Part A Graph Theory

DRAFT NOTES

Trinity Term 2025, 8 lectures

Oliver Riordan

Last updated: 16/5/2025

These notes are to accompany the lectures in TT 2025 on the Part A short option Graph Theory. They are in rough form, may contain errors[1], and are *not for distribution*. I will update the notes shortly after (or maybe sometimes before) the lectures, possibly stopping at some point when there are no further significant changes from previous years.

You need to add figures!

You may also find the 2023 notes (Marc Lackenby) or 2024 notes (Richard Earl) useful. The former are more compact, the latter expanded with figures etc (not as many as in lectures). No significant changes of content are planned (the syllabus is unchanged), but especially near the start of the course there will be some minor changes in content, in the order of material, and in notation/definitions.

1 Introduction

We generally think of graphs as representing some notion of pairwise direct connections between objects. Formally the definition is as follows:

A *graph* G is an ordered pair (V, E), where V is a non-empty finite[2] set and E is a set of 2-element subsets of V. The elements of V are called the *vertices* of G and the elements of E the *edges* of G. We define $V(G)$ to be V, the *vertex set* of G, and $E(G)$ to be E, the *edge set* of E.

We often write xy for an edge $\{x, y\}$ (so xy means the same as yx). We say that x and y are *adjacent* in G, or *neighbours*, if xy is an edge of G. Graphically, we represent vertices as points (or more often blobs) and edges as lines or curves joining pairs of points (blobs); how a graph is drawn is irrelevant as far as the structure of the graph itself is concerned: the graph is defined by V and E, the set of vertices and the set of edges. The reason for using blobs is that it makes clear in the drawing where the vertices are: we may have to draw the lines/curves for two edges so that they cross even though the edges do not share a vertex.

Often, graphs represent physical or abstract networks, but they don't have to. Graphs can encode any yes/no relationship between pairs of objects (as long as it is

[1] If you find any errors, please first check the website to see if the error has already been corrected, and if not, e-mail riordan@maths.ox.ac.uk.

[2] In this course; there is a theory of infinite graphs also.

symmetric). For example taking $V = \{1, 2, \ldots, n\}$ and $xy \in E$ if and only if x and y are coprime defines a graph.

Graphs G and H are *isomorphic* if there exists a bijection $\varphi : V(G) \to V(H)$ such that, for each $x, y \in V(G)$, we have $xy \in E(G)$ iff $\varphi(x)\varphi(y) \in E(H)$. Often we do not make a distinction between isomorphic graphs, treating them as the same.

A graph H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

For an edge e, we write $G - e$ [3] for the subgraph $(V, E \setminus \{e\})$, obtained by deleting the edge e from G.

For a vertex v, we write $G - v$ for the subgraph obtained from G by deleting v and (as we must) all edges incident with v.

Examples

The *complete graph* on n vertices is K_n, $n \geqslant 1$, with vertex set $V = \{1, 2, \ldots, n\}$ and E the set of all $\binom{n}{2}$ possible edges.

The *empty graph* on n vertices is E_n, $n \geqslant 1$, with n vertices and no edges.

The *path* of length n, for $n \geqslant 0$, has $V = \{0, 1, \ldots, n\}$ and $E = \{01, 12, \ldots, (n-1)n\}$. Note that a path of length n has $n + 1$ vertices and n edges. Some people denote it by P_n, others by P_{n+1}, so best to say 'path of length n'. (I prefer P_n.) Note that a single vertex forms a path of length 0.

Finally, the *cycle* C_n of length n, for $n \geqslant 3$, has $V = \{1, 2, \ldots, n\}$ and $E = \{12, 23, \ldots, (n-1)n, n1\}$.

Paths, cycles and walks in graphs

A *walk* W in a graph G is a sequence $v_0 v_1 \cdots v_t$ of (not necessarily distinct) vertices of G such that $v_i v_{i+1} \in E$ for each $i = 0, 1, \ldots, t-1$. The *length* of a walk is the number of steps, here t. A *path* in G is a walk in which $v_0, \ldots, v_t$ are distinct. This is essentially[4] equivalent to a subgraph of G which is (isomorphic to) a path.

