vector<Vertex*> vPtr;
vPtr.push_back(new Vertex(“Air”));

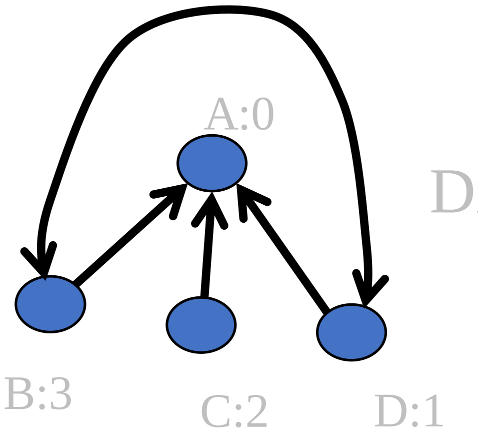
Representation of edges: Edge array

Assign a unique number to each vertex.

int edges[ ][2] = {
  {0,1}, {0, 2},
  {1, 2},
  {2, 3}
};



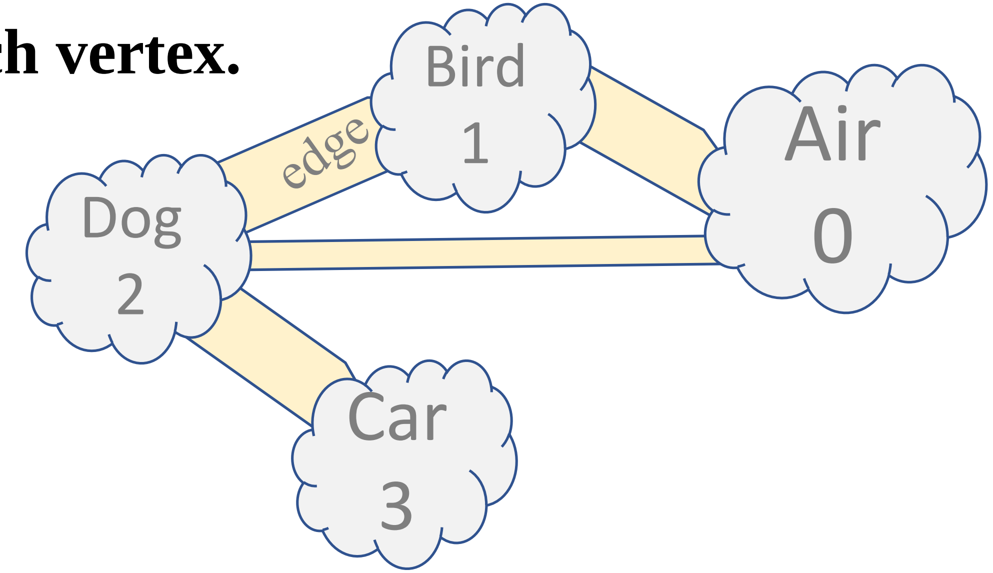
int dir_edges[ ][2] = {
  {1, 0}, {1, 3}
  {2, 0},
  {3,0}, {3, 1}
};



Directed graph

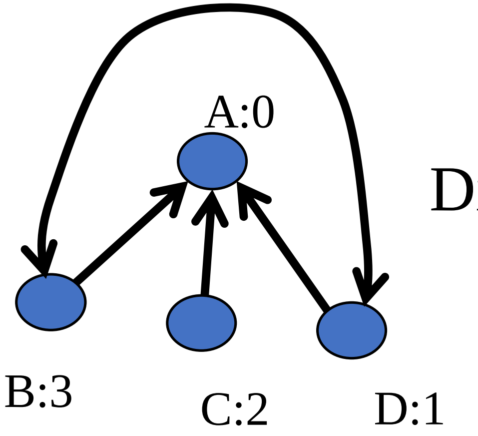
Representation of edges: Edge array

Assign a unique number to each vertex.




```
int edges[ ][2] = {
  {0,1}, {0, 2},
  {1, 2},
  {2, 3}
};
```

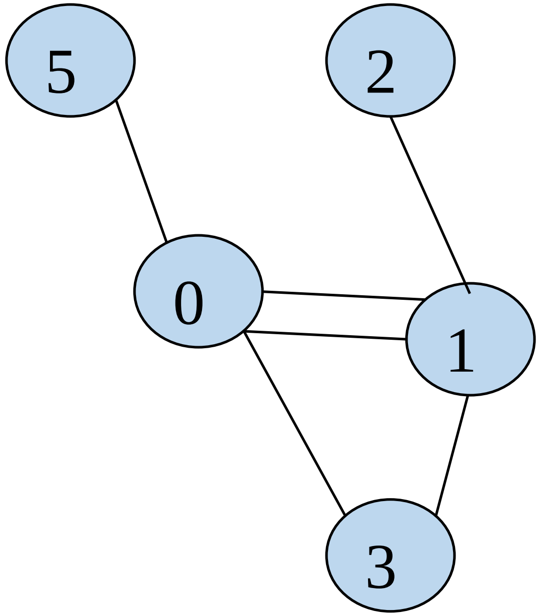
```
int dir_edges[ ][2] = {
  {1, 0}, {1, 3}
  {2, 0},
  {3,0}, {3, 1}
};
```



