Graphs and Unordered Collections

2501ICT

Logan

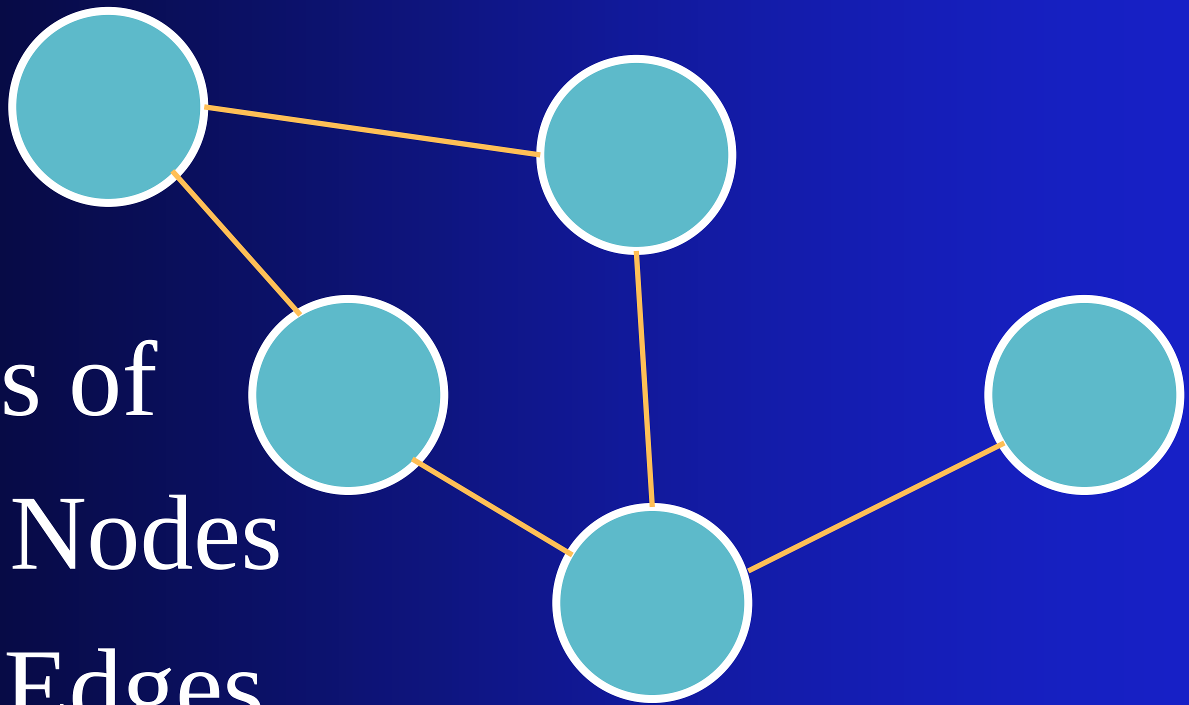
Contents

- Definition
- Graph Representations
- Basic Operations
  - Graph Traversals
  - Topological Sort
  - Trees within Graphs
- Unordered Collections

Graph Definition

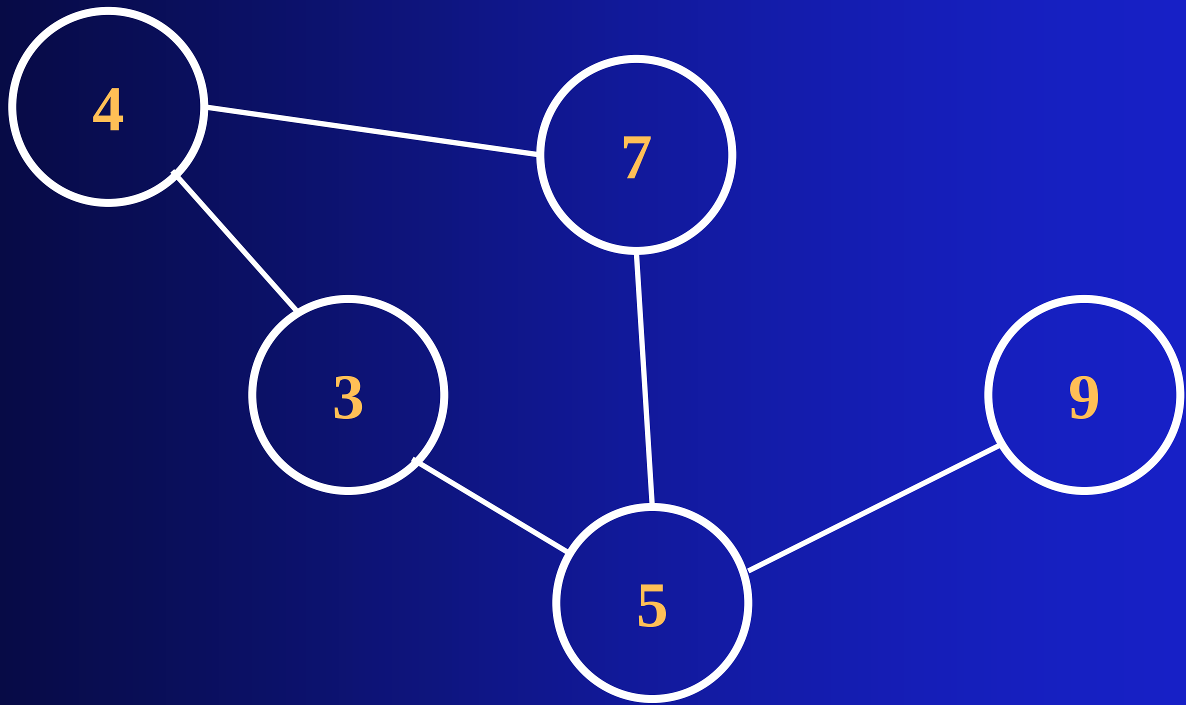
- Multiple Successors/Predecessors
  - Lists: *one* successor, *one* predecessor
  - Trees: *several* successors, *one* predecessor
- A Graph is a
  - Set of points connected by line segments
  - Points are called vertices (**V**) or nodes
  - Lines are called edges (**E**)

