Part A Graph Theory

Marc Lackenby*

This course provides a short introduction to Graph Theory (the mathematical theory of 'networks'). Our approach will be to develop the theory as it is needed for the rigorous analysis of practical problems, namely Minimum Cost Spanning Trees, Shortest Paths, Bipartite Matching, and the Chinese Postman Problem.

These notes are based on those of Peter Keevash.

1 Introduction

Networks are everywhere in our world: transportation networks (roads, railways, ...), communication networks (phones, email, ...), social networks (Facebook, ...), ... many other kinds of networks. Mathematically, a network is just a collection of points called 'vertices' joined by lines called 'edges'. Mathematicians call these graphs. What kind of mathematical questions arise when we think about networks?

One such question that gripped the popular imagination in the mid 20th century, and was the subject of a famous 1967 experiment by Stanley Milgram, was the 'small world problem', also known as 'six degrees of separation': can any two people be linked by a chain of people, of length at most six, such that any link in the chain is a pair of people who know each other? For a modern day version, assume that Facebook has 10^9 users and each of them has exactly 100 friends. (A rough approximation to the truth.) Is six degrees of separation logically possible under these assumptions?

Another popular question, posed by Francis Guthrie in 1852, appears at first to be unrelated to networks: given any map, is it possible to colour the countries using only four different colours, so that any two countries sharing a border receive different colours? The relationship to networks appears when we replace each country by a vertex and join two vertices by an edge when the corresponding countries share a border. The Four Colour Conjecture did not become the Four Colour Theorem until 1976, after a controversial computer-assisted proof by Appel and Haken.

2 Connectedness

The government wants to build a new high speed rail network that links all of the major cities in the country. Of course, the fastest network will be achieved by linking every pair of cities, but this will be very expensive, and perhaps unpopular for environmental reasons. In fact, the government's main priority is not to minimise journey times, but rather to minimise the cost subject to making a connected network (even if this forces everyone to go via London). Let us make some definitions and formulate this problem mathematically.

A *graph* $G = (V(G), E(G))$ consists of two sets $V(G)$ (the *vertex set*) and $E(G)$ (the *edge set*), where each element of $E(G)$ consists of a pair of elements of $V(G)$. We represent G visually by drawing a point for each vertex and a line between any pair of points that form an edge. In our example, vertices will represent cities and edges will represent potential rail lines.

*Mathematical Institute, University of Oxford, Oxford, UK. Email: lackenby@maths.ox.ac.uk

(We will always assume without further comment that $|V(G)|$ is finite. We use the term 'graph' where some would say 'simple graph', using 'graph' for a more general structure which allows several 'parallel' edges between a given pair of vertices and 'loop' edges that join a vertex to itself. We write $uv = \{u, v\} = vu$ for the (unordered) pair representing an edge between u and v.)

Next we want to formalise the concept of connectedness. Let G be a graph. A *walk* in G is a sequence W of vertices $v_1, \ldots, v_t$ such that $v_i v_{i+1} \in E(G)$ for all $1 \leq i < t$. If we want to specify the start and end then we call W an *xy-walk* where $x = v_1$ and $y = v_t$. If the vertices in W are distinct we call it a *path*, or if we want to specify the ends an *xy-path*. If $x = y$ we call W a *closed walk*. If $x = y$ but the vertices are otherwise distinct and W has at least 3 vertices then we call W a *cycle*. We also regard paths and cycles as subgraphs of G.

We say that G is *connected* if for any x, y in $V(G)$ there is an xy-walk in G. We say that two vertices x and y of a graph G *lie in the same component* if they are joined by an xy-walk. Clearly this forms an equivalence relation and the partition of $V(G)$ into equivalence classes expresses G as a union of disjoint connected graphs called its *components*.

Let G be a connected graph. Suppose that for each edge $e \in E(G)$ we are given a 'cost' $c(e) > 0$. For any $S \subseteq E(G)$ we call $c(S) = \sum_{e \in S} c(e)$ the cost of S. Our task:

Find $S \subseteq E(G)$ with minimum possible $c(S)$ such that $(V(G), S)$ is a connected graph.

A silly way of solving this task would be to list all $S \subseteq E(G)$, check each one to see whether $(V(G), S)$ is a connected graph, compute $c(S)$ for each, and take the best one. This is silly because there are $2^{|E(G)|}$ subsets of $E(G)$, so we could never check them all in practice unless G is very small. We are interested in 'efficient algorithms'. We will not define this concept precisely in this course, but it will be exemplified by the algorithms that we present.

What can we say about the possible $S \subseteq E(G)$ that solves our task? One obvious property is that $(V(G), S)$ is 'minimally connected', i.e. $(V(G), S)$ is connected but $(V(G), S \setminus \{e\})$ is not connected for any $e \in S$ (otherwise we contradict minimality of $c(S)$). This motivates the next section.

