Lecture 7: Graphs

CAB203 Discrete Structures

Matthew McKague

Queensland University of Technology

matthew.mckague@qut.edu.au

Outline

Readings

This week

- Pace: 7.3

Next week

- None

Outline

Graph

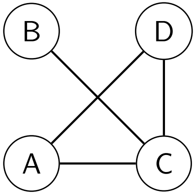
A *graph* is an irreflexive, symmetric relation. Special notation for graphs:

- A graph G is a pair (V, E)
- The set V is the *vertices* or *nodes* of G
- The set $E \subseteq V \times V$ is the *edges* of G
- For every $(u, v) \in E$, we also have $(v, u) \in E$
- There are no edges (v, v)

This type of graph is called a *loop-free undirected graph*, referring to irreflexivity and symmetry. There are other types of graphs as well.

Graph visualisation

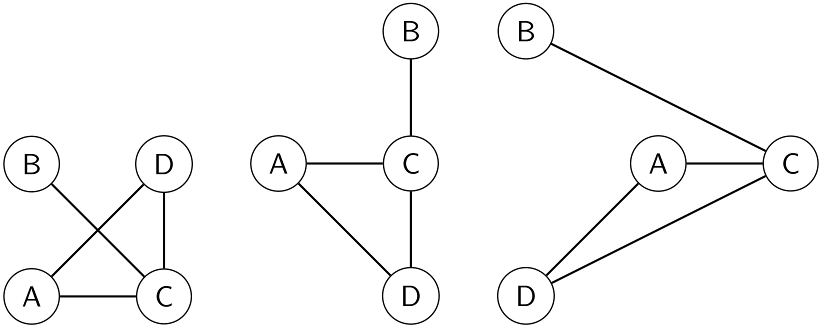
We usually visualise graphs as a set of points (vertices) connected by lines (edges).



The arrangement of the vertices does not matter, and can be chosen arbitrarily.

Some equal graphs

These are all visualisations of the same graph.



Graph definitions

- If $(u, v) \in E$ then we say u and v are *adjacent*. We also say that u and v are *neighbours*
- The *neighbourhood* of a vertex u is the set of u's neighbours;

$$N_G(u) = \{v \in V : (u, v) \in E\}$$

- The neighbourhood of a *set of vertices* $S \subseteq V$ is the set of all neighbours of all vertices in U

$$N_G(S) = \{v \in V : (u, v) \in E, u \in S\}$$

- We may at times be sloppy and use N_V instead of N_G if the set of edges is clear from the context

Neighbourhoods in Python

```
def N(V, E, u):
   return { v for v in V if (u,v) in E }

def NS(V, E, S):
    return { v for v in V for u in S if (u,v) in E }
```

Graph definitions (2)

- If $e = (u, v) \in E$ then we say e is *incident* with u (and v)
- Since for edge $(u, v) \in E$ we also have $(v, u) \in E$, we just call them the same edge, i.e. the edge between u and v. They are also one edge for the purposes of counting.

Graph examples

- The set of all computers (including routers, hubs, etc.) in a network, where u and v are adjacent if they are connected physically (say with Ethernet, fibre, or Wi-Fi)
- The set of cities in the world, where u is adjacent to v if there is a direct flight between u and v
- The set of all road intersections in the world, where u is adjacent to v if they are directly connected by a road

Mapping software use graphs to determine how to get from one place to another. They typically use some variant of the A^* algorithm.

More examples

- The set of all programs and resources in a computer, where program u is adjacent to resource v if u needs v to run
- The set of all people in Australia, where u and v are adjacent if they are married

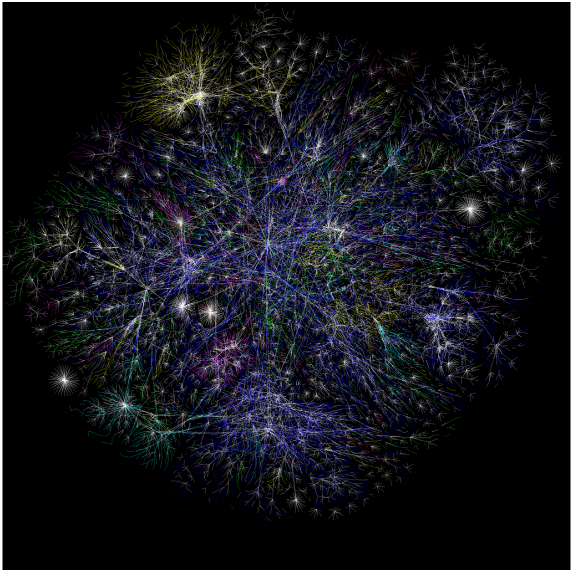
These are both *bipartite* graphs, which naturally splits into two sets of vertices such that there are no edges within the sets. In the first example the sets are programs and resources. In the second example you can divide in many ways, for example the older spouse in set A and the younger in set B.

