Graphs

2501ICT/7421ICTNathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

1. Introduction to Graphs
   - Overview
   - Basic Graph Definitions
   - Directed Graphs
2. Graph Algorithms and Implementations
   - Graph Representations
   - Graph Algorithms and Implementations
3. Unordered Collections
   - Sets and Maps

Graphs

Graphs and Unordered Collections

Overview

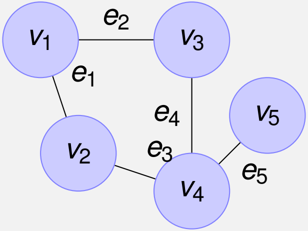
- Definition
- Graph Representations
- Basic Operations
  - Graph Traversals
  - Topological Sort
  - Trees within Graphs
- Unordered Collections

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)

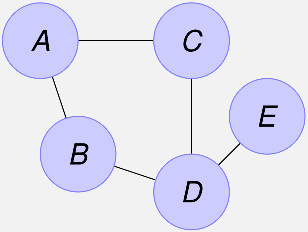
Graph Layout

- Sets of
  - → V: Vertices (Nodes)
  - → E: Edges
    - Each Edge $e \in E$ connects two Nodes $v \in V$



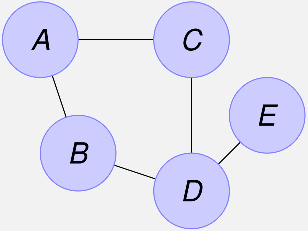
Unlabelled Graphs

- No labels for
  - → Vertices
  - → Edges



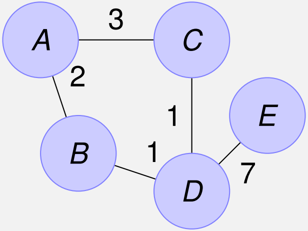
Labelled Vertices

- Only Vertices have names



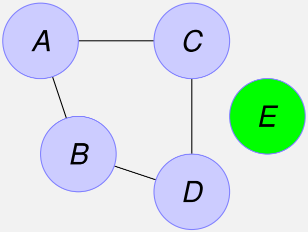
Weighted Graphs

- Labelled Vertices and Edges
- Edge labels typically are numbers
    - interpreted as the weight
    - → cost of going from one vertex to the next
    - → if no edge weight is given, 1 is assumed



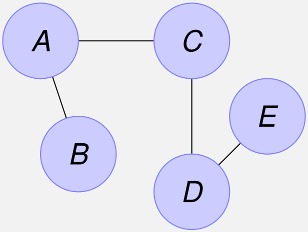
Disconnected Graphs

- One or more Nodes are not connected



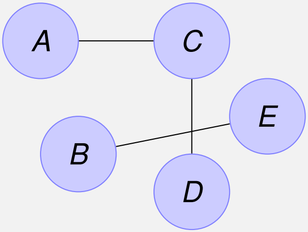
Connected Graphs

→ at least one *path* exists from each to every other vertex



Connected Components

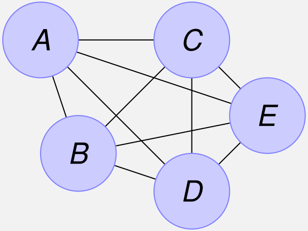
- the whole graph may not be connected, but it consists of connected subgraphs (components)



Complete Graphs

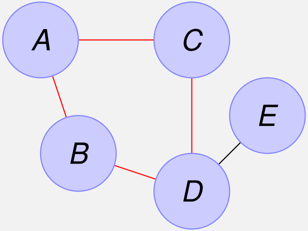
→ All possible connections exist

- For a Set of n Nodes:
  - $n - 1$ edges for each node



Cycles

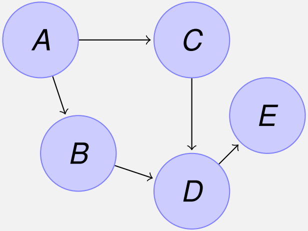
- a cycle is the possibility of following a path from a vertex back to itself without ever following the same edge more than once