3 Trees

A *tree* is a minimally connected graph. (Draw some pictures to see why the name is apt.) We postpone the task proposed in the previous section until we have proved some basic properties of trees.

If a graph G has no cycle we call it *acyclic*.

Lemma 1. *Any tree is acyclic.*

Proof. Let G be a tree, i.e. G is minimally connected. Suppose for a contradiction that G contains a cycle C. Let $e \in E(C)$. We will obtain our contradiction by showing that $G - e := (V(G), E(G) \setminus \{e\})$ is connected. Let P be the path obtained by deleting e from C. Consider any x, y in $V(G)$. As G is connected, there is an xy-walk W in G. Replacing any use of e in W by P gives an xy-walk in $G - e$. Thus $G - e$ is connected, contradiction. □

There are many equivalent characterisations of trees, any of which could be taken as the definition. Here is one:

Lemma 2. *G is a tree if and only if G is connected and acyclic.*

Proof. If G is a tree then G is connected by definition and acyclic by Lemma 1. Conversely, let G be connected and acyclic. Suppose for a contradiction that $G - e$ is connected for some $e = xy \in E(G)$. Let W be a shortest xy-walk in $G - e$. Then W must be a path, i.e. have no repeated vertices, otherwise we would find a shorter walk by deleting a segment of W between two visits to the same vertex. Combining W with xy gives a cycle, contradiction. □

Remark. The fact that a shortest walk between two points is a path is often useful. More generally, considering an extremal (shortest, longest, minimal, maximal, ...) object is often a useful proof technique. Another example:

Lemma 3. *Any two vertices in a tree are joined by a unique path.*

Proof. Suppose for a contradiction that this fails for some tree G. Choose x, y in $V(G)$ so that there are distinct xy-paths P_1, P_2, and P_1 is as short as possible over all such choices of x and y. Then P_1 and P_2 only intersect in x and y, so their union is a cycle, contradicting Lemma 1. □

We will continue to study trees. First some more terminology. Let G be a graph. If $uv \in E(G)$ we say that u and v are *neighbours*. We also say that u and v are *adjacent*. The degree $d(v)$ of v is the number of neighbours of v in G. A *leaf* is a vertex of degree one, i.e. with a unique neighbour.

Lemma 4. *Any tree with at least two vertices has at least two leaves.*

Proof. Consider any tree G. Let P be a longest path in G. The two ends of P must be leaves. Indeed, an end cannot have a neighbour in $V(G) \setminus V(P)$, or we could make P longer, and cannot have any neighbour in $V(P)$ other than the next in the sequence of P, or we would have a cycle. □

The existence of leaves in trees is useful for inductive arguments, via the following lemma. Given $v \in V(G)$, let $G-v$ be the graph with $V(G-v) = V(G) \setminus \{v\}$ and $E(G-v) = \{xy \in E(G) : v \notin \{x, y\}\}$.

Lemma 5. *If G is a tree and v is a leaf of G then $G - v$ is a tree.*

Proof. By Lemma 2 it suffices to show that $G - v$ is connected and acyclic. Acyclicity is immediate from Lemma 1. Connectedness follows by noting for any x, y in $V(G) \setminus \{v\}$ that the unique xy-path in G is contained in $G - v$. □

An easy example of such an inductive argument:

Lemma 6. *Any tree on n vertices has $n - 1$ edges.*

Proof. By induction. A tree with 1 vertex has 0 edges. Let G be a tree on $n > 1$ vertices. By Lemma 4, G has a leaf v. By Lemma 5, $G - v$ is a tree. By induction hypothesis, $G - v$ has $n - 2$ edges. Replacing v gives $n - 1$ edges in G. □

We conclude this section with another characterisation of trees. First we note that any connected graph G contains a minimally connected subgraph (i.e. a tree) with the same vertex set, which we call a *spanning tree* of G.

Lemma 7. *A graph G is a tree on n vertices if and only if G is connected and has $n - 1$ edges.*

Proof. If G is a tree then G is connected by definition and has $n - 1$ edges by Lemma 6. Conversely, suppose that G is connected and has $n - 1$ edges. Let H be a spanning tree of G. Then H has $n - 1$ edges by Lemma 6, so $H = G$, so G is a tree. □

4 Minimum Cost Spanning Trees

We return to the high speed rail question. Recall G is a connected graph and we have some cost $c(e) > 0$ for every edge $e \in E(G)$. For any $S \subseteq E(G)$ we call $c(S) = \sum_{e \in S} c(e)$ the cost of S. The problem we wish to solve is:

Find $S \subseteq E(G)$ with minimum possible $c(S)$ such that $(V(G), S)$ is a connected graph.