Directed graph

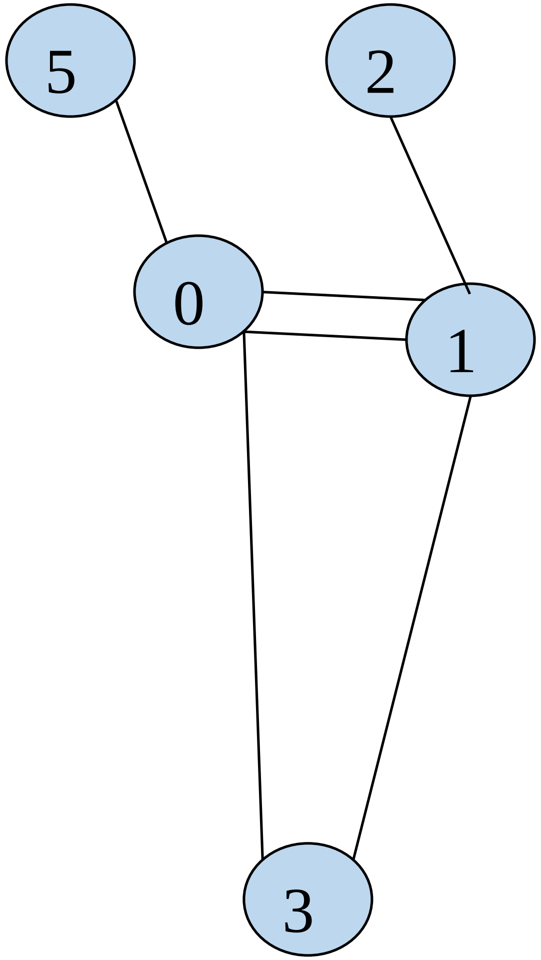
Representation of edges: Edge array

```
int edges[ ][2] = {
  {0, 1}, {0, 3}, {0, 5},
  {1, 0}, {1, 2}, {1, 3},
  ……
};
```



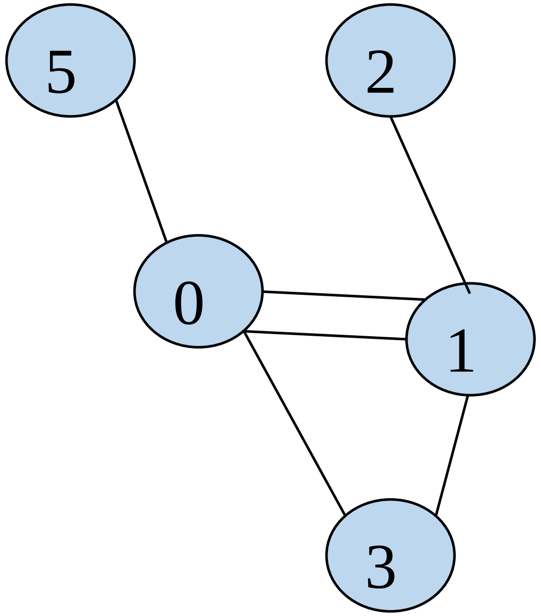
Representation of edges: Edge array

```
int edges[ ][2] = {
  {0, 1}, {0, 3}, {0, 5},
  {1, 0}, {1, 2}, {1, 3},
  ……
};
```

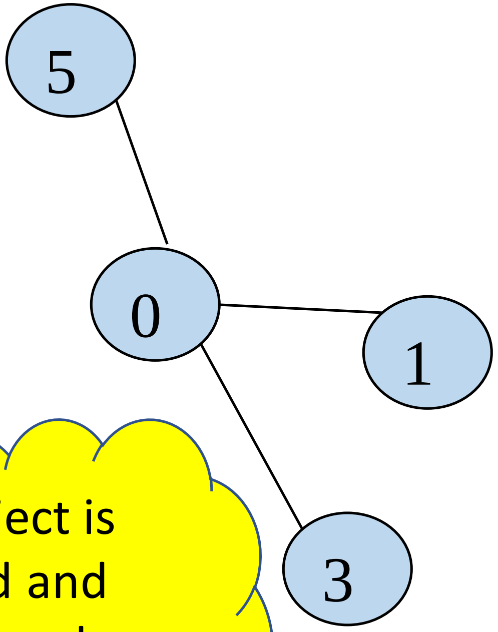


Representation of edges: Edge object

```
class Edge
{
public:
  int edge_id;
  int u;
  int v;
  double weight;
  ...... // other attributes
  Edge(int u, int v)
  {
    this->u = u;
    this->v = v;
  }
};
```