Unlabelled Graphs

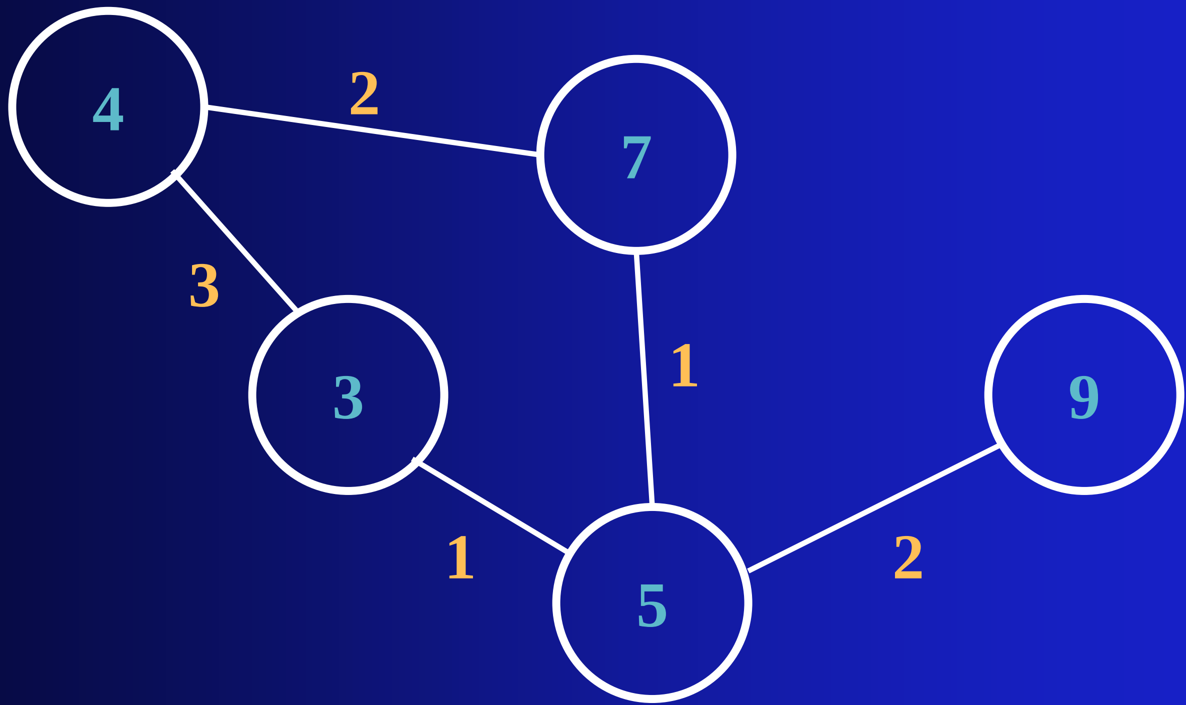


- Sets of
  - **V**: Nodes
  - **E**: Edges
    - Each Edge $e \in E$ connects two Nodes $v \in V$