More examples

- The set of countries, where u is adjacent to v if u and v share a border
- The set of all people, where u and v are adjacent if they are friends on Facebook

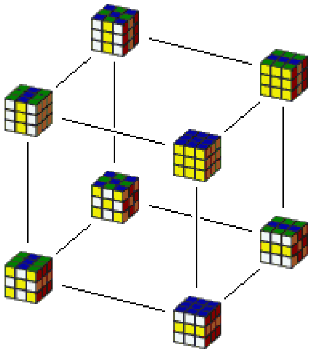
The first example here is the subject of the four colour theorem, which states that any map can be coloured with only four colours, with no two adjacent countries the same colour.

The internet



http://www.opte.org/maps/

Rubik's cube



Here the vertices are particular configurations of a Rubik's cube, which are adjacent if there is a move that takes one to the other.
https://www.jaapsch.net/puzzles/cayley.htm

Degrees

The *degree* of a vertex u is the size of its neighbourhood:

$$d(u) = |N_u|$$

Every edge (u, v) adds 1 to the degree of u and 1 to the degree of v, so we have

$$2|E| = \sum_{u \in V} d(u).$$

(Remember we count (u, v) and (v, u) together as a single edge.)

Handshaking lemma

$$2|E| = \sum_{u \in V} d(u).$$

Since the sum of the degrees is even, **there must be an even number of vertices with odd degree.** This is called the *handshaking lemma*.

“The number of people who have shaken an odd number of hands is even.”

Proof Part 1

Proof. In any graph, each edge connects to exactly two vertices, u and v. Therefore, each edge contributes one to the degree of both vertices. Hence, when we sum the degrees, since the degree is the number of edges connected to any given vertex, each edge will be counted twice making the sum two times the number of edges in the graph. Therefore,

$$\sum_{u \in V} d(u) = 2|E|$$

Proof Part 2

The sum in the previous part can be broken down into sums of the odd and even components.

$$\sum_{u \in V} d(u) = \sum_{\substack{u \in V \\ \text{deg u odd}}} d(u) + \sum_{\substack{u \in V \\ \text{deg u even}}} d(u)$$

The second term on the right of this equation must be even due to the fact that it's summing vertices whose degrees are all even, and $\sum even = even$. The sum on the left is even as per the previous proof. The first term on the right is summing over the odd degrees and *must* be even, since if it were odd, odd + even is not even. In order for a sum of odd degrees to be even, there must be an even number of them, and so the number of odd degree vertices is even.

$\square$

Outline

Paths

A *path* is a sequence of vertices $v_1, v_2, \ldots, v_j$ such that

- No vertex appears more than once
- v_k is adjacent to v_{k+1} for each $k = 1 \ldots j - 1$.
- The length of the path is $j - 1$ (which is the number of edges)

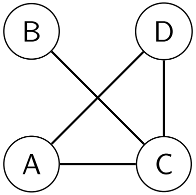
We sometimes talk about edges being in a path. An edge (s, t) is in a path of $s = v_k$ and $t = v_{k+1}$ for some k.

Cycles

A *cycle* is like a path, but $v_1 = v_j$, so that it loops back on itself.

- The length of the cycle is $j - 1$, which is the number of edges, and also the number of vertices.
- For cycles, we don't care about the starting vertex, so A, B, C, A is the same cycle as B, C, A, B.
- We might not duplicate the start/end vertex and just specify the cycle A, B, C as opposed to the path A, B, C.

Path and cycle examples



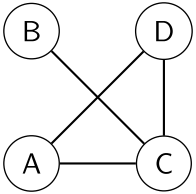
Here A, D, C, A is a cycle of length 3 and A, D, C, B is a path of length 3. Other paths include A, C (length 1), and D, C, A (length 2), etc..

Connectedness

If there is a path from u to v for every $u, v \in V$, then we say that G is *connected*. Otherwise we say G is *disconnected*.

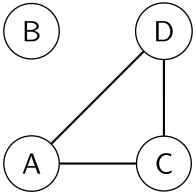
- If for every $u, v \in S \subseteq V$ there is a path between u and v, and there are no paths to any vertices outside S, then we call S a *connected component* of G.

Connected examples



Connected

Connected examples



Disconnected. The connected components are $\{B\}$ and $\{A, C, D\}$.

Distances

The *distance* between vertices u and v is the length of a shortest path from u to v

Calculating distances

Fix vertex u in V. Then we define D_j to be the set of vertices distance j from u, and V_j the set of vertices that are distance at least j from u. More specifically:

$$V_j = \begin{cases} V & : j = 0 \\ V_{j-1} \setminus D_{j-1} & : j > 1 \end{cases}$$

$$D_j = \begin{cases} \{u\} & : j = 0 \\ N_{V_j}(D_{j-1}) & : j > 0 \end{cases}$$

This pair of recursive definitions depend on each other.

This idea is the basis for Dijkstra's algorithm for shortest path finding.

Using Python to calculate distances

```
def distanceClasses(V, E, u):
    V0 = V                  # V_0 = V
    D = [ {u} ]             # D[0] = D_0 = {u}
    return distanceClassesR(V0, E, D)

def distanceClassesR(V, E, D):
    Vnew = V - D[-1]              # V_{j} = V_{j-1} / D_{j-1}
    if len(Vnew) == 0:            # Already considered all elements?
        return D
    Dnew = D + [ NS(Vnew, E, D[-1]) ]  # D_{j} = N_{V_j}(D_{j-1})
    return distanceClassesR(Vnew, E, Dnew)
```

To find the distance from vertex *u* we do:

```
D = distanceClasses(V, E, u)
```

This returns a Python list. We can retrieve D_j using `D[j]`.

