Introduction to Graph Algorithms
COMP4500/7500
Advanced Algorithms & Data Structures

Overview

- What have we done so far and why?
- Introduction to graphs
- Representing graphs
- Breadth-first search
- Depth-first search

What have we done and why?

Goal: **To be able to efficiently solve complex problems**
First: we need to be able to **analyse the efficiency of algorithms**.

- There are different measures of efficiency, e.g. time, space.
- How efficient an algorithm is depends on its *input size* and *input value*.
- We can describe the *best*, *average* or *worst case* as a function of input size.
- Functions can be described and compared using *asymptotic notation*.
  - Θ: asymptotic tight bounds
  - O: asymptotic upper bounds
  - Ω: asymptotic lower bounds

Quick question

Which asymptotic notation (e.g. O, Ω, Θ) do we use to describe

- worst case time complexity?
- best case time complexity?
- average case time complexity?

What have we done and why?

Non-trivial algorithms use loops and/or recursion

- Loops can be analysed using *summations*

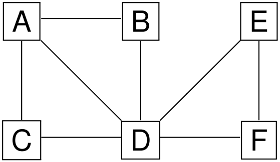
$$T(n) = \sum_{i=0}^{n} i^2$$

- Recursive programs can be analysed using *recurrences*

$$T(n) = 2T(n/2) + n$$

Use mathematical/logical reasoning to convert these into a statement in terms of asymptotic notation: $\Theta(n^2)$.

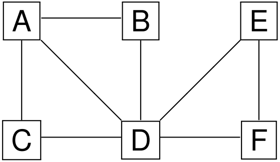
Graphs



Made up of

- Vertices
  A, B, C, D, E, F
- Edges
  (A,B), (A,D), (A,C), (B,D), (C,D), (E, D), (E, F), (D, F)

Graphs

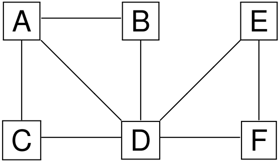


Could be used to represent

- Geographical information (Vertices = cities, Edges = roads)
- Networks (Vertices = computers, Edges = cables)
- Brains (Vertices = neurons, Edges = synapses)
- Programs (Vertices = statements, Edges = control flow)

Graphs: Key property: **directed** or **undirected**

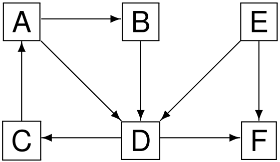
Undirected:



- Cannot have self-loops.
- If $(u, v) \in E(G)$, v is **adjacent** to u in G (symmetric).
- Edge $(u, v) \in E(G)$ is **incident on** vertices v and u
- The **degree** of a vertex v is the number of edges incident upon it.

Graphs: Key property: **directed** or **undirected**

Directed:



- Can have self-loops.
- If $(u, v) \in E(G)$, v is **adjacent** to u in G (asymmetric).
- Edge $(u, v) \in E(G)$ is **incident from** (leaves) vertex u and **incident on** (enters) vertices v.
- The **out-degree** of a vertex v is the number of edges leaving it.
- The **in-degree** of a vertex v is the number of edges entering it.
- The **degree** of a vertex is the sum of its in-degree and out-degree.

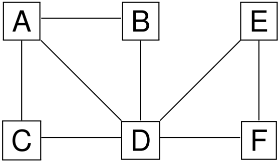
Quick question

What (exactly) is the maximum number of edges in an:

- directed graph with n vertices? n^2
- undirected graph with n vertices? $\sum_{i=0}^{n-1} i \; = n(n-1)/2$

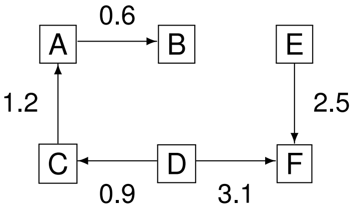
Graphs: Key property: **weighted** or **unweighted**

Unweighted:



Graphs: Key property: **weighted** or **unweighted**

Weighted:



Graphs: terminology

For a graph $G = (V, E)$:

- A **path** of length k from a vertex v_0 to a vertex v_k is a sequence $\langle v_0, v_1, v_2, \ldots, v_k \rangle$ of vertices such that $(v_{i-1}, v_i) \in E$ for $i \in 1, 2, \ldots, k$.
- u is **reachable** from v if there is a path from v to u.
- A path is **simple** if all vertices in the path are distinct.
- A path forms a **cycle** if $v_0 = v_k$ and $k > 1$.
- A cycle is **simple** if $v_1, \ldots, v_k$ are distinct and all of its edges are distinct.
- A graph with no simple cycles is **acyclic**.