Labelled Vertices

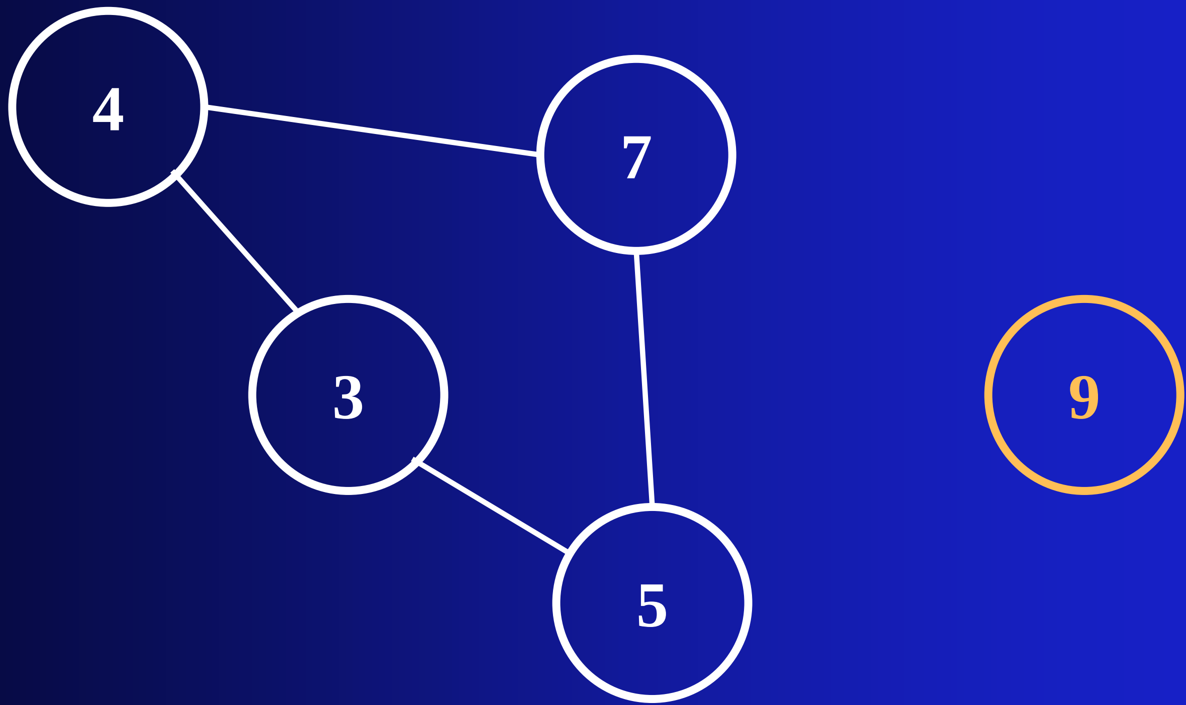


Weighted Graph



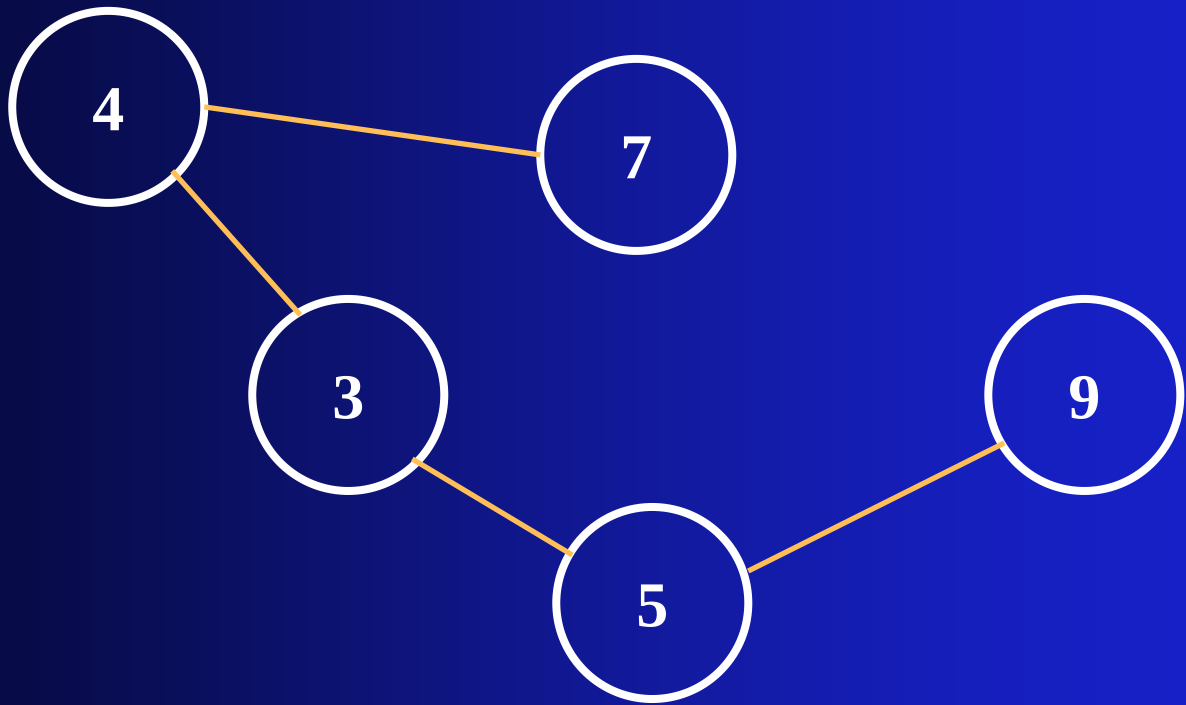
- Labelled Vertices and Edges

Disconnected Graph



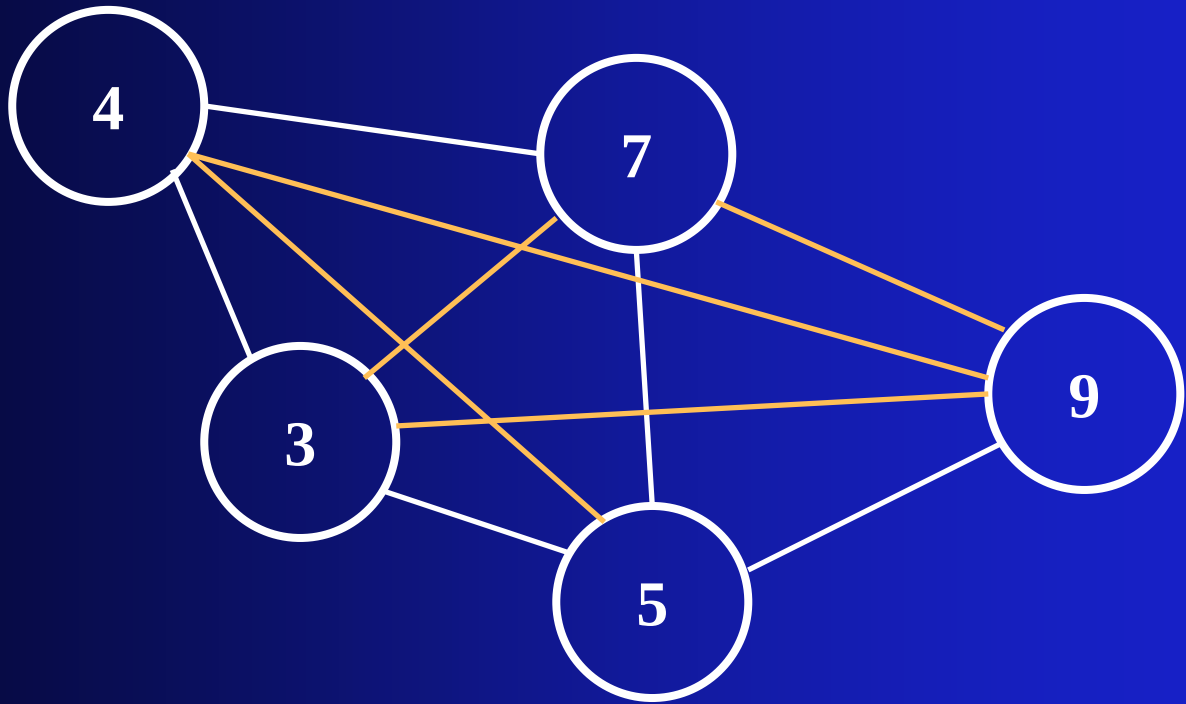
- One or more Nodes not connected

Connected Graph



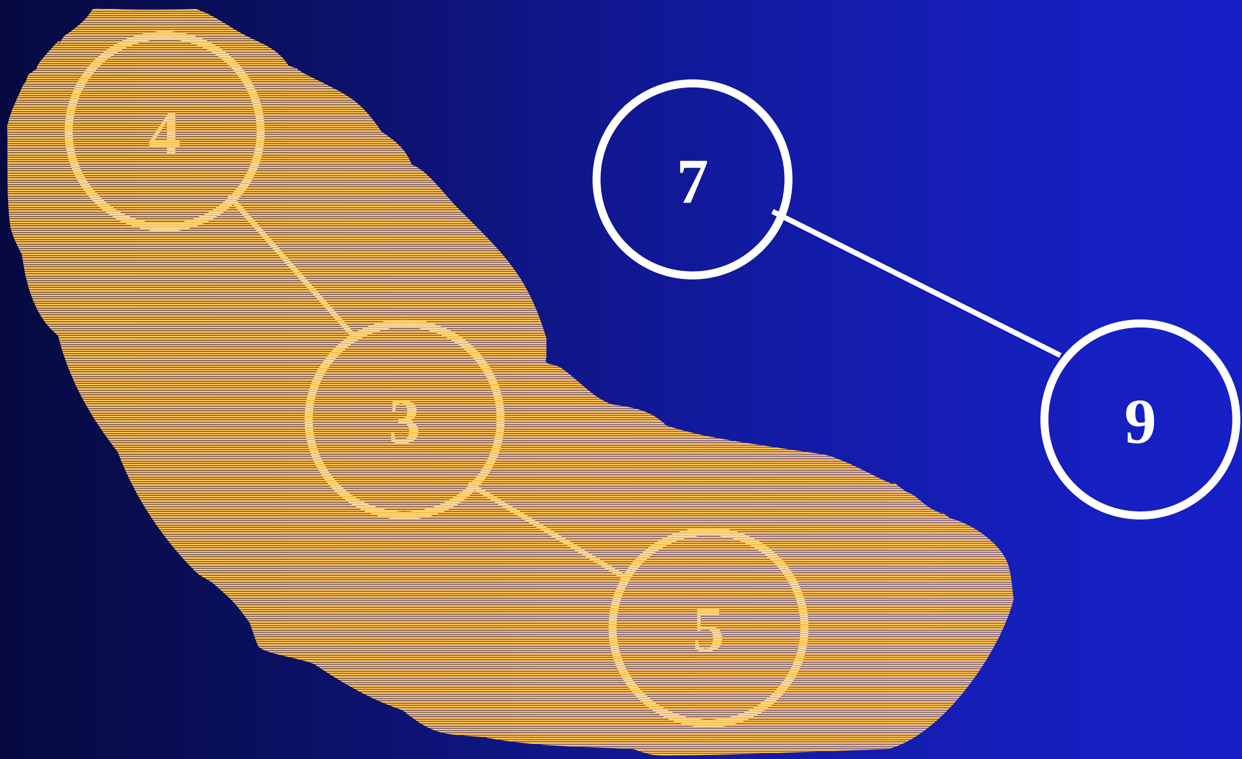
- A Path from each to every other Vertex