If $x = v_0$ and $y = v_t$ then we speak of a walk from x to y, or an x–y walk; an x–y path is defined similarly. A walk $v_0 v_1 \cdots v_t$ is *closed* if $v_t = v_0$.

A *cycle of length t in G* is a subgraph isomorphic to C_t. We usually just list the vertices to describe a cycle. Thus $v_1 v_2 \cdots v_t$ is a *cycle* in G if and only if $t \geqslant 3$, $v_1, \ldots, v_t$ are distinct vertices of G, and $v_1 v_2, \ldots, v_{t-1} v_t, v_t v_1$ are edges of G. A graph is *acyclic* if it contains no cycles.

Lemma 1.1. *Let G be a graph and $x, y \in V(G)$. Then G contains an x–y walk if and only if G contains an x–y path.*

[3] Some people write $G \setminus e$ for $G - e$. I will avoid this as it looks too much like G/e, which means something completely different.

[4] The two definitions of path in G are not quite the same: for existence, they are equivalent, but for counting paths, they differ by a factor of 2 for $t \geqslant 1$. A similar comment applies to cycles with a different factor.

In other words, if we want to get from x to y, then allowing ourselves to revisit vertices does not help. This simple observation is useful, allowing us to switch back and forth between using paths and walks to define connectedness, at any point using whichever definition is easiest to work with.

Proof. Any path is a walk, so one direction is trivial. For the converse, suppose that G contains an x–y walk, and let $v_0 \cdots v_t$ be a *shortest* x–y walk in G. If $v_i = v_j$ for some $i < j$, we can construct a shorter x–y walk, namely $v_0 \cdots v_i v_{j+1} \cdots v_t$ (or just $v_0 \dots v_i$ if $j = t$). Therefore all the vertices in the walk are distinct, and hence it is in fact a path. □

Remark. This trick of considering a *shortest* or in general minimal or sometimes maximal structure in a graph is very useful, especially in making sure that the structure 'looks like the (simple) picture', which isn't always necessarily the case. E.g., walks can be very complicated.

Two vertices x, y of G are *connected* (in G) if G contains an x–y walk/path. A graph G is *connected* if this holds for all $x, y \in V(G)$.

Note that $x \sim y$ if x and y are connected in G defines an equivalence relation (use walks to see this most simply). The equivalence classes partition $V(G)$, and so divide G up into disjoint connected subgraphs, called the *components* of G.

2 Trees

A *tree* is simply an acyclic connected graph.[5]

A *minimal connected graph* is a graph G such that G is connected, but $G - e$ is not connected for any $e \in E(G)$. In other words, keeping the same set of vertices, all the edges are required for connectivity.

Lemma 2.1. *G is a tree if and only if G is a minimal connected graph.*

Proof. For the forward implication, suppose T is a tree. Then T is connected by definition. If $T - xy$ were connected for some edge xy of T then there would exist an x–y path $x = v_0 v_1 \cdots v_t$ in $T - xy$. Since $t \geqslant 2$ (we can't use the edge xy) then $v_0 v_1 \cdots v_t$ forms a cycle in T, contradicting T being a acyclic.

Conversely, suppose that T is a minimal connected graph. If T contains a cycle $v_1 v_2 \cdots v_t$, then removing the edge $e = v_1 v_t$ from T leaves $T - e$ connected: for any two vertices x and y, there is an x–y path in T, and if that path uses e, we can replace it with $v_1 v_2 \cdots v_t$ or $v_t \cdots v_2 v_1$ to obtain an x–y walk $T - e$. So $T - e$ is connected, a contradiction to T being minimal connected. Thus T is acyclic and hence a tree. □

[5]This is the most standard definition. In previous years, the lecturer used minimal connected graph instead; it doesn't matter as the two are equivalent.

Remark. More generally, the second argument above shows that deleting an edge in a cycle cannot affect connectivity.

Another basic fact about trees is that within them, any two vertices are joined by a unique path.