A solution is necessarily a spanning tree for G. Recall that this is a tree $T = (V(G), S)$ where $S \subseteq E(G)$. We say that T is a *minimum cost spanning tree* of G if any other spanning tree T' satisfies $c(T') \geq c(T)$. How can we find one efficiently?

One natural method to try is the 'greedy algorithm': choose edges one at a time, each time choosing the cheapest edge that does not create a cycle. There are various versions of this algorithm; we will describe the one due to Kruskal.

Kruskal's Algorithm. At step $i \geq 0$, we will keep track of a subset $A_i \subseteq E(G)$. This will have the property that $(V(G), A_i)$ is acyclic. Start with $A_0 = \emptyset$. At step $i \geq 0$, is there an edge $e \in E(G) \setminus A_i$ such that $(V(G), A_i \cup \{e\})$ is acyclic? If no, then output $A = A_i$ and stop. If yes, then set $A_{i+1} = A_i \cup \{e\}$ for one such e such that $c(e)$ is minimal, and proceed to step $i + 1$.

You should try a few examples on small graphs to understand the algorithm and check that it does find minimum cost spanning trees in your examples. However, it is not obvious that it will always work, and indeed there are different problems in graph theory for which greedy algorithms don't always work. Fortunately, the greedy algorithm always works for the minimum cost spanning tree problem, as shown by the following theorem.

Theorem 8. *$(V(G), A)$ is a minimum cost spanning tree of G.*

Proof. The first step of the proof is to show that $(V(G), A)$ is a spanning tree of G. To see this, we note that A_i is acyclic for any $i \geq 0$ by definition. Suppose for a contradiction that the algorithm terminates with $A = A_i$ such that $(V(G), A)$ is not connected. As G is connected, there is at least one edge e of G whose endpoints are in different components of $(V(G), A)$. Then $A_i \cup \{e\}$ is acyclic, so the algorithm did not terminate at step i. This contradiction shows that $(V(G), A)$ is a spanning tree of G.

Now let $\mathcal{M}$ be the set of $B \subseteq E(G)$ such that $(V(G), B)$ is a minimum cost spanning tree of G. We will prove by induction on $i \geq 0$ that

(*) there is $B \in \mathcal{M}$ with $A_i \subseteq B$.

Note that (*) will suffice to prove the theorem, as when we apply it to $A_i = A$ we will have $|A| = |B|$ by Lemma 6, and $A \subseteq B$, so $A = B \in \mathcal{M}$.

For the base case $i = 0$ of (*) we have $A_0 = \emptyset$, so any $B \in \mathcal{M}$ satisfies (*). For the induction step, suppose for some $i \geq 0$ we have $A_i \subseteq B \in \mathcal{M}$. We can suppose $A_i \neq A$, otherwise the proof is complete. Consider $A_{i+1} = A_i \cup \{e\}$ given by the algorithm. We need to find $B' \in \mathcal{M}$ with $A_{i+1} \subseteq B'$. We can assume $e \notin B$, otherwise we could take $B' = B$.

Let $e = xy$ and let P be the unique xy-path in the spanning tree $(V(G), B)$. Then $C = P \cup \{e\}$ is a cycle. As A_{i+1} is acyclic, we can choose $f \in C \setminus A_{i+1}$. Let $B' = (B \setminus \{f\}) \cup \{e\}$. To finish the proof we need to show that

i. $A_{i+1} \subseteq B'$,
ii. $(V(G), B')$ is a spanning tree, and
iii. $c(B') \leq c(B)$.

For (i), note that $A_{i+1} = A_i \cup \{e\} \subseteq B'$, as $A_i \subseteq B$ and $f \notin A_{i+1}$. For (ii), note that B' is connected, for the following reason. Any two vertices in $V(G)$ are joined by a path in B. Replace each occurence of f in this path by $C \setminus \{f\}$. Also B' has $|V(G)| - 1$ edges. So it is a spanning tree by Lemma 7. For (iii), note that $A_i \cup \{f\} \subseteq B$, so $A_i \cup \{f\}$ is acyclic. Now e was chosen so that $c(e)$ is minimal among all edges e such that $A_i \cup \{e\}$ is acyclic. Hence, $c(e) \leq c(f)$. So $c(B') = c(B) - c(f) + c(e) \leq c(B)$. This finishes the proof of the inductive step of (*), and so of the theorem. □

How fast is this algorithm? To make this question mathematically precise would take us far afield (we would need to define a model of computation). In this course, we will take the intuitive approach of estimating the number of 'steps' taken by an algorithm, where a 'step' should be a 'simple' operation. In each iteration we add an edge, so there will be $|V(G)| - 1$ iterations. If at each stage of the algorithm, we naively find the next edge by checking every edge then there will be $|E(G)|$ steps in each iteration, giving about $|V(G)||E(G)|$ steps in total.