Directed Graphs

→ Digraph

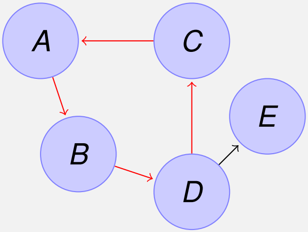
- contains *directed edges* between *sources* and *destinations*



Directed Cyclic Graphs

→ Cyclic Digraph

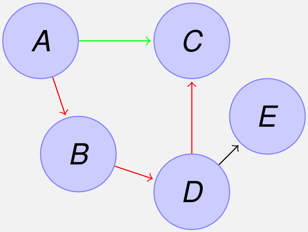
- contains at least one cyclic path along directed edges



Directed Acyclic Graphs

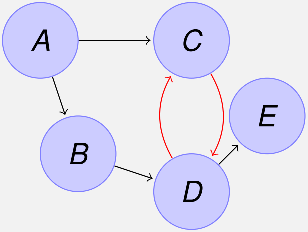
→ DAG
- directed graph that contains no cycles

→ many graph algorithms require DAGs



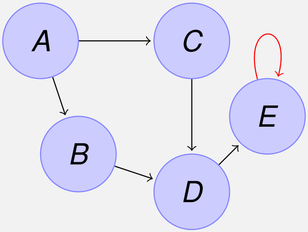
Bidirectional Connectors

- undirected edges can be modelled by two directed edges
  - → the two directions may have different weight (more flexible than a weighted undirected edge)!



Looping Edges

- an edge may loop from a node back to itself
  - → used in automata and state machines

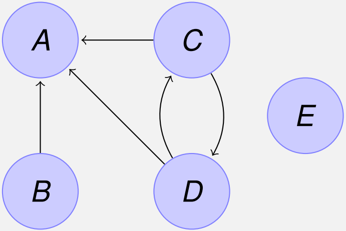


Graph Algorithms and Implementations

Graph Algorithms and Implementations

Graph Representations

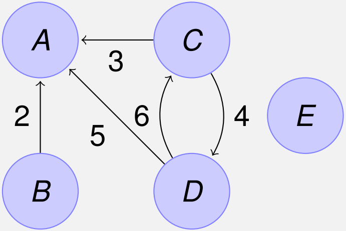
- Adjacency Matrix
  → a two-dimensional array of numbers
  - a cell `[i,j]` contains 1 if there is an edge from vertex i to vertex j, `0` otherwise (zeroes are not shown in the table below)



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

Graph Representations (2)

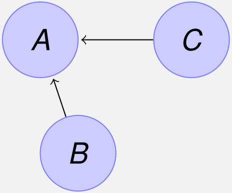
- Weighted Graphs
  → store weight instead of just 1 (true) or 0 (false)



i \ j		0 A	1 B	2 C	3 D	4 E
0	A					
1	B	2				
2	C	3			4	
3	D	5		6		
4	E					

Graph Representations (3)

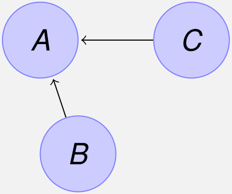
- Adjacency List
  - store the information about a graph in an array of linked lists
  - the i^{th} linked list contains all Vertices that receive an Edge from Vertex i



i: 0	A	→ `nil`		
1	B	→	A	→ `nil`
2	C	→	A	→ `nil`

Graph Representations (4)

- Adjacency List
  - edge weights may be included in the nodes of the list
  - space efficiency: good for sparse graphs, i.e. graphs without many edges



i: 0	A	$\rightarrow$ `nil`		
1	B	$\rightarrow$	A	$\rightarrow$ `nil`
2	C	$\rightarrow$	A	$\rightarrow$ `nil`