Complete Graph

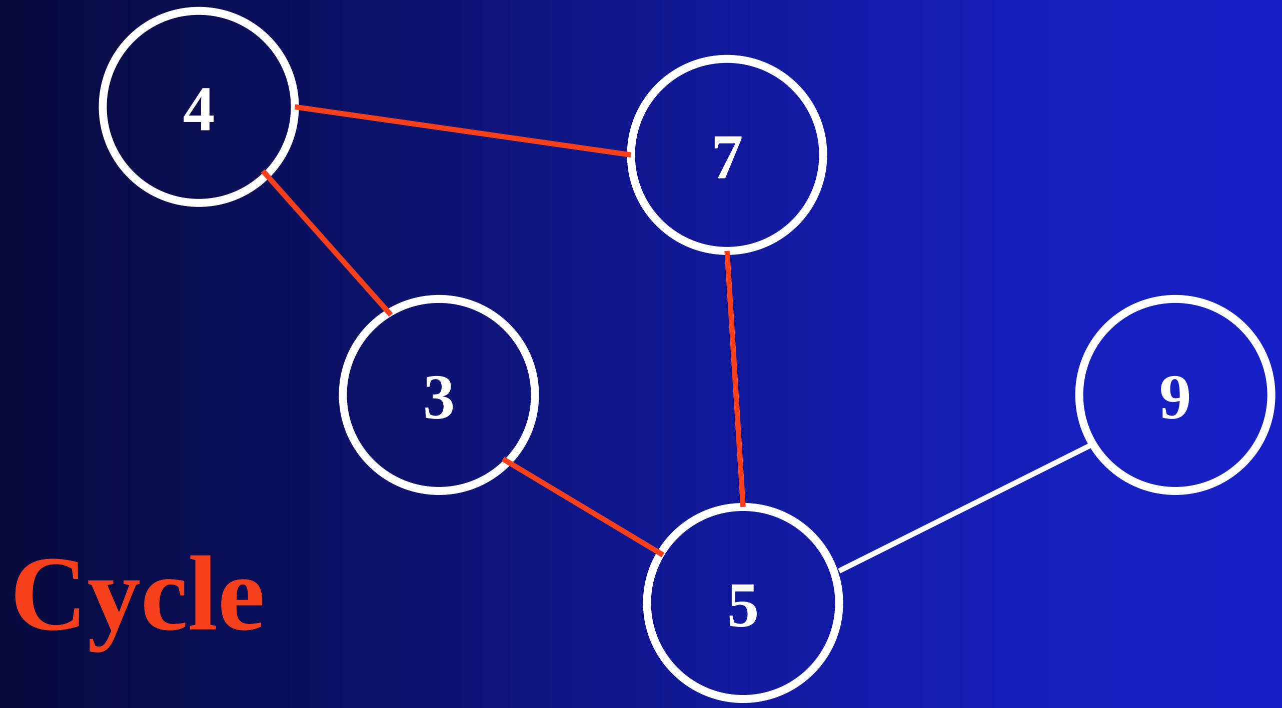


- For a Set of *n* Nodes:
  - *n-1* Edges for each Node

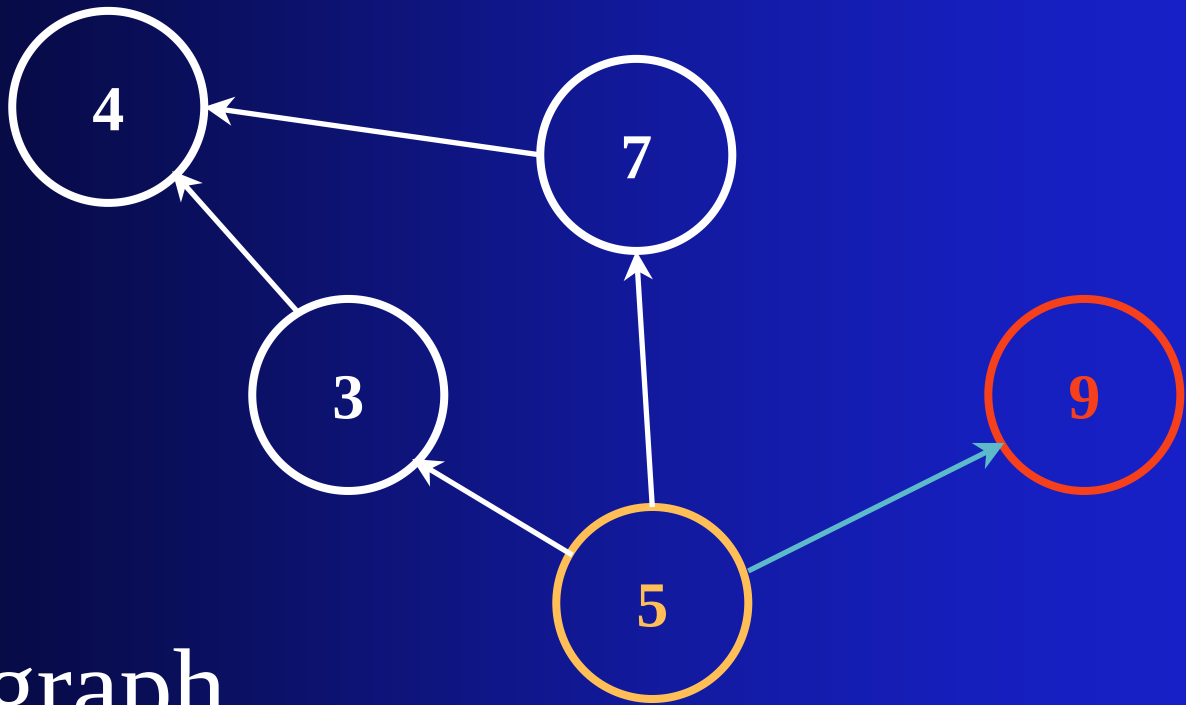
Connected Component



Cycles

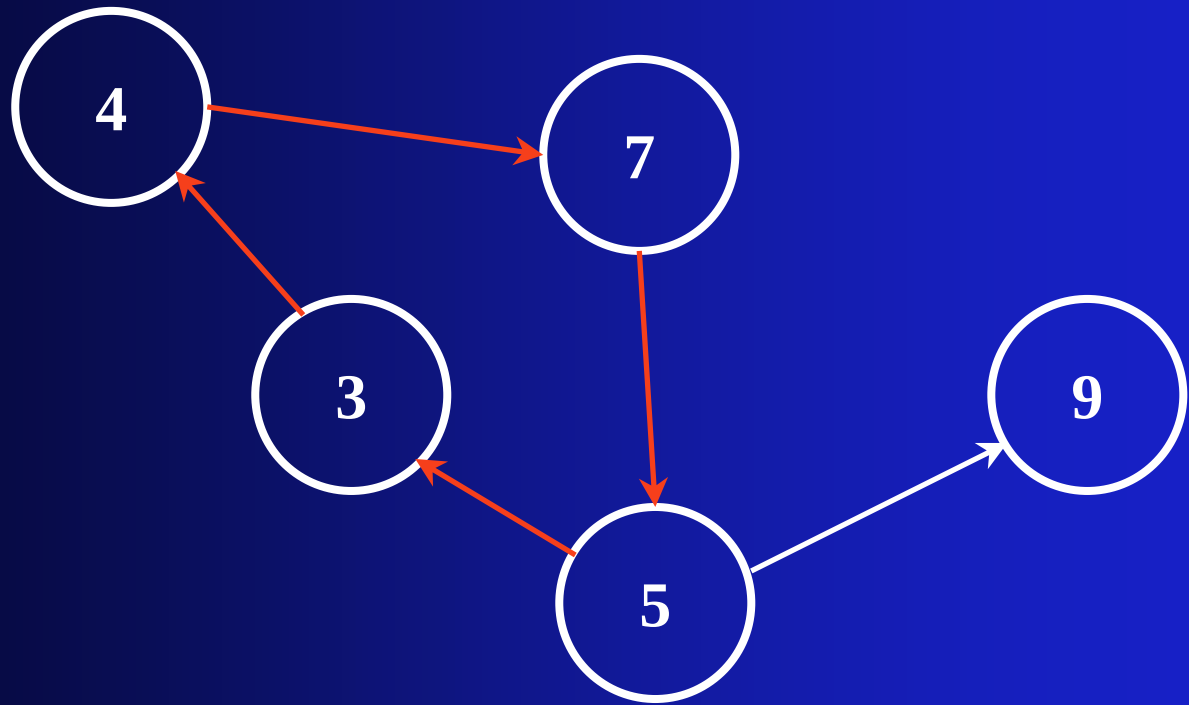


Directed Graph

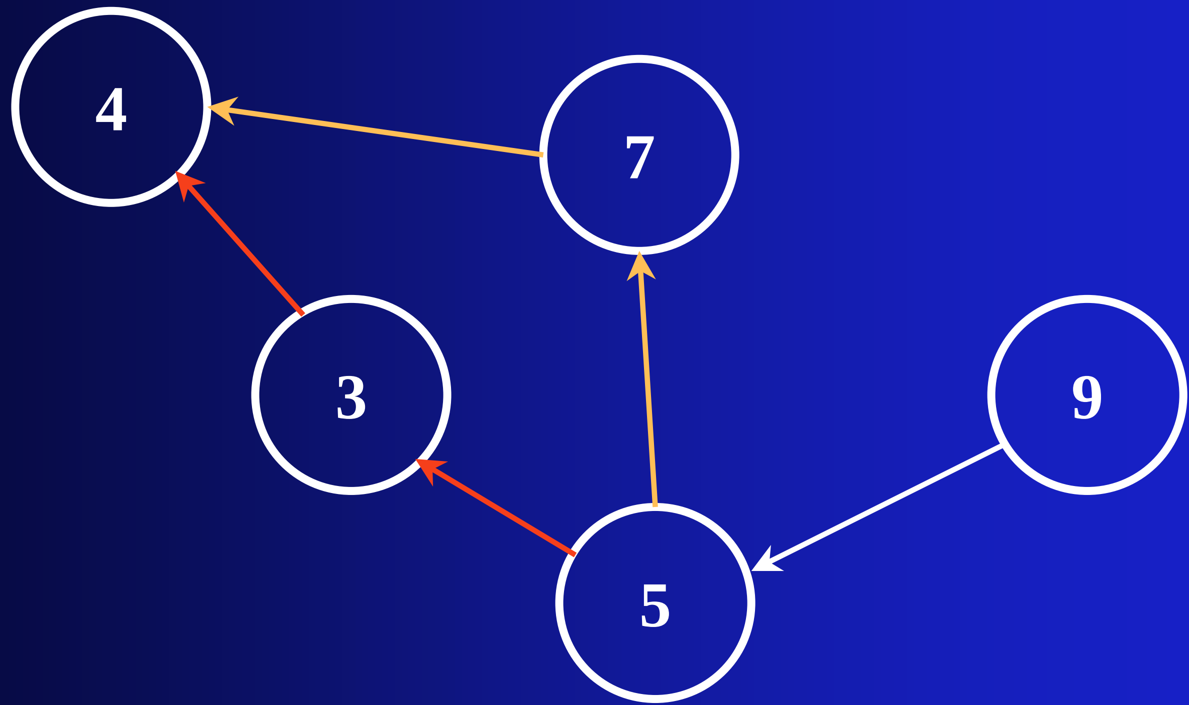


- Digraph
  - Contains directed Edges between Sources and Destinations

Cyclic Digraph

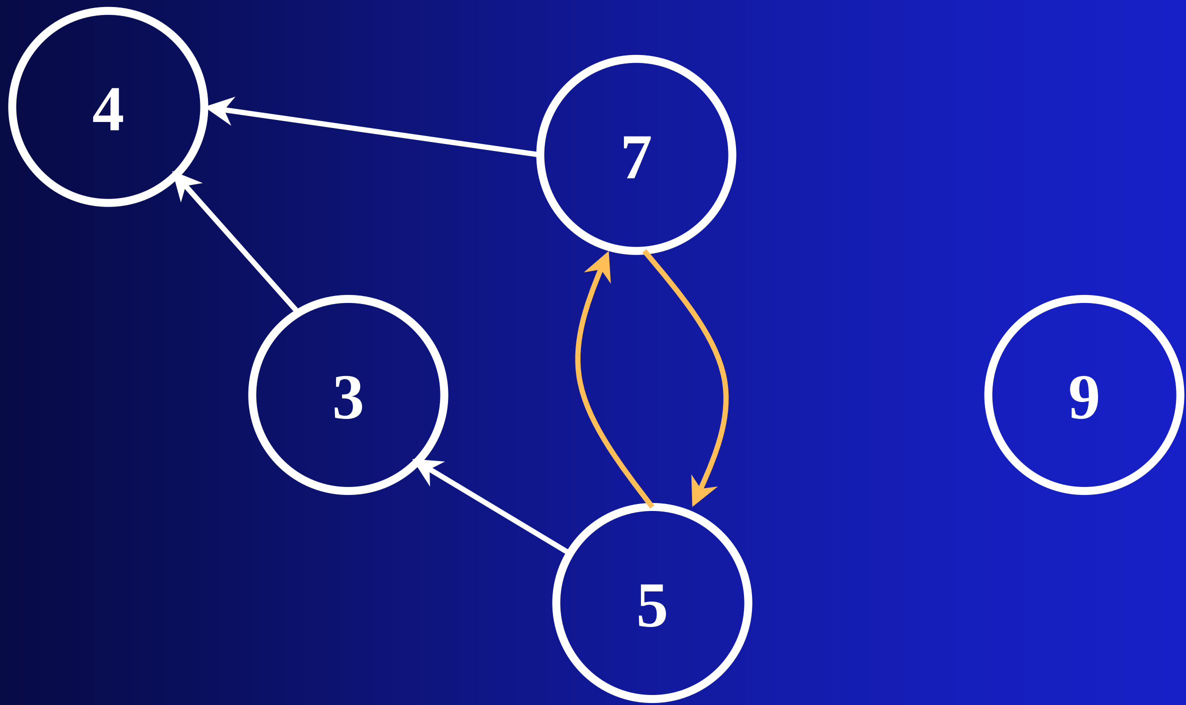


Directed Acyclic Graph



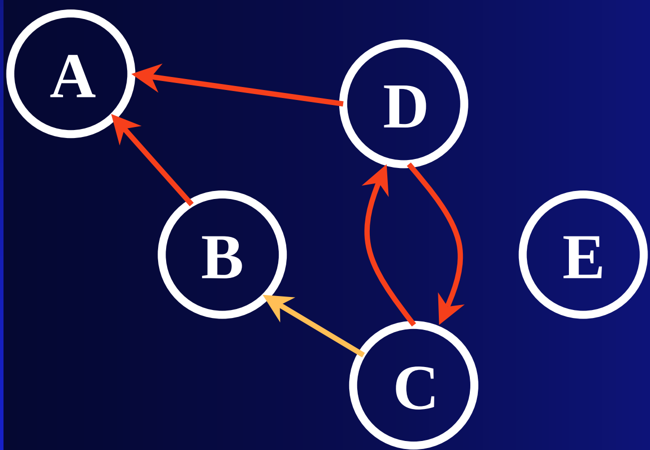
- DAG

Bidirectional Connector



Graph Representations

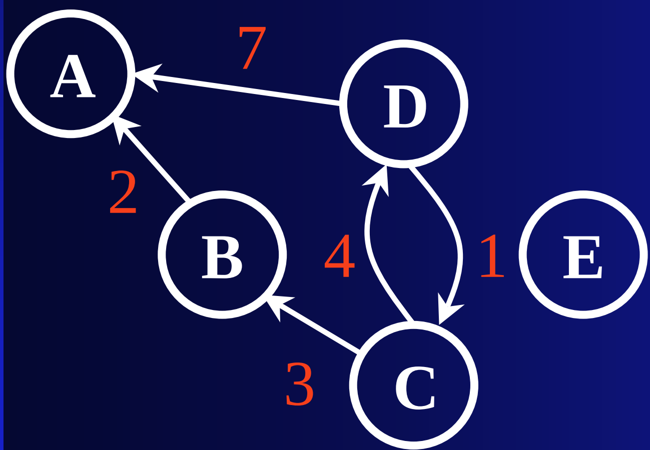
- Adjacency Matrix
  - E.g., in a two-dimensional array of booleans
  - A cell [i, j] contains a ‘1’ value if there is an edge from vertex i to vertex j,‘0’ otherwise



	j	0	1	2	3	4
i		A	B	C	D	E
0	A					
1	B	1				
2	C		1		1	
3	D	1		1		
4	E					

Graph Representations (2)

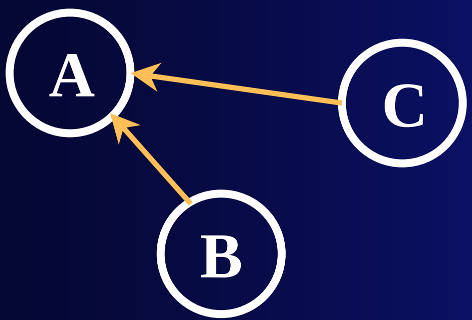
- Weighted Graphs
  - Store weight instead of just 1 (true) or 0 (false)



	j	0	1	2	3	4
i		A	B	C	D	E
0	A					
1	B	2				
2	C		3		4	
3	D	7		1		
4	E					

Graph Representations (3)

- Adjacency List
  - Store the information about a graph in an array of linked lists
  - The i^{th} linked list contains all Vertices that receive an Edge from Vertex i



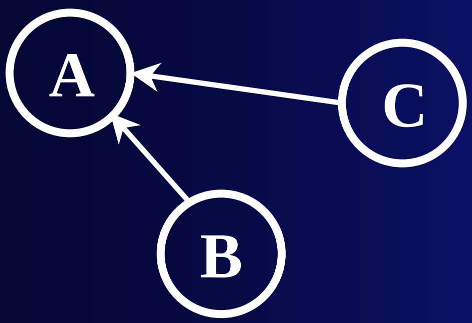
i: 0 A → NULL

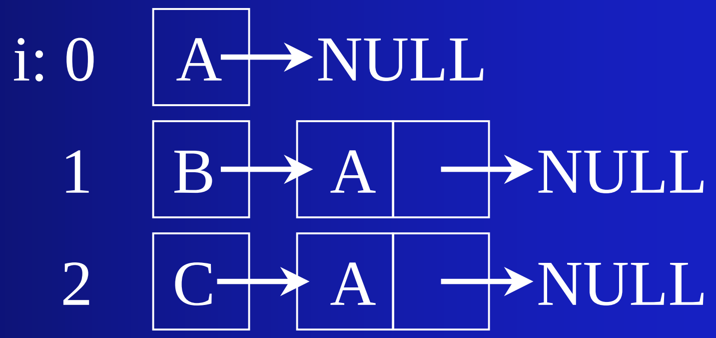