We say that the running time is $O(|V(G)||E(G)|)$, where the 'big O' notation means that there is a constant C so that for any graph G the running time is at most $C|V(G)||E(G)|$. Here 'running time' could be measured in any units, say milliseconds on your favourite computer, as changing the units or using a different computer will just replace C by a different constant.

A smarter implementation is to start by making a list of all edges ordered by cost, cheapest first. Then at each step we go through the list from the start, discarding edges that make a cycle until we find the first edge which can be added. This gives a running time that is 'roughly comparable' [1] with the number of edges, which is essentially best possible.

5 Euler tours

A very early theorem of Graph Theory, perhaps even the first, was proved in 1766 by Euler, concerning a popular problem of the time called 'the bridges of Königsberg'. Königsberg is divided into 4 districts by the river Pregel and has 7 bridges. The problem was to decide whether it is possible to take a walk that crosses every bridge exactly once. To translate this into graph theory we construct a graph in which there is a vertex for each district and an edge representing each bridge.

Some of the bridges join the same pair of districts, so correspond to parallel edges between the same pair of vertices. This is not allowed by the definition of 'graph' we are generally using in this course (our graphs are 'simple', in that we do not loops or parallel edges), but in fact the results in this section are also true if we allow parallel edges.

Let W be a walk in a graph G. We call W an *Euler trail* if every edge of G appears exactly once in W. Let W be an Euler trail. We call W an *Euler tour* if it is closed, i.e. it starts and ends at the same vertex. Here we will only solve the problem of finding an Euler tour; the solution of the Euler trail problem can be deduced (Q7 on Problem Sheet 1).

What can we say about a graph G with an Euler tour W? Clearly, G must be connected after we delete all *isolated* vertices (i.e. vertices of degree zero). Next we note that each visit of W to a vertex v uses two edges at v (one to arrive and one to leave). This is also true of the start and end vertex of W if we consider them to be a single visit. (Or we can think of the vertex sequence of W as being written around a circle rather than along a line, so that there is no start or end, and each visit uses two edges.) As every edge is used exactly once, we deduce that every vertex has even degree;

[1] Here we ignore many subtleties, such as the time required to sort $E(G)$ by cost or to maintain a component data structure that allows quick checking of whether an edge creates a cycle.

we call a graph with this property *Eulerian*. These necessary conditions are also sufficient:

Theorem 9. *(Euler) Let G be a connected Eulerian graph. Then G has an Euler tour.*

In fact, we will show that we can find an Euler tour efficiently, using the following algorithm.

Fleury's Algorithm. Start at any vertex. We will follow a walk, erasing each edge after it is used (erased edges cannot be used again). At each stage, ensure that the following holds:

i. when the edge is removed, the resulting graph is connected once isolated vertices are removed, and
ii. we do not run along an edge to a leaf, unless this is the only edge of the graph.

We require a useful lemma.

Lemma 10. *In any graph, there are an even number of vertices with odd degree.*

Proof. Since every edge has two endpoints,

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

Therefore, in the sum, there must be an even number of occurrences of $d(v)$ for which $d(v)$ is odd. □

Proof of Theorem. We show that Fleury's Algorithm produces an Euler tour. Note first that at each stage of the algorithm, either there are two vertices of odd degree (the initial vertex u and the current one) or there are no vertices of odd degree.

Suppose for a contradiction Fleury's Algorithm fails. Say it stops at some vertex v and can go no further. Let H be the component of the current graph containing v. The degree of v in H must be positive, as otherwise the previous step violated (ii). If the degree of v in H is one, then we can continue the walk. So there are at least two edges of H containing v. Since the algorithm cannot continue, the graph $H - e$ is disconnected for each edge e containing v. Hence, the edges e incident to v all have endpoints in distinct components of $H - v$. So, we can choose one edge vw, such that the component C of $G - vw$ which contains w does not contain the first vertex u of the walk. But then w is the only vertex of odd degree in C, which is impossible by Lemma 10. □

6 Hamiltonian cycles

In the previous section, we investigated graphs that admit an Euler tour, which is a closed walk that traverses every edge exactly once. There turned out be a very simple characterisation of such graphs. In particular, to decide whether a graph G is Eulerian requires only polynomially many 'steps' as a function of $|V(G)|$ and $|E(G)|$.

There is a seemingly related question that one can ask about a graph G: does there exists a closed walk that visits every vertex exactly once? In fact, such a walk is a cycle (provided G has more than two vertices) and is known as a *Hamiltonian cycle*. When a graph G contains such a cycle, it is *Hamiltonian*.