Bipartite graphs

A *bipartite* graph, is a set of graph vertices decomposed into two disjoint sets (A, B) (sets that don't share any elements) such that no two graph vertices within the same set are adjacent. (A, B) is then called a *bipartition* of G.

In our distance finding method, set $A = D_0 \cup D_2 \cup D_4 \ldots$ (the union of all vertices an even distance away from the starting vertex) and $B = D_1 \cup D_3 \cup D_5 \ldots$ (the union of all vertices an odd distance away). Then if no two vertices in the same distance set (D_j) are adjacent, (A, B) is a bipartition of G.

Bipartite Graphs

Theorem

A graph G is bipartite if and only if G has no cycles of odd length.

Outline

Computational problems

Many, many computational problems can be expressed in terms of graphs:

- Finding optimal airline route maps
- Finding optimal network topologies
- Finding optimal routes on a network (data, road, airline, etc.)
- Determining if a network is connected
- Searching a parameter space
- Determining if there is deadlock between processes waiting for resources
- Determining an order to execute programs with multiple dependencies
- . . . and many more

Some generic problems in graphs

- Find a spanning tree
- Find a shortest path
- Find a cycle
- Visit every vertex in a graph by moving along edges (graph traversal)

Two generic methods for solving these problems:

- Depth-first traversal (search)
- Breadth-first traversal (search)

Distance finding revisited

Recall our idea for finding shortest paths, which we can summarise as:

- Set $D_0 = \{u\}$ (starting vertex)
- Set $D_1 =$ neighbours of u
- Set $D_2 =$ neighbours of D_1 that we haven't seen before
- ...

We process vertices (add them to some D_j) in order of distance, closest vertices to u first.

Breadth first traversal

Breadth first traversal process the vertices in a graph in order of distance from some initial vertex

- Set $D_0 = \{u\}$ and process u
- Set $D_1 =$ neighbours of u, and process them
- Set $D_2 =$ neighbours of D_1 that we haven't seen before and process them
- ...

The difference is that we do some other processing on the vertices, and we may not bother to keep track of the sets D_j.

Breadth first traversal in Python

```
def breadthFirst(V, E, u):
    print(u)                    # Process vertex u
    V0 = V                      # V_0 = V
    D = {u}                     # D[0] = D_0 = {u}
    return breadthFirstR(V0, E, D)

def breadthFirstR(V, E, D):
    Vnew = V - D                # V_{j} = V_{j-1} / D_{j-1}
    if len(Vnew) == 0:          # Already considered all elements?
        return
    Dnew = NS(Vnew, E, D)       # D_{j} = N_{V_j}(D_{j-1})
    for u in Dnew:
        print(u)                # Process vertex u
    return breadthFirstR(Vnew, E, Dnew)
```

We call this like:

```
breadthFirst(V, E, u)
```

Here we just print, but you can do additional processing on each vertex.

Uses of breadth first search

- Building spanning trees
- Finding shortest paths
- Finding cycles
- Finding connected components
- Testing if a graph is bipartite
- Finding maximum flows (more on this next time)

Depth first traversal

- Breadth first traversal tries to stay as close to u as possible, only moving further away when we've visited everything close by.
- Depth first traversal tries to go as far away from u as possible, and then backtracking when it can't go any further.
- Depth first search always moves between adjacent vertices, so it is good for applications like maze solving where you can't instantly move to another vertex elsewhere in the graph.

Depth first traversal

```
def depthFirst(V, E, u):
    T = {u}              # set of vertices already seen
    depthFirstR(V, E, u, T)

def depthFirstR(V, E, u, T):
    print(u)                     # process vertex u
    if len(T) == len(V):         # already seen all?
       return T
    Nu = N(V, E, u) - T          # Neighbours not already seen
    T.update(Nu)                 # Update set of vertices seen
    for v in Nu:
        T.update(depthFirstR(V, E, v, T))  # add vertices seen
    return T                               # in recursive call
```

Call like:

```
depthFirst(V, E, u)
```

Uses of depth first traversal

- Building spanning trees
- Finding cycles
- Finding connected components
- Maze solving
- Finding a topological ordering (more on this next time)

Important example

If we have a situation where we have to make many choices, we can make a graph (often a tree), where each vertex represents a choice we need to make, and following an edge means making a choice. Leaves are situations where there are no more choices to make.

- In a maze, set the vertices to be the "interesting" places (start, end, places where we must choose which way to go). Edges go between places that are connected in the maze. Then we can use depth first traversal to find a path from the start to the end
- In a game (chess, Rubik's cube, etc.) we can have the vertices be configurations of the game, and edges are legal moves. Then BFT can be used to find the choices to make to get to a winning position as fast as possible.

In these cases, we often don't know what the graph is ahead of time, but we can compute allowed choices.