Graphs: terminology

An undirected graph $G = (V, E)$ is:

- **connected** if every vertex is reachable from all other vertices.
- a **forest** if it has acyclic.
- a **tree** if it is a forest with only one connected component.

Quick question

(For an undirected graph:)

What are the **minimum** and **maximum** number of edges in a:

- **tree** with n vertices? $(n-1,\ n-1)$
- **forest** with n vertices? $(0,\ n-1)$
- **connected** graph with n vertices? $(n-1,\ n(n-1)/2)$

Graphs: terminology

An directed graph $G = (V, E)$ is:

- **strongly connected** if there every two vertices are reachable from each other.

Graphs: terminology

- Graph $G' = (V', E')$ is a **sub-graph** of $G = (V, E)$ when $V' \subseteq V$ and $E' \subseteq E$.
- Additionally, G' is a **spanning sub-graph** of G if $V' = V$ also holds.
- The sub-graph of $G = (V, E)$ that is **induced by V'** is the graph $G' = (V', E')$ where $E' = \{(u, v) \in E : u \in V' \wedge v \in V'\}$.

Graphs

Information about graphs we may need:

- Shortest *path* from vertex v to vertex u
- Shortest path from vertex v to every other vertex
- Does the graph contain cycles?
- Is it connected?
- Find a minimum spanning tree?
- Shortest tour of all vertices(?)

Graphs will be used to explore different programming styles throughout the course.

Types of graphs

- Directed Acyclic Graph (DAG)
- Connected graph
- Trees are special types of graphs
- Lists/vectors are also simple graphs

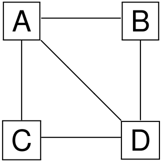
Graph representations

There are two main approaches to representing graphs:

- Adjacency list
- Adjacency matrix

(Representing the set of vertices and edges directly is usually too inefficient.)

Graph representations: adjacency list (undirected)

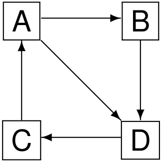


Node	Connections
A	B, D, C
B	A, D
C	A, D
D	A, B, C

For undirected graph $G = (V, E)$ with **V** vertices and **E** edges what is the worst-case:

- **space complexity**?
  $\Theta(V + \sum_{v \in G.V} degree(v)) = \Theta(V + 2E) \in \Theta(V + E)$
- **time complexity of** *isAdjacentTo*(u, v)?
  $\Theta(V)$
- **time complexity to list all adjacent vertex pairs**?
  $\Theta(V + \sum_{v \in G.V} degree(v)) = \Theta(V + 2E) \in \Theta(V + E)$

Graph representations: adjacency list (directed)

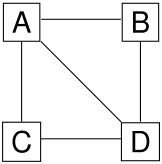


Node	Connections
A	B, D
B	D
C	A
D	C

For directed graph $G = (V, E)$ with **V** vertices and **E** edges what is the worst-case:

- **space complexity**?
  $\Theta(V + \sum_{v \in G.V} outDegree(v)) = \Theta(V + E)$
- **time complexity of** $isAdjacentTo(u, v)$?
  $\Theta(V)$
- **time complexity to list all adjacent vertex pairs**?
  $\Theta(V + \sum_{v \in G.V} outDegree(v)) = \Theta(V + E)$

Graph representations: adjacency matrix (undirected)

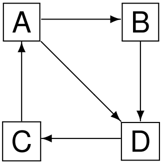


	A	B	C	D
A	-	✓	✓	✓
B	✓	-	-	✓
C	✓	-	-	✓
D	✓	✓	✓	-

For undirected graph $G = (V, E)$ with **V** vertices and **E** edges what is the worst-case:

- **space complexity**?
  $\Theta(V^2)$
- **time complexity of** *isAdjacentTo*(u, v)?
  $\Theta(1)$
- **time complexity to list all adjacent vertex pairs**?
  $\Theta(V^2)$

Graph representations: adjacency matrix (directed)



	A	B	C	D
A	-	✓	-	✓
B	-	-	-	✓
C	✓	-	-	-
D	-	-	✓	-

For undirected graph $G = (V, E)$ with **V** vertices and **E** edges what is the worst-case:

- **space complexity**?
  $\Theta(V^2)$
- **time complexity of** *isAdjacentTo*(u, v)?
  $\Theta(1)$
- **time complexity to list all adjacent vertex pairs**?
  $\Theta(V^2)$

Comparison of graph representations

The **adjacency list** representation is often the most efficient is the graph is **sparse**: few edges relative to the number of vertices.