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_k
  - Matrix: always visit all N cells — O(n)
  - List: list for the given vertex — O(n)
    - → small number for sparse graphs — O(1)

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

Example (pseudo code)

```
void traverseFromVertex(Graph *G, Vertex *startNode)
{
  mark_unvisited(G);                    // all vertices:        O(n)
  insert startNode 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(1)
      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

Example (pseudo code)

```
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

  → a depth-first traversal creates a depth-first search tree
- Spanning Tree
  - a sub-graph starting at a given vertex and 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 the edges' weights
- (Minimum) Spanning Forest
  - repeatedly apply the (minimum) spanning tree algorithm on all graph components

Minimum Spanning Tree Algorithm

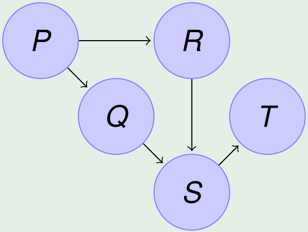
Example (pseudo code)

```
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;
  }
}
```

$\Rightarrow$ Complexity: $O(n \cdot m)$

Topological Orders

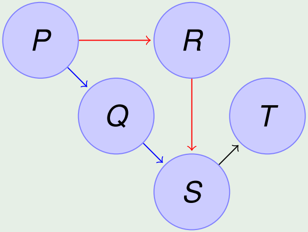
Example (Graph Example)



- DAGs may have certain orderings among the vertices
  $\rightarrow$ Topological Orders

Topological Sort

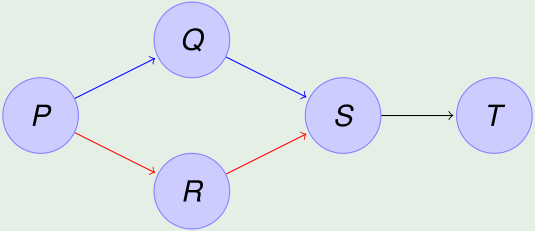
Example (Graph Example)



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

Topological Sort (2)

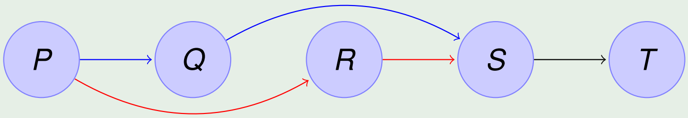
Example (attempt to flatten the graph)



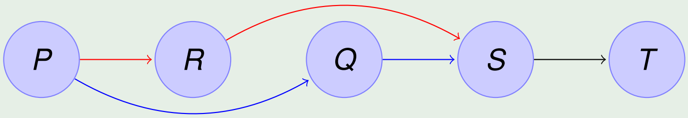
- Find a topological order of vertices using a traversal (DFT, BFT)

Topological Sort (3)

Example (one topological order)



Example (another topological order)



- Multiple equivalent orderings are possible

Shortest Path Problems

- Single-Source Shortest Path
  - shortest Path 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 greater than 0
  - a single source vertex *s*
- Output: two-dimensional array:
  - *N* rows: vertices
  - three columns
    1. vertex number
    2. distance from source
    3. predecessor
  - temporary array of booleans: vertex included in path
- Two steps
  - initialisation
  - computation

Initialisation

Dijkstra's Algorithm: 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 adjacent to source
  {
    distance[row] = edge_weight(s,v);
    path[row] = s;
    included[row] = false;
  }
  else                                  // all other nodes
  {
    distance[row] = infinite;
    path[row] = undefined;
    included[row] = false;
  }
}
```

Initialisation Results

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

Computation

Dijkstra's Algorithm: 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 unincluded 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 Implementations

A Graph Class

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 of labels, weights, etc.

Array Graph Interface

Example (Objective-C)

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

- initWithSize:  (int) N;

- (void) addVertex:  label;

- (void) addEdgeFrom:  (int) src
                 to:  (int) dst
             weight:  w;

- (int) edgeFrom:  (int) src
             to:  (int) dst;

- (int) findVertex:  label;

@end
```