Lemma 2.2. *Let x and y be vertices of a tree T. Then T contains a unique x–y path.*

Proof. Exercise! A (brief) proof is in the older notes, but do try it yourself first. And if you do look up the proof, expand it and fill in the details. □

In much of the following, unless otherwise indicated, the implicitly assumed setting is an arbitrary graph $G = (V, E)$.

A vertex w is a *neighbour* of v if v and w are adjacent, i.e., $vw \in E$. The *neighbourhood* of v is $\Gamma(v) = \Gamma_G(v) = \{w \in V : vw \in E\}$, i.e., the set of neighbours of v. The *degree* of a vertex v is $d(v) = |\Gamma(v)|$, i.e., the number of neighbours of v. Or, equivalently, the number of edges incident to v. We write $d_G(v)$ if we want to specify the graph.

A *leaf* in a graph G simply means a vertex v with $d(v) = 1$. A basic fact is that trees (with one silly exception) have leaves.

Lemma 2.3. *Let T be a tree with at least two vertices. Then T has at least two leaves.*

Proof. Let $P = v_0 v_1 \cdots v_t$ be a *longest* path in T, which exists because T is finite. Note that $t \geqslant 1$ since T has at least one edge. Suppose that v_0 has a neighbour $x \notin \{v_0, \ldots, v_t\}$. Then $x v_0 \cdots v_t$ would be a longer path, a contradiction. Suppose instead that v_0 has a neighbour v_i on P, where $i \geqslant 2$. Then $v_0 \cdots v_i$ would form a cycle in T, a contradiction. Thus v_0 has exactly one neighbour, namely v_1, so v_0 is a leaf. Similarly $v_t \neq v_0$ is a leaf. □

Notation. If G is a graph we write $|G|$ (or $v(G)$) for $|V(G)|$, i.e., the number of vertices of G. Write write $e(G)$ for $|E(G)|$, the number of edges.

Recall that we write $G - v$ for the graph formed from G by deleting the vertex v and also any edges containing v. Thus $e(G - v) = e(G) - d(v)$. A key property of leaves in trees is the following.

Lemma 2.4. *Let v be a leaf of a tree T. Then $T - v$ is a tree.*

Proof. Since T is acyclic, so is any subgraph of T, in particular, $T - v$. We must show that $T - v$ is connected. So let $x, y \in V(T - v)$. Since T is connected, T contains an x–y path $x = v_0 v_1 \cdots v_t$. Now v is not an endpoint of this path (since $x, y \neq v$). But it also is not an interior vertex, i.e., v_i for $1 \leqslant i \leqslant t - 1$, because otherwise it would have two *distinct* neighbours in T, namely v_{i-1} and v_{i+1}. So P is also a path in $T - v$, and $T - v$ is indeed connected. □

The above lemma is extremely important for induction on trees. Note the lemma works by *shrinking* T, not by growing it. This is exactly what we want for induction: when we prove something about objects of size n by induction, the proof always involves forming a smaller object from a larger one. Indeed, we have to start with an arbitrary large object, and are allowed to assume the result for smaller objects.

Lemma 2.5. *Let T be a tree with $|T| = n$. Then $e(T) = n - 1$.*

Proof. Induction on n. The base case is $n = 1$, when T has no edges. So suppose $|T| = n \geqslant 2$, and the result holds for smaller trees. Then T has a leaf v, and $T - v$ is a tree with $n - 1$ vertices. So by induction $T - v$ has $n - 2$ edges. But we deleted exactly one edge, so T has $n - 1$ edges. □

A *spanning subgraph* of a graph G is a subgraph H with $V(H) = V(G)$, i.e., including all the vertices. A *spanning tree* of G is a spanning subgraph which is a tree.

Observation. Any connected graph G has at least one spanning tree. Indeed, we can just remove edges one by one keeping the graph connected until there is no edge that can be removed. What remains is minimal connected, so a tree.