1 B → A → NULL

2 C → A → NULL

Graph Representations (4)

- Adjacency List (2)
  - Possible Edge weights may be included in the Nodes of the List
  - Space Efficiency: good for sparse graphs, i.e. graphs without many edges





Complexity Analysis

- Check existing Edge between any two given vertices v_1 and v_2
  - Matrix: index operation **O(1)**
  - List: follow links **O(n)**
- Find all v_i adjacent to given v_1
  - Matrix: always visit all N cells
  - List: simply visit the list for the given vertex (less than N for sparse graphs)

Complexity Analysis (2)

- Iterate across all neighbours of v_1
  - Number of edges in a complete graph with N vertices
    - Directed: N * (N-1)
    - Undirected: N * (N-1) / 2
  - Matrix: worst case **$O(n^2)$**
  - List: depends on number of neighbours
    - sparse graphs: **$O(n)$**
    - dense graphs: **$O(n^2)$**

Traversals

- Remember Tree traversals:
  - Start at top, visit all nodes
- Graph:
  - Start from a given vertex, visit all vertices to which it connects
- Complexity
  - Matrix: iterate across the row: **O(n)**
  - List: traverse the vertex's linked list: **O(n)**

Traversal Example

```
void traverseFromVertex(Graph *G, Vertex *start)
{
  mark_unvisited(G);    // all vertices:  O(n)
  insert start in empty collection  //    O(1)
  for each vertex in collection {   //    O(n)