Unlike the case of Eulerian tours, it turns out that there is, almost certainly, no efficient algorithm to determine whether a general graph G is Hamiltonian. By 'efficient', we mean that the algorithm gives the answer after polynomially many 'steps', as a function of $|V(G)|$ and $|E(G)|$. But what do we mean by 'almost certainly'? Currently, mathematicians do not have a proof that there is no efficient algorithm to determine whether a general graph G is Hamiltonian. However, we do no know that there is no efficient algorithm if we assume the famous conjecture $P \neq NP$. But to discuss this conjecture would take us too far afield.

We will therefore content ourselves with a sufficient condition for a graph to be Hamiltonian.

Theorem 11 (Ore's theorem)**.** *Let G be a connected graph with $n \geq 3$ vertices. Suppose that for every pair of non-adjacent vertices x and y, $d(x) + d(y) \geq n$. Then G is Hamiltonian.*

Corollary 12 (Dirac's theorem)**.** *If G is connected with $n \geq 3$ vertices and for every vertex v, $d(v) \geq n/2$, then G is Hamiltonian.*

We first note if G is Hamiltonian and has n vertices, then the length of the longest cycle is n and the length of the longest path is $n - 1$. (The *length* of a path is its number of edges.)

Lemma 13. *If G is connected and non-Hamiltonian, then the length of the longest path is least the length of the longest cycle.*

Proof. Let C be a longest cycle, with length ℓ. Since G is non-Hamiltonian, there is some vertex not in C. Since G is connected, there is therefore some edge uv with one endpoint u in C and one endpoint v not in C. Removing an edge of C incident to u and adding uv gives a path of length ℓ. □

Proof of Theorem. Suppose that G is not Hamiltonian. Let $P = x_1 \cdots x_k$ be a longest path. It has length $k - 1$. So by Lemma 13, G does not have a cycle of length k. So x_1 and x_k are not adjacent. (Here, we are using that $n \geq 3$ and hence that $k \geq 3$.) Hence, by our assumption, $d(x_1) + d(x_k) \geq n$.

There is no integer i such that x_1 is adjacent to x_{i+1} and x_k is adjacent x_i. Otherwise, $x_1 \cdots x_i x_k x_{k-1} \cdots x_{i+1} x_1$ would be a cycle of length k. So the sets

$$A = \{i : x_1 x_{i+1} \in E(G)\}, \qquad B = \{i : x_i x_k \in E(G)\}$$

are disjoint subsets of $\{1, \cdots, k-1\}$. Every neighbour of x_1 lies in P, and similarly every neighbour of x_k lies in P, as P is a longest path. So, A has size $d(x_1)$, and B has size $d(x_k)$. Since A and B are disjoint, $d(x_1) + d(x_k) \leq k - 1 < n$, which is a contradiction. Hence, G must be Hamiltonian. □

7 Shortest Paths

How do you find the quickest route from A to B? Maybe you ask your satnav, but how does your satnav find the route? It doesn't check all options, as there are too many: it uses an efficient algorithm. We formulate the problem mathematically as follows. Let G be a connected graph. Let $\ell(e) > 0$ for $e \in E(G)$ be the 'length' of the edge e. The *ℓ-length* of a path P is $\ell(P) = \sum_{e \in E(P)} \ell(e)$. Given x and y in $V(G)$, an *ℓ-shortest xy-path* is an xy-path P that minimises $\ell(P)$.

We will now describe Dijkstra's Algorithm for finding an ℓ-shortest xy-path. The idea of the algorithm is to maintain a 'tentative distance from x' called $D(v)$ for each $v \in V(G)$. At each step of the algorithm we finalise $D(u)$ for some vertex u. At the end of the algorithm all $D(u)$ will be equal to the correct value, i.e. $D(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*.

Dijkstra's Algorithm. Start by letting $U = V(G)$, $D(x) = 0$, $D(v) = \infty$ for all $v \neq x$.

Repeat the following step: if $U = \emptyset$ stop, otherwise pick $u \in U$ with $D(u)$ minimal, delete u from U, and for any $v \in U$ with v adjacent to u and satisfying $D(v) > D(u) + \ell(uv)$ replace $D(v)$ by $D(u) + \ell(uv)$.

This algorithm was short to describe, but it is not obvious what it does (try some examples), or that it works: we will prove this below.

In fact, Dijkstra's Algorithm can be used to do more: for any $x \in V(G)$ we can construct a spanning tree T such that for any $y \in V(G)$ the unique xy-path in T is an ℓ-shortest xy-path. We call T an *ℓ-shortest paths tree rooted at x.*

We now describe how to obtain T. For any vertex $v \neq x$, the *parent* of v is the last vertex u such that we replaced $D(v)$ by $D(u) + \ell(uv)$ during the algorithm. We obtain T by drawing an edge from each vertex $v \neq x$ to the parent of v.