Lemma 2.6. *Suppose $|G| = n$. Then G is a tree if and only if G is connected and $e(G) = n - 1$.*

Proof. If G is a tree then it is connected by definition, and has $n - 1$ edges by Lemma 2.5. Conversely, suppose G is connected and $e(G) = n - 1$. Since G is connected, it has a spanning tree T. T is a tree with n vertices, so it has $n - 1$ edges. Thus $T = G$: it is a subgraph with all the vertices, and $n - 1$ out of $n - 1$ of the edges of G. So G is a tree. □

Notation. We write $G + e$ for the graph formed from G by adding the edge e, *whenever this makes sense*. Thus if $G = (V, E)$, then $G + e = (V, E \cup \{e\})$, and we must have that $e = xy$ for distinct vertices x, y of G, and that $e \notin E(G)$.

A *maximal acyclic* graph G is an acyclic graph G such that there does not exist e with $G + e$ acyclic, i.e., it is impossible to add an edge (keeping the same vertex set) while preserving acyclicity.

Exercise. *Show that G is a tree if and only if G is maximal acyclic.*

Exercise. *Let G be a graph with n vertices. Show that G is a tree if and only if G is acyclic and $e(G) = n - 1$*

These results together with what we already show that any two out of 'connected', 'acyclic' and 'exactly $n - 1$ edges' imply the third.

3 Minimum Cost Spanning Trees

Suppose we wish to build a rail network connecting all the cities in a country, by building a number of direct routes between pairs of cities. Not every route is plausible, so we may form a graph G with V the set of cities, and xy an edge if it is plausible to build a direct route between x and y. Of course, each such route will have a *cost*. The question is how to minimize the total cost while connecting all the cities (ignoring travel times and other considerations, just saving money).

Mathematically, we can phrase this question as follows. Let G be a graph with a *cost function* c on the edges, i.e., a real number $c(e) > 0$ for each edge e of G. (These costs do not have to be distinct.) We would like to find a spanning subgraph H with minimum cost, where $c(H) = \sum_{e \in E(H)} c(e)$.

Of course, any such cheapest H will be *minimally* connected, i.e., H will be a spanning tree. This motivates the following definition.

Definition. Let $G = (V, E)$ be a connected graph with a cost function $c : E \to (0, \infty)$. A *minimum cost spanning tree*, or *MCST* in G is a spanning tree T such that $c(T) = \sum_{e \in E(T)} c(e)$ is minimum.

Every connected graph has at least one spanning tree and hence (by finiteness) at least one MCST. But how do we find one? One approach is just to consider all possible spanning trees, but in general there can be many of these, in fact even more than exponentially many.[6] Instead we will try to find a MCST by a *greedy algorithm*, an algorithm that proceeds step by step, in each step making the 'locally best' choice. In general there is no reason that such an algorithm should work, and in this case it is far from obvious, and will be our first substantial result.

There is more than one possible greedy algorithm to consider. Perhaps the most natural is the following: starting with all the vertices and no edges, just add edges of G one by one always choosing the cheapest new edge that keeps the current graph acyclic, until we can't add any more. Here is a formal description.

Kruskal's Algorithm. Let $G = (V, E)$ be a graph with a cost function $c : E \to (0, \infty)$. Start with $H_0 = (V, \emptyset)$. Given H_i, if there is any edge e of G not included in H_i such that $H_i + e$ is acyclic, then let e be a cheapest such edge, set $H_{i+1} = H_i + e$, and continue. Otherwise, stop, writing H_t for the final graph H.

Theorem 3.1. *Let G be a connected graph with a positive cost $c(e)$ for each edge. Then Kruskal's Algorithm ends with a MCST for G.*

In the course of the proof we will use the simple observation that adding an edge $e = xy$ to a graph H creates a cycle if and only if H contains an x–y path.

[6] By *Cayley's Formula*, the complete graph K_n has n^{n-2} spanning trees, though this is not obvious and we will not prove it in this course; for a proof see the Part B notes.