    if (!vertex.visited()) {        //    O(1)
      vertex.setVisited();          //    O(1)
      do_something(vertex);         //    O(?)
      collection.add(vertex.adjacent());//O(n)
    }
  }
}
```

Traversal Types

- Depth First (DFT)
  - Go deeply into the graph before backtracking on another path
  - ➢ Use a Stack as the collection
  - ➢ Use recursion
- Breadth First (BFT)
  - Visit each adjacent vertex first
  - ➢ Use a Queue as the collection

Recursive Depth-First Example

```
void traverseFromVertex(Graph *G, Vertex *start)
{
  mark_unvisited(G);    // all vertices:  O(n)
  depth_first(G, start);
}

void depth_first(Graph *G, Vertex *v)
{
  v.setVisited();
  do_something(v);
  for each w in vertex.adjacent()
    if (!w.visited())
      depth_first(G, w);
}
```

Trees within Graphs

- Traversal from a vertex
  - Only includes a sub-graph of the main graph
  - E.g., a depth-first traversal creates a depth-first search tree
- Spanning Tree
  - A sub-graph containing the minimum number of edges possible while retaining the connection between all the vertices in the sub-graph

Minimum Spanning Tree

- Minimum Spanning Tree
  - Traversal using a minimum number of edges
  - For weighted edges: minimising the sum of edge's weights
- (Minimum) Spanning Forest
  - Repeatedly apply the (minimum) spanning tree on all graph components