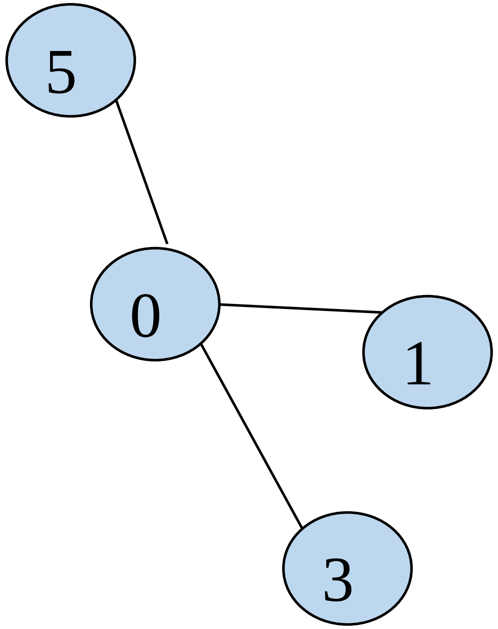
Representation of edges: Edge object




```
vector<Edge> edges;

edges.push_back(Edge(0, 1));
edges.push_back(Edge(0, 3));
edges.push_back(Edge(0, 5));
...

//Alternative
Edges.emplace_back (0, 1);
Edges.emplace_back (0, 3);
Edges.emplace_back (0, 5);
```

The object is created and destroyed after being used.

Representation of edges: Edge object

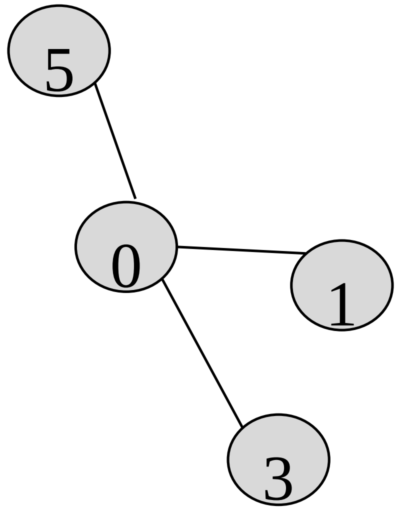



```
vector<Edge> edges;

edges.push_back(Edge(0, 1));
edges.push_back(Edge(0, 3));
edges.push_back(Edge(0, 5));
...

//Alternative
Edges.emplace_back (0, 1);
Edges.emplace_back (0, 3);
Edges.emplace_back (0, 5);
```

C11 version.
Accept argument(s)

Representation of edges: Edge array

```
vector<Edge*> edgeVector;
Edge *e = new Edge(0, 1);
edgeVector.push_back( e );
edgeVector.push_back( new Edge(0, 3) );
edgeVector.push_back( new Edge(0, 5) );
...
```




```
vector<Edge> edgeVector;
edgeVector.push_back(Edge(0, 1));
edgeVector.push_back(Edge(0, 3));
edgeVector.push_back(Edge(0, 5));
...
```

Representation of adjacency edge list: Array list

```
vector<vector<Edge*>> neighbors(12);
neighbors[0].push_back(new Edge(0, 1));
neighbors[0].push_back(new Edge(0, 3));
neighbors[0].push_back(new Edge(0, 5));
neighbors[1].push_back(new Edge(1, 0));
neighbors[1].push_back(new Edge(1, 2));
neighbors[1].push_back(new Edge(1, 3));
...
...

vector<vector<vector<vector<Edge>>*>> x;
…
x[0][1][2][3][][][][][]……
```

Representation of adjacency edge list: Array list

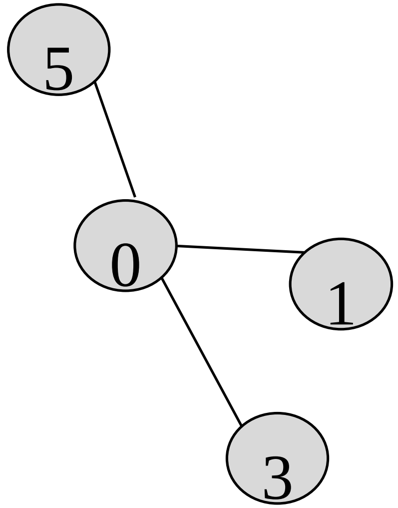
```
vector<vector<Edge*>> neighbors(12);
neighbors[0].push_back(new Edge(0, 1));
neighbors[0].push_back(new Edge(0, 3));
neighbors[0].push_back(new Edge(0, 5));
neighbors[1].push_back(new Edge(1, 0));
neighbors[1].push_back(new Edge(1, 2));
neighbors[1].push_back(new Edge(1, 3));
...
...

vector<vector<vector<vector<Edge>>*>> x;
vector<vector<vector<Edge>>*> y;
x.push_back(y);
```

Representation of edges: Adjacency vertex list

vector<**int**> neighbors(**6**);

neighbors[0] = { 1, 3, 5};
neighbors[1] = { 0 };
neighbors[2] = { };
neighbors[3] = { 0 };
neighbors[4] = { };
neighbors[5] = { 0 };

neighbors[0].size( ) := 3
neighbors[0][2] := 5
neighbors[3][0] := 0



Representation of edges: Adjacency vertex list

```
vector<Edge> neighbors(6);

neighbors[0] := { Edge(0, 1), Edge(0, 3), Edge(0,5) };
neighbors[1] := { Edge(1, 0) };
neighbors[2] := {  };
neighbors[3] := { Edge(3, 0) };
neighbors[4] := {  };
neighbors[5] := { Edge(5, 0) };
```