Proof. First we claim that H_t is a spanning tree of G. It spanning since $V(H_t) = V(H_0) = V$. It is acyclic by construction. Suppose H_t is not connected, and let C be a component of H_t. Then C does not contain all vertices, so we can pick a vertex v inside C and a vertex w outside. Now G contains a v–w path, and at some point this path must leave C. So there is an edge $e = xy$ of G with x inside C and y outside. Thus x and y are not connected in H_t, and $H_t + e$ is acyclic. This contradicts the stopping rule in the algorithm. Hence H_t is connected, and is indeed a spanning tree. In particular $t = n - 1$, where $n = |G|$, since trees have $n - 1$ edges.

It remains to show that H_{n-1} is a MCST. Let $\mathcal{P}_i$ be the proposition that there is *some* MCST T of G with $H_i \subseteq T$. We shall show that $\mathcal{P}_i$ holds for all $0 \leqslant i \leqslant n - 1$ by induction on i. Certainly $\mathcal{P}_0$ holds, since H_0 has no edges. If $\mathcal{P}_{n-1}$ holds, then $H_{n-1} \subseteq T$ but both have $n - 1$ edges, so $H_{n-1} = T$ and H_{n-1} is a MCST as required. So it remains to prove the induction step.

Suppose that $i < n - 1$, and that $\mathcal{P}_i$ holds, so $H_i \subseteq T$ for a MCST T of G. Let e be the next edge added by the algorithm, so $H_{i+1} = H_i + e$. If e is an edge of T then $H_{i+1} \subseteq T$ and we are done. So suppose not.

Let $e = xy$. Then T contains a (unique) x–y path, so $T + e$ contains a (unique) cycle C. Since H_{i+1} is acyclic, there is some edge f of H not contained in H_{i+1}. Let $T' = T + e - f$. To establish $\mathcal{P}_{i+1}$ we will show that $H_{i+1} \subseteq T'$, and that T' is a MCST of G.

Now $H_{i+1} \subseteq T'$ by construction: $H_{i+1} = H_i + e \subset T + e$, and the edge f we remove is not in H_{i+1}. To show that T' is a tree observe that it is obtained from the connected graph $T + e$ by deleting an edge in a cycle, so it is definitely connected. It has $n-1$ edges, so by Lemma 2.6 it is a tree. So T' is indeed a spanning tree of G. To see that T' has minimum cost, note that $H_i + f \subseteq T$ (since $H_i \subset T$ and f is an edge of $C \subset T + e$ distinct from e) which is a tree and so acyclic. Thus f was a possible edge to add at step i, and by the definition of the algorithm $c(e) \leqslant c(f)$. Hence $c(T') \leqslant c(T)$ and as T has minimum cost, so does T', completing the rpoof. □

How fast is Kruskal's algorithm? Making this mathematically precise would take us far afield (we would need to define a model of computation). Instead, we will take an intuitive approach, estimating the number of 'steps' taken by an algorithm, where a 'step' should be a 'simple' operation.

If G has n vertices and m edges, then in each iteration of the algorithm we add one edge, so there are $n - 1$ iterations. If at each stage of the algorithm, we naively find the next edge by checking every edge then there will be m steps in each iteration, giving about nm steps in total.[7]

We say that the running time is $O(nm)$, where the 'big O' notation means that there a positive constant C so that for any graph G the running time is at most Cnm,

[7] I'm cheating a bit here, because we should also think about how many simple operations it takes to check whether adding the edge forms a cycle. This can be done quite quickly, however.

where $n = |G|$ and $m = e(G)$. 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' with the number of edges, which is essentially the best possible.

At a crude level, the key thing is that the running time in *poynomial* (in the size of the input), meaning bounded by Cm^k (or Cn^k; it doesn't matter) for some constants C and k. This is in sharp contrast to the exhaustive search algorithm that tries every possible spanning tree.

Remark. There is a variant of Kruskal's Algorithm called *Prim's Algorithm*. This builds a subtree of T working out from an arbitrary vertex v, each time adding the cheapest edge of G connecting any vertex currently included in T to any vertex not yet included. This alternative greedy algorithm also constructs a MCST of G.