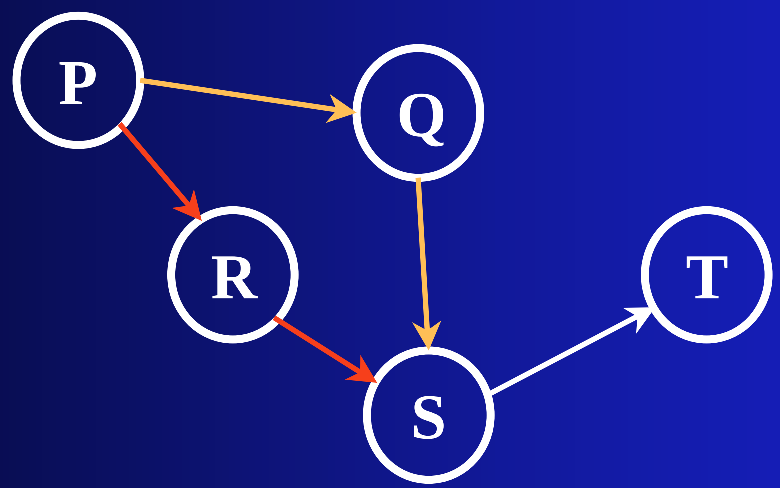
Minimum Spanning Tree (2)

```
void minimumSpanningTree(Graph *G)
{
  mark_unvisited(G);     // all vertices:  O(n)
  mark some vertex v as visited
  for (k = 1; k < n; k++)     // for each vertex
  {
    find the smallest weight from a visited
         vertex to an unvisited vertex w
    mark the edge and w as visited
  }
}
```

- **Complexity:** `O(n*m)`

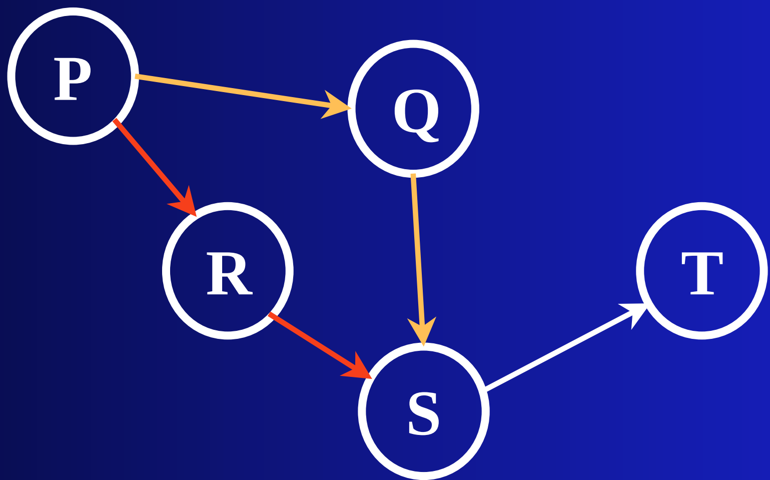
Topological Orders

- DAGs may have certain orderings among the vertices
  - Topological Orders



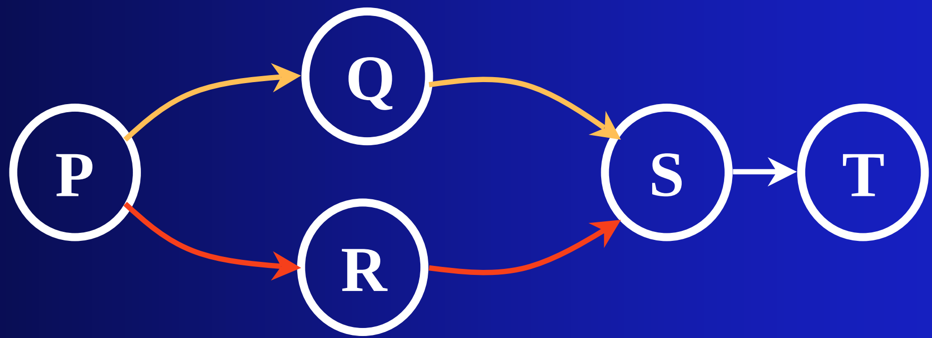
Topological Sort

- Find a topological order of vertices using a traversal (DFT, BFT)



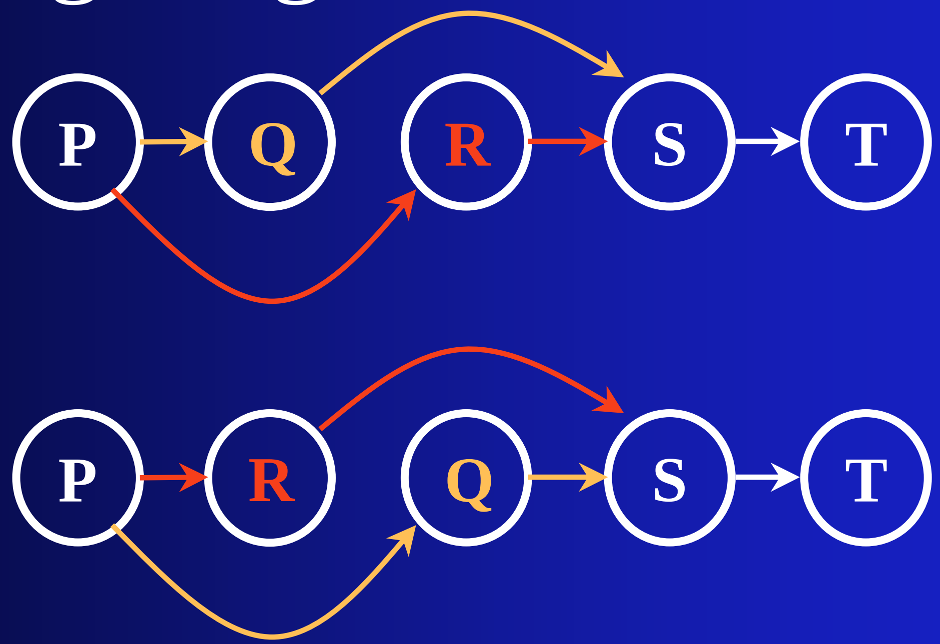
Topological Sort (2)

- Find a topological order of vertices using a traversal (DFT, BFT)



Topological Sort (3)

- Multiple possible (equivalent) orderings, e.g.



Shortest Path Problems

- Single-Source Shortest Path
  - Shortest Poth from a given vertex to all other vertices
  - ➢Dijkstra’s algorithm: **$O(n^2)$**
- All Pairs Shortest Path
  - Set of all the shortest paths in a graph
  - ➢Floyd’s algorithm: **$O(n^3)$**

Dijkstra's Algorithm

- Inputs
  - DAG with edge-weights > 0
  - Single source vertex **s**
- Output: two-dimensional array:
  - N rows: Vertices
  - Three columns: Vertex #, Distance from source, Predecessor
  - Temporary array of booleans: vertex included in path
- Two steps
  - Initialisation and Computation

Initialisation

```
for each vertex v in the graph (each row in results) {
  vertexnumber[row] = v;
  if v == source vertex s {                  // source node
    distance[row] = 0;
    path[row] = undefined;
    included[row] = true; }
  else if there is an edge from s to v {// nodes
    distance[row] = edge_weight(s,v);    // adjacent
    path[row] = s;                          // to source
    included[row] = false; }
  else {
    distance[row] = infinite;               // all other
    path[row] = undefined;                  // nodes
    included[row] = false; }
}
```

Initialisation Results

- `included[]`
  - All cells are *false* except for source vertex cell
- `distance[]`
  - `== 0` (source vertex)
  - `> 0` (adjacent vertices)
  - **infinity** (all other vertices)
- `path[]`
  - Source vertex (adjacent vertices) or undefined

Computation

```
do {
  find vertex F that is not yet included and has
       minimal difference
  included[F] = true;
  for each other vertex T not included {
    if there is an edge from F to T {
      newdist = distance[F] + edge_weight(F,T);
      if newdist < distance[T] {
        distance[T] = newdist;
        path[T] = F;
      }
    }
} while not all vertices are included;
```

Shortest Path Complexity

- Critical Step
  - Nested if statement
    - Resets distance and predecessor for an unicnluded vertex if a new minimal distance has been found
- Initialisation: every vertex $O(n)$
- Computation: nested loops $O(n^2)$
- Total: $O(n^2)$

Graph Interface

- Needs to define
  - Mutators: Adding/removing edges and vertices
  - Accessors: checking/returning edges/weights
  - Iterators
    - Over vertices, labels, adjacent vertices
    - Over edges, edges connected to a specific vertex
  - Other interfaces
    - Getting/Setting labels, weights, etc.