Example (C++)

```
class Graph
{
  string *vertex;// vertices array
  int *edge;     // edge array
  int n, size;   // # of vertices

public:
  Graph(int N);

  void Graph::addVertex(const string &label);

  void Graph::addEdge(int from,
                      int to,
                      int w);

  int Graph::getEdge(int from,
                     int to);

  int Graph::findVertex(const string &label);

};
```

Array Graph Implementation

Example (Initialiser)

```
@implementation Graph

- initWithSize:  (int) N
{ // in real life:  check errors!
  if (![super init]) return nil;
  vertex = calloc(N, sizeof(id));
  edge = calloc(N*N, sizeof(int));
  size = N;
  n = 0; // no vertices yet

  return self;
}
```

Example (Constructor)

```
/** Graph Implementation:  Constructor */

Graph::Graph(int N)
{ // in real life:  check errors!

  vertex = new string[N];
  edge = new int[N*N];
  size = N;
  n = 0; // no vertices yet


}
```

Adding a Vertex/Edge

Example (Objective-C)

```
- (void) addVertex:  label
{
  vertex[n++] = label;
}


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

Example (C++)

```
void Graph::addVertex(const string &label)
{
  vertex[n++] = label;
}


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

Retrieving Data

Example (Objective-C)

```
/*
 * return weight of edge
 */
- (int) edgeFrom: (int) src
              to: (int) dst
{
  return edge[src + size * dst];
}

/*
 * find vertex with label
 */
- (int) findVertex: label
{
  for (int i = 0; i < n; i++)
    if ([vertex[i] isEqual: label])
      return i;

  return -1;  // not found
}
```

Example (C++)

```
/*
 * return weight of edge
 */
int Graph::getEdge(int src,
                   int dst)
{
  return edge[src + size * dst];
}

/*
 * find vertex with label
 */
int Graph::findVertex(const string &label)
{
  for (int i = 0; i < n; i++)
    if (vertex[i] == label)
      return i;

  return -1;  // not found
}
```

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

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

Unordered Collection Class Examples

- Unordered Set
  - `NSSet` / `NSMutableSet` — Objective-C
  - `std::unordered_set` — C++
- Ordered Set
  - `NSOrderedSet` / `NSMutableOrderedSet` — Objective-C
  - `std::set` — C++
- Counted Set (Multi Set, Bag)
  - `NSCountedSet` — Objective-C
  - `std::multiset` — C++
- Dictionary (Map)
  - `NSDictionary` / `NSMutableDictionary` — Objective-C
  - `std::map` / `std::multimap` — C++

NSDictionary Example

Example (an English/German dictionary in Objective-C)

```
#import <Foundation/Foundation.h>

int main(int argc, char *argv[])
{
    @autoreleasepool
    {
        NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
        /*
         *      @"German:",     @"English:",
         */
                @"Eins",        @"One",
                @"Zwei",        @"Two",
                @"Drei",        @"Three",
                nil];

        id key = @"Three";                          // a key to search for
        id value = [dict objectForKey:  key];     // its corresponding value

        printf("The German translation for '%s' is '%s'\n",
                        [key UTF8String], [value UTF8String]);
    }

    return EXIT_SUCCESS;
}
```

std::map Example

Example (an English/German dictionary in C++)

```
#include <iostream>
#include <map>

using namespace std;

int main(int argc, char *argv[])
{
        map<const char*, const char *> dict;

        /*
         *   English      German
         */
        dict["one"]     = "eins";
        dict["two"]     = "zwei";
        dict["three"]   = "drei";

        const char *key = "three";              // a key to search for
        const char *value = dict[key];          // its corresponding value

        cout << "The German translation for ";
        cout << key << " is "<< value << endl;

        return EXIT_SUCCESS;
}
```