4 Euler Tours

One of the most famous questions in graph theory concerns the *bridges of Königsberg*. At that time (1736) there were seven bridges in the city, and a puzzle was whether it is possible to walk around the city, ending back where you started, crossing each bridge exactly once. This can easily be abstracted to a graph theory question, asking whether a certain (multi)-graph contains a closed walk using each edge exactly once. Here we will stick to simple graphs, which makes no difference (by subdividing any repeated edges).

Definition. An *Euler trail* in a graph G is a walk $W = v_0 \cdots v_t$ in G which uses each edge of G exactly once. We call W an *Euler tour* or *Euler circuit* if in addition $v_0 = v_t$ (as in the original question above).

An *isolated vertex* in G is a vertex v with $d_G(v) = 0$.

Suppose G has an Euler tour. What can we say about G? Firstly, *after deleting any isolated vertices*, G is connected (because the tour now visits all vertices). But also, each time W visits a vertex $v \neq v_0, v_t$ it uses two edges incident with V, one to enter and one to leave. This is also true of $v = v_0 v_t$, thinking of one visit which enters v at the end of W and leaves it at the start. (Or consider the walk written around a circle, rather than in a line; there is nothing special about the starting vertex then.) Since every edge is used exactly once, every vertex has even degree. Such a graph is sometimes called *Eulerian*.

Theorem 4.1. *Let G be a connected graph. Then G has an Euler tour if and only if $d_G(v)$ is even for every vertex v of G*

This result is sometimes attributed to Euler in 1736, but if you look at his paper, he only really proved the 'easy' part above. The other direction is not too hard, however. There are different ways to show it (including a fairly simple inductive proof). We will give an algorithmic proof due to Fleury.

Fleury's Algorithm. We construct a walk $v_0v_1\cdots$ in G step by step as follows. Start with v_0 an arbitrary vertex, and let $H = G$. Each time we add a step v_iv_{i+1} to the walk, we delete the edge v_iv_{i+1} from H, and if v_i then becomes isolated, we also delete v_i. Choose the next vertex to add as follows: if v_i is isolated in H, stop. If not, pick any neighbour v_{i+1} of v_i in H such that, after the update, H is still connected; if there is no such neighbour, the algorithm fails.

In other words, we build the walk, picking up the track behind us that does not need to be revisited, which includes any vertex we have already left which has no further edges. So H is the graph of what we still have to visit in the walk. It is not obvious that this algorithm works, but we will show that it does. First we make an observation.

Observation. Let $W = v_0v_1\cdots v_t$ be a walk. For a vertex v let $d_W(v)$ be the number of edges in the walk (counted with multiplicity if there are repeats) incident with v. Then $d_W(v)$ is even, unless $v_0 \neq v_t$ and $v \in \{v_0, v_t\}$, in which case $d_W(v)$ is odd.

This is a slight extension of the comment above about Euler tours. We see that $d_W(v)$ counts two for every $1 \leqslant j \leqslant t-1$ such that $v_j = v$ (the edges $v_{j-1}v_j$ and v_jv_{j+1}), and one if $v_0 = v$, and one if $v_t = v$.

We also need perhaps the most basic (but important) result in graph theory.

Lemma 4.2. *Let G be a graph. Then $\sum_{v\in V(G)} d(v) = 2e(G)$.*

Proof. Each edge $e = xy$ contributes one to the degree of each of its ends, in this case x and y. □

Corollary 4.3. *In any graph G, the number of vertices with odd degree is even.*

Proof. The sum of the degrees is zero mod 2 by Lemma 4.2. □

This (either the lemma or the corollary) is sometimes called the *handshaking lemma*, since it solves the following puzzle: at a party, some of the participants shake hands with each other. Must the number of people who have shaken hands with an odd number of people be even?

With these easy preliminaries out of the way, we proof 'Euler's Theorem'.

Proof of Theorem 4.1. If G has an Euler tour $v_0 \cdots v_t$ then (recalling that $v_0 = v_t$ by definition), by the observation above all degrees are even.

Conversely, suppose that G is connected and all degrees are even. We run Fleury's algorithm, updating the graph H of remaining edges as we go. Note that initially $H = G$ is connected, and that by definition of the algorithm, after every step H remains connected.