Lemma 14. *T is a tree, and for each $u \in V(G)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T.*

Proof. After any step, we have defined the parents of all vertices in $C = V(G) \setminus U$, other than x. Let T_C be obtained by drawing an edge from each $v \in C \setminus \{x\}$ to its parent. So $V(T_C) = C$. We show by induction on $|C|$ that T_C is a tree and for each $u \in V(T_C)$ we have $D(u) = \ell(P_u)$ where P_u is the unique xu-path in T_C.

Base case: we start with $V(T_C) = \{x\}$ and no edges, which is a tree, with $D(x) = 0 = \ell(P_x)$. Induction step: when we delete u from U, we add u to C, and add an edge from u to the parent v of u, i.e. we add a leaf to T_C, and so obtain another tree. By definition of parent and induction we have $D(u) = D(v) + \ell(vu) = \ell(P_v) + \ell(vu) = \ell(P_u)$. □

Theorem 15. *T is an ℓ-shortest paths tree rooted at x.*

Proof. For each $u \in V(G)$ let $D^*(u) = \ell(P_u^*)$ for some ℓ-shortest xu-path P_u^*. We show by induction that in each step of the algorithm, when u is deleted we have $D(u) = D^*(u)$. For the base case we have $u = x$ and $D(u) = D^*(u) = 0$. For the induction step, consider the step where we delete some u from U, and suppose for contradiction that $D(u) > D^*(u)$. Let yy' be the first edge of P_u^* with $y \notin U$ and $y' \in U$. By the induction hypothesis, $D(y) = D^*(y)$. Now

$$D(y') \leq D(y) + \ell(yy') = D^*(y) + \ell(yy') = \ell(P_y^*) + \ell(yy') \leq \ell(P_u^*) = D^*(u) < D(u).$$

The first inequality uses the update rule for y and y': when y was removed from U, $D(y')$ was replaced by $D(y) + \ell(yy')$ if that was smaller, and so after this, $D(y') \leq D(y) + \ell(yy')$. The second inequality holds as the subpath of P_u^* from x to y must be an ℓ-shortest xy-path (or we would find a shorter path to u). However, $y' \in U$ with $D(y') < D(u)$ contradicts the choice of u in the algorithm. So $D(u) = D^*(u)$. □

Remark. The running time of this implementation of Dijkstra's Algorithm is $O(|V(G)||E(G)|)$. A better implementation (which we omit) gives a running time of $O(|E(G)| + |V(G)| \log |V(G)|)$.

8 Matching

The Marriage Problem: given n men and n women, under what conditions is it possible to pair each man with a woman such that every pair know each other?

(This 'non-PC' version of the problem is the easiest to solve. It becomes more interesting if we allow same-sex couples or larger groups, but we will not consider these problems in this course.)

As usual, we require some definitions for a mathematical formulation of the problem. Let G be a graph. We say $M \subseteq E(G)$ is a *matching* if the edges in M are pairwise disjoint. We say M is *perfect* if every vertex belongs to some edge of M. We say that G is *bipartite* if we can partition $V(G)$ into two sets A and B so that every edge of G crosses between A and B.

In this terminology, the marriage problem asks when a bipartite graph has a perfect matching. We will return to this question later. First we consider the algorithmic question of how to find a matching of maximum size.

A natural first attempt is the greedy algorithm: keep picking edges where in each step we choose an edge disjoint from all previous choices. However, this does not work: it does produce a matching that is maximal in the sense that no edge can be added, but it may not have maximum size. For example, suppose G is a path with three edges. If we were foolish enough to choose the middle edge we would be stuck, whereas the maximum matching is obviously obtained by choosing the two outer edges. This suggests the following method for improving a matching. Suppose G is a graph, M is a matching in G, and P is a path in G. We say P is *M-alternating* if every other edge of P is in M. We say P is *M-augmenting* if P is M-alternating and its end vertices are not in any edge of M.

Lemma 16. *Let M be a matching in G. Then M is not of maximum size if and only if there is an M-augmenting path in G.*

Proof. If there is an M-augmenting path P in G then we can find a larger matching by 'flipping' P: replace M by $M \setminus (M \cap E(P)) \cup (E(P) \setminus M)$. Conversely, suppose that M^* is a matching in G with $|M^*| > |M|$. Let $H = M \cup M^*$. Every vertex has degree at most 2 in H, so each component of H is an edge, path or cycle, the edge components consist of $M \cap M^*$, and the edges in path and cycle components alternate between M and M^*. As $|M^*| > |M|$ we can find a path component with more edges of M^* than M: this is an M-augmenting path in G. □

Lemma 16 reduces the algorithmic question of finding a maximum matching in G to the following: given a matching M in G, find an M-augmenting path or show that there is none.