Array Implementation

```
class Graph {
   char **vertex;  // array of vertices
   int *edge;      // array of edges with weight
   int n, size;    // number of vertices

public:
   Graph(int N)
   {               // in real life: check errors!
       vertex = (char **)malloc(N*sizeof(char *));
       edge   = (int *)malloc(N*N * sizeof int);
       size   = N;
       n      = 0; // no actual vertices yet
   }
}
```

Adding a Vertex/Edge

```
void Graph::addVertex(char *label)
{
  /* remove all edges pointing to new vertex */
  memset(edge + n * size, 0,
         size * sizeof (int *));

  vertex[n++] = label;
}

void Graph::addEdge(int from, int to, int w)
{
  edge[from + size * to] = w;
}
```

Retrieving Data

```
int Graph::getEdge(int from, int to)
{ /* return weight of edge (0 if no edge) */
  return edge[from + size * to];
}

char *Graph::findVertex(char *label)
{
  int i;
  for (i = 0; i < n; i++)
    if (strcmp(vertex[i], label) == 0)
      return vertex[i];
  return NULL;
}
```

Other Functions/Methods

- **`deleteVertex()`**
  - Remove vertex from array
  - Remove Gap!
- **`deleteEdge()`**
  - Same as **`addEdge(from, to, 0);`**
- **`numVertices()`** return **`n;`**
- **`numEdges()`**
  - number of edges with weight **`> 0`**

Unordered Collections

- Items in no particular position
- Set
  - Unique items in no particular order
- Counted Set (Multi Set, Bag)
  - Items in no particular order
  - Same item can be present multiple times
- Dictionary (Map)
  - Values associated with unique keys

Implementations

- Standard Template Library (STL)
  - Unique Set:
    - `set`
  - Bag:
    - `multiset`
  - Dictionary:
    - `map`
    - `multimap`

Hash Tables Revisited

- Each Item has a unique hash value
  - Used as index into an array
- Hash value
  - Is computed in constant time
  - Computed by a hash function
- Makes insertion, access, and removal **O(1)**
  - Used to implement Dictionaries as Arrays

References

- Lambert, K. A., & Osborne, M. (2004): *A Framework for Program Design and Data Structures*: Brooks/Cole.
  - Chapter 13-14

- http://en.wikipedia.org/wiki/Graph_%28mathematics%29
- http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/GraphTheoryIII.pdf
- http://students.ceid.upatras.gr/~papagel/project/contents.htm

References

- Lambert, K. A., & Osborne, M. (2004): *A Framework for Program Design and Data Structures*: Brooks/Cole. Chapter 13-14
- Scott Meyers. Effective STL. Addison-Wesley, 2001.
- Matthew H. Austern. Generic Programming and the STL. Addison-Wesley, 1999.
- Bjarne Stroupstrup. The C++ Programming Language, 3rd Edition. Addison-Wesley, 1997.
- http://www.sgi.com/tech/stl
- http://www.cppreference.com/cppstl.html

Appendix

Plain C Examples

Array Graph Implementation

```
typedef struct arrayGraph {
  char **vertex;    // array of vertices
  int *edge; // array of edges with weight
  int n, size;      // number of vertices
} Graph;

Graph *newGraph(int N)
{                   // in real life: check errors!
  Graph *graph      = malloc(sizeof Graph);
  graph->vertex     = malloc(N * sizeof(char *));
  graph->edge = malloc(N*N * sizeof(int));
  graph->size = N;
  graph->n          = 0;   // no actual vertices yet
  return graph;
}
```

Adding a Vertex/Edge

```
void addVertex(Graph *g, char *label)
{
  /* remove all edges pointing to new vertex */
  memset(g->edge + g->size * g->n, 0,
         g->size * sizeof (int *));

  g->vertex[g->n++] = label;
}

void addEdge(Graph *g, int from, int to, int w)
{
  g->edge[from + g->size * to] = w;
}
```

Retrieving Data

```
int getEdge(Graph *g, int from, int to)
{ /* return weight of edge (0 if no edge) */
  return g->edge[from + g->size * to];
}

char *findVertex(Graph *g, char *label)
{
  int i;
  for (i = 0; i < g->n; i++)
    if (strcmp(g->vertex[i], label) == 0)
      return g->vertex[i];
  return NULL;
}
```

Other Functions/Methods

- **`deleteVertex()`**
  - Remove vertex from array
  - Remove Gap!
- **`deleteEdge()`**
  - Same as `addEdge(g, from, to, 0);`
- **`numVertices()`** return `g->n;`
- **`numEdges()`**
  - number of edges with weight `> 0`

Appendix

Objective-C Examples

Array Graph Interface

```
@interface Graph: NSObject {
   id *vertex;      // array of vertices
   int *edge;       // array of edges with weight
   int n, size;     // number of vertices
}

- initWithSize:(int) N;
- (void) addVertex: label;
- (void) addEdgeFrom: (int) src  to: (int) dst
              weight: w;
- (int) getEdgeFrom: (int) src  to: (int) dst;
- findVertex: label;

@end
```

Array Graph Interface

```
@implementation Graph

- initWithSize:(int) N
{       // in real life: add error checking!
        [super init];
        vertex = malloc(N * sizeof(id));
        edge   = malloc(N*N * sizeof(int));
        size   = N;
        n      = 0; // no actual vertices yet

        return self;
}
```

Adding a Vertex/Edge

```
- (void) addVertex: label
{
  /* remove all edges pointing to new vertex */
  memset(edge + size * n, 0,
         size * sizeof (int *));

  vertex[n++] = label;
}

- (void) addEdgeFrom: (int) src  to: (int) dst
              weight: (int) w
{
   edge[src + size * dst] = w;
}
```

Retrieving Data

```
- (int) getEdgeFrom: (int) src   to: (int) dst
{ // return weight of edge (0 means no edge)
  return edge[src + size * dst];
}

- findVertex: label
{
  int i;

  for (i = 0; i < n; i++)
    if ([vertex[i] isEqual: label])
      return vertex[i];

  return nil;
}
```

Other Functions/Methods

- **`-deleteVertex:`**
  - Remove vertex from array
  - Remove Gap!
- **`-deleteEdgeFrom:To:`**
  - Same as **`addEdgeFrom:To:Weight:0;`**
- **`-numVertices`**
  - return **`n`**;
- **`-numEdges`**
  - number of edges with weight **`> 0`**