The **adjacency matrix** representation is often most efficient if the graph is **dense**: many edges relative to the number of vertices.

Graph traversal algorithms

- Breadth-first search
- Depth-first search

Breadth-First Search (BFS)

For an **unweighted graph** G:

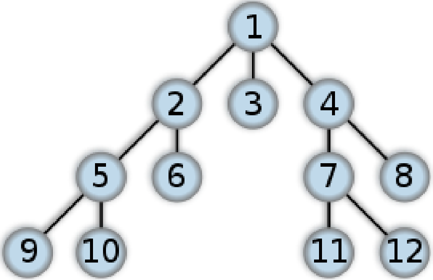
- The **length of a path** is the number of edges in that path. E.g. the length of $\langle v_1, v_2, v_3 \rangle$ is 2.
- The **distance** from vertex u to v is the length of the shortest path from u to v in G.

Breadth-first search (BFS):

- Takes as input a unweighted graph $G = (V, E)$ and a designated *start vertex* $v \in G.V$.
- Traverses vertices in G in order of their **distance** from a v.
- Finds **shortest paths** from v to every other vertex in G.

Breadth-First Search (BFS)

E.g. a possible traversal order of a tree:



Breadth-First Search (BFS): How do we implement it?

What do we know?

1. Start vertex v is at a distance 0 from itself.
2. The vertices adjacent to v, other than those at distance 0, are at distance 1.
3. The vertices adjacent to v's adjacent vertices, other than those at distance 0 or 1, are at distance 2.
4. etc.

Breadth-First Search (BFS): How do we implement it?

We will use a **queue** data structure:

- Vertices are:
  - *enqueued* in order of their distance from the start vertex,
  - *dequeued* in order of their distance from the start vertex.
- The first element added to the queue is the start vertex.
- When a vertex is dequeued, its adjacent vertices that have *not yet been reached* are enqueued.

How do we keep track of which vertices have *been reached*?

Breadth-First Search (BFS): How do we implement it?

Colours are used distinguish vertices at different stages within the algorithm:

- **White**: not reached yet
  (i.e. never enqueued – a shortest path not yet found)
- **Grey**: reached but all adjacent vertices not reached yet
  (i.e. enqueued but not dequeued – a shortest path found)
- **Black**: reached and completed
  (i.e. enqueued and dequeued – a shortest path found)

BFS pseudo-code

Finding a path: keep track of the predecessor of each vertex

BFS(G, v)

```
for u in G.V - {v}
        u.distance = ∞ ; u.colour = white ; u.parent = NULL
v.distance = 0 ; v.colour = grey ; v.parent = NULL
Q.initialise()
Q.enqueue(v)
while not Q.isEmpty()
        current = Q.dequeue()
        for u in G.adjacent[current]
                if u.colour == white
                        u.distance = current.distance + 1
                        u.colour = grey ; u.parent = current
                        Q.enqueue(u)
        current.colour := black
```

BFS: worst-case time complexity?

BFS(G, v)

```
for u in G.V - {v}
        u.distance = ∞ ; u.colour = white
v.distance = 0 ; v.colour = grey
Q.initialise() ; Q.enqueue(v)
while not Q.isEmpty()
        current = Q.dequeue()
        for u in G.adjacent[current]
                if u.colour == white
                        u.distance = current.distance + 1
                        u.colour = grey
                        Q.enqueue(u)
        current.colour := black
```

$$\Theta(V)+\Theta(1)+(\sum_{v\in G.V}\Theta(1)+outDegree(v)\times\Theta(1)) = \Theta(V+E)$$

Depth-first Search (DFS)

Doesn't necessarily find shortest-paths.