Now suppose that G is bipartite, with parts A and B. The latter problem can be further reduced to 'search in a one-way road system' (also known as a directed graph). Indeed, suppose that we put directions on $E(G)$, so that all edges in M are one-way from B to A, and all edges not in M are one-way from A to B. Let A^* and B^* be the vertices in A and B that are 'uncovered', i.e. not in any edge of M. Then an M-augmenting path is equivalent to a directed path from A^* to B^*, i.e. a path that respects directions of edges. We can find such a path or show that none exists by the following simple algorithm, which applies to any directed graph G.

Search Algorithm. Let G be a directed graph and $R \subseteq V(G)$. Repeat the following step: if there is any edge directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

The search algorithm stops when we have added to R all vertices that can be reached from the initial R by a directed path. To find an M-augmenting path, we apply it with $R = A^*$ and see whether the final R intersects B^*. If it does not then there is no M-augmenting path, so M has maximum size. If it does, then we can work backwards to find an M-augmenting path, and use this to increase the matching.

The above algorithm for finding a maximum matching in a bipartite graph G is known as the *Hungarian Algorithm*.

The running time of the search algorithm is $O(|V(G)||E(G)|)$. This is because each step of the algorithm increases the number of vertices in R by at least one, and at each step, we need to go through all the edges of the graph to see whether they lead to a way of enlarging R. In the Hungarian algorithm, there are at most $|V(G)|/2$ iterations that increase the size of the matching. So it has running time $O(|V(G)|^2|E(G)|)$.

To continue our study of matchings in bipartite graphs, we start by briefly illustrating the idea of Duality of Linear Programs, which is of fundamental importance, in mathematical theory and practice. Recall that the maximum matching problem for G is to choose a maximum size set M of edges such that every vertex belongs to at most one edge of M. The 'dual' problem is the minimum cover problem for G, which is to choose a minimum size set C of vertices such that every edge contains at least one vertex of C (we call C a *cover* of G).

It is clear that these two problems have the following relationship known as 'weak duality': for any matching M in G and any cover C of G we have $|M| \leq |C|$. (To see this, define an injective map $f : M \to C$, where $f(e)$ is any vertex of $e \cap C$.)

This suggests the question of whether equality holds. If it does, then C provides a short proof that M is a maximum matching, and M provides a short proof that C is a minimum cover. (This is sometimes called a 'min-max property' or a 'good characterisation'.) The answer to the question is 'no' in general, e.g. if G is a triangle then the maximum matching has size 1 but the minimum cover has size 2. The following result shows that the answer is 'yes' in bipartite graphs.

Theorem 17. *(König's Theorem) In any bipartite graph, the size of a maximum matching equals the size of a minimum cover.*

Proof. Let G be a bipartite graph with parts A and B. Let M be a maximum matching in G. It suffices to find a cover C with $|C| = |M|$. Recall that we write A^* and B^* for the uncovered vertices in A and B. Consider the search algorithm for an M-augmenting path in G. The algorithm terminates with some set R that consists of all vertices reachable by M-alternating paths starting in A^*. As M is maximum there is no M-augmenting path, so $R \cap B^* = \emptyset$.

Let $C = (A \setminus R) \cup (B \cap R)$. We claim that C is a cover with $|C| = |M|$. We start by showing that C is a cover. Suppose not. Then there is $ab \in E(G)$ with $a \in A \cap R$ and $b \in B \setminus R$. However, this contradicts the definition of R, as b must be reachable from A^*: if $ab \in M$ we must reach a via b or if $ab \notin M$ we can reach b via a. Thus C is a cover.

It remains to show $|C| = |M|$. It suffices to show that every vertex in C is covered by some edge of M, and that no edge of M covers two vertices of C. (This will show $|C| \leq |M|$, and we noted previously that $|M| \leq |C|$ is immediate from the definitions.) Firstly, any $a \in A \setminus R$ is covered by M as $A^* \subseteq R$. Secondly, any $b \in B \cap R$ is covered by M, or $b \in B^* \cap R = \emptyset$ gives a contradiction. Finally, if $ab \in M$ with $a \in A \setminus R$, $b \in B \cap R$ then we can reach a via b, contradicting $a \notin R$. Thus $|C| = |M|$. □

We conclude this section with a solution to the 'marriage problem' of characterising when a bipartite graph has a perfect matching. Let G be a bipartite graph with parts A and B. We consider the more general question of whether there is a matching that covers every vertex in A; if $|B| = |A|$ then this will be perfect. For $S \subseteq A$ the *neighbourhood* of S is $N(S) = \cup_{a \in S}\{b : ab \in E(G)\}$. Note that if G has a matching M covering A then each $a \in S$ has a 'match' a' with $aa' \in M$, and the matches are distinct, so $|N(S)| \geq |S|$. This gives a necessary condition for G to have a matching; it is also sufficient:

Theorem 18. *(Hall's Theorem) Let G be a bipartite graph with parts A and B. Then G has a matching covering A if and only if every $S \subseteq A$ has $|N(S)| \geq |S|$.*

Proof. We have already remarked that the condition is necessary. Conversely, suppose that every $S \subseteq A$ has $|N(S)| \geq |S|$. Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geq |A|$. To see this, let $S = A \setminus C$, and note that by definition of 'cover' we have $N(S) \subseteq B \cap C$. Then $|C| = |A \cap C| + |B \cap C| \geq |A| - |S| + |N(S)| \geq |A|$. □

9 The Chinese Postman Problem

A postman collects a sack of letters from the sorting office, walks along every street to deliver them, and returns to the office. How can (s)he find the shortest route?

We formulate the problem using graph theory. Let G be a connected graph. Let W be a closed walk in G. We call W a *postman walk* in G if it uses every edge of G at least once. For each $e \in E(G)$ let $c(e) > 0$ be the length of e. The length of W is $c(W) = \sum_{e \in W} c(e)$. We want to find a shortest postman walk.

This is reminiscent of the Euler Tour problem considered earlier. There we wanted to use every edge exactly once. We can interpret a postman walk W as an Euler Tour in an *extension* of G, in which we introduce parallel edges, so that the number of parallel edges joining vertices x and y is the number of times that xy is used in W. Thus an equivalent reformulation of the Chinese Postman Problem is to find a *minimum weight Eulerian extension* G^* of G, i.e. G^* is obtained from G by copying some edges, so that all degrees in G^* are even, and $c(G^*)$ is as small as possible.

We will describe an algorithm due to Edmonds, which draws together several other elements of the course: Shortest Paths, Matchings and Euler Tours. We assume that we have access to an algorithm for finding the minimum weight perfect matching in a weighted graph (an algorithm for this problem was also found by Edmonds, but it is beyond the scope of this course).

Edmonds' Algorithm for the Chinese Postman Problem.

i. Let X be the set of vertices with odd degree in G. For each $x \in X$ find a c-shortest paths tree T_x rooted at x. Define a weight function w on pairs in X: let $w(xy) = c(P_{xy})$, where P_{xy} is the unique xy-path in T_x
ii. Find a perfect matching M on X with minimum w-weight. Let G^* be the Eulerian extension of G obtained by copying all edges of P_{xy} for all $xy \in M$.
iii. Find an Euler Tour W in G^*. Interpret W as a postman walk in G.

Note that the perfect matching step makes sense as $|X|$ is even, by Lemma 10. We require the following simple lemma for the analysis of the algorithm.

Lemma 19. *Let H be a graph in which not all degrees are even. Then there is a path in H such that both ends have odd degree.*

Proof. Pick a component of H containing a vertex of odd degree. By Lemma 10, there is another vertex of odd degree in H. Pick a path joining these two vertices. □

Theorem 20. *Edmonds' Algorithm finds a minimum length postman walk.*

Proof. Let W^* be a minimum length postman walk. It suffices to show that the algorithm finds a postman walk that is no longer than W^*. Let G^* be the Eulerian extension of G defined by W^*. Let H be the graph of copied edges: $E(H) = E(G^*) \setminus E(G)$. Note that the set of vertices with odd degree in H is X (i.e. the same set as for G).

We construct a set of paths in H by repeating the following procedure: if the current graph has any vertices of odd degree, apply Lemma 19 to find a path P such that both ends have odd degree, delete the edges of P and repeat. This procedure pairs up the vertices in X so that each pair is connected by a path in H.

Let $H' \subseteq H$ be the graph formed by the union of these paths. Let G' be the Eulerian extension of G defined by copying the edges of H'. Let W' be an Euler tour in G', interpreted as a postman walk in G. Then $c(W') \leq c(W^*)$, and by definition of the algorithm it finds a postman walk that is no longer than W'. □

10 Conclusion

We have discussed a few topics in Graph Theory that were chosen to illustrate both the mathematical theory and the algorithms that can be used for efficient solutions. The subject also contains many beautiful mathematical theorems that are not necessarily related to any practical applications, some of which will appear in the Part B Graph Theory course.

For most of the problems we considered, we were able to find the optimum solution efficiently. However, there are many important problems for which it is believed that this is impossible. An example of this is the question of whether a graph is Hamiltonian, which has no polynomial-time solution provided the conjecture "$P \neq NP$" is true. There are many other problems that have no efficient solution if $P \neq NP$. One of the best known examples is the Travelling Salesman Problem (see Q9 and Q10 of Problem Sheet 2). For such problems we may be happy if there is an efficient 'approximation algorithm', which finds a solution that is approximately optimal.