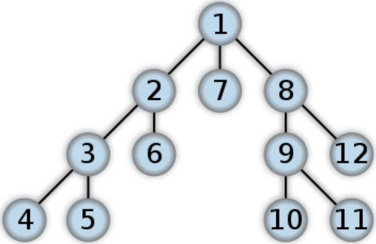
Often used as a subroutine within other algorithms.

A recursive algorithm that visits a vertex v by:

- choosing a unvisited vertex u adjacent to v
- visiting u and all of its unvisited reachable vertices,
- before backtracking to v and continuing until v has no more unvisited adjacent vertices.

Depth-first Search (DFS)

E.g. a possible traversal order of a tree:



DFS pseudo-code

DFS(G)

```
for u in G.V
        u.color = white
for u in G.V
        if u.colour == white then DFS-VISIT(G, u)
```

DFS-VISIT(G, v)

```
v.colour = grey
for u in adjacent[v]
        if u.colour == white
                DFS-VISIT(G, u)
v.colour = black
```

DFS pseudo-code: recording predecessor subgraph

DFS(G)

```
for u in G.V
        u.color = white  ;  u.parent = NULL
for u in G.V
        if u.colour == white then DFS-VISIT(G, u)
```

DFS-VISIT(G, v)

```
v.colour = grey
for u in adjacent[v]
        if u.colour == white
                u.parent = v
                DFS-VISIT(G, u)
v.colour = black
```

DFS: worst case time complexity?

```
DFS(G)
for u in G.V
        u.color = white
for u in G.V
        if u.colour == white then DFS-VISIT(G, u)
```

```
DFS-VISIT(G, v)
v.colour = grey
for u in adjacent[v]
        if u.colour == white
                DFS-VISIT(G, u)
v.colour = black
```

$$\Theta(V) + (\sum_{v \in G.V} \Theta(1) + outDegree(v) \times \Theta(1)) = \Theta(V + E)$$

DFS pseudo-code: with timestamps

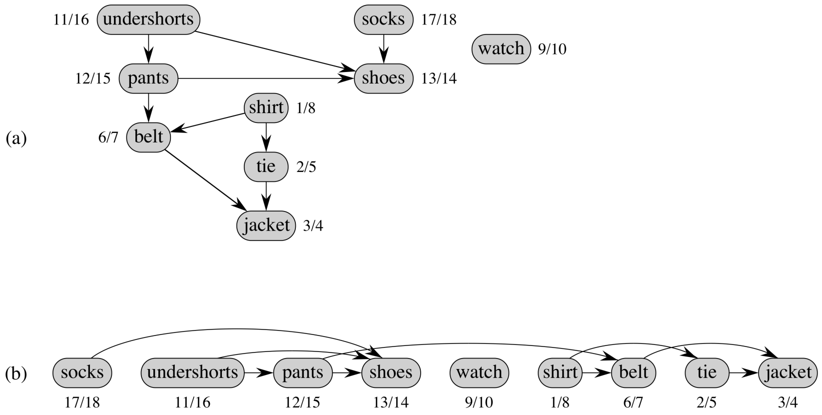
```
DFS(G)
time = 0
for u in G.V
        u.color = white
for u in G.V
        if u.colour == white then DFS-VISIT(G, u)

DFS-VISIT(G, v)
time = time + 1  ;  u.discovered = time
v.colour = grey
for u in adjacent[v]
        if u.colour == white
                DFS-VISIT(G, u)
v.colour = black
time = time + 1  ;  u.finished = time
```

Topological sort

- A **directed acyclic graph (DAG)** is a directed graph with no cycles.
- The directed edges in a DAG can be thought of as **dependencies** between vertices.
- A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of V such that:
  - if $(u, v) \in E$,
    then u comes before v in the topological ordering.

Topological sort



Topological sort

TOPOLOGICAL-SORT(G)

initialise an empty linked-list of vertices
call DFS(G)
as each vertex is finished, insert it onto the front of a linked list
return the list of vertices

i.e. returns vertices in descending order of their DFS finish time.

Recap

- Graphs are a commonly occurring structure in difficult problems
- Can be
  - directed, undirected
  - cyclic, acyclic
  - weighted, unweighted
- Often represented as an adjacency list or adjacency matrix
- Breadth-first search is used for finding shortest-paths
- Depth-first search is used for probing the structure of the graph (e.g., topological sort)