We delete one edge in every step, so the algorithm certainly stops at some point, say having built the walk $v_0 \cdots v_i$. We consider the following cases.

1) $d_H(v_i) = 0$. Then, since H is connected, H has one vertex and no edges, so $W = v_0 \cdots v_i$ includes all edges of G and is an Euler *trail*. If $v_0 \neq v_i$ then by the observation above, v_0 and v_i would have odd degree in G, a contradiction. Thus W is an Euler tour as required.

2) $d_H(v_i) = 1$. Then v_i is a leaf of H, so $H - v_i$ is connected. But this means that the walk can continue to the unique neighbour of v_{i+1} of v_i, and the algorithm would not have stopped.

3) $d_H(v_i) = d \geqslant 2$. If v_i has a neighbour w in H such that $H - v_i w$ is connected, then the walk could have continued to w. Thus, for every neighbour w, $H - v_i w$ is disconnected. It follows that $H - v_i$ has d components $C_1, \ldots, C_d$, and that v_i has exactly one neighbour w_j in each. (Draw a picture!) With $W = v_0 \cdots v_i$ the walk we have followed so far, for any $v \notin \{v_0, v_i\}$ we have $d_W(v)$ even and hence (since $d_G(v)$ is even), $d_H(v)$ is even. Let C be one of the C_j not containing v_0 (which exists since $d \geqslant 2$). Then all vertices of C have even degree *in* H. But exactly one vertex (the relevant w_j) has a neighbour outside C. So within the graph C, all vertices have even degree except for w_j, which has odd degree. This contradicts Corollary 4.3.

We see that the only case not leading to a contradiction is 1), in which case we obtain the required Euler tour. □

We note that the running time of this algorithm is polynomial: there are $m = e(G)$ steps in building the walk, and in each step we only have to check whether certain graphs (at most $d(v) \leqslant n - 1$) are connected or not, something that can be done quickly.

5 Hamilton cycles

A *Hamilton cycle* in a graph G is simply a cycle $C \subseteq G$ including all the vertices, so $|C| = |G|$. A graph G is *Hamiltonian* if it contains a Hamilton cycle.

Here are two superficially similar questions:

Q1) given a graph G, does G contain an Euler tour, i.e., a closed walk using each edge of G exactly once?

Q2) given a graph G, does G contain a Hamilton cycle, i.e., a closed walk using each vertex of G exactly once?

We have seen that Q1 is 'easy': there is a polynomial time algorithm to decide whether the answer is yes or no. (Indeed, we simply check connectivity and the vertex degrees and use Theorem 4.1.) On the other hand Q2 appears to be 'hard' – there is no known polynomial time algorithm, and it seems very unlikely that there is one.[8] Given this, we can't hope for a simple necessary and sufficient condition. So we will look just for sufficient conditions.

Given a graph G, let $\delta(G)$ denote its *minimum degree*, i.e., $\min\{d(v) : v \in V(G)\}$, and similarly $\Delta(G)$ its *maximum degree*.

Theorem 5.1 (Dirac)**.** *If $|G| = n \geqslant 3$ and $\delta(G) \geqslant n/2$ then G is Hamiltonian.*

Note that $n/2$ is 'sharp' here, in that if n is even, then the disjoint union of two copies of $K_{n/2}$ is a non-Hamiltonian graph with $\delta(G) = n/2 - 1$. Of course, *some* graphs with smaller minimum degree are Hamiltonian, but not all.

There is a slight strengthening of Dirac's Theorem, due to Ore.

Theorem 5.2 (Ore)**.** *Suppose that $|G| = n \geqslant 3$ and every pair x, y of distinct non-adjacent vertices of G satisfies $d(x) + d(y) \geqslant n$. Then G is Hamiltonian.*

Note that Theorem 5.2 implies Theorem 5.1.

The rough plan of the proof is to show that given a path P in G, we can either find a longer path, or a cycle C with $|C| = |P|$ and $e(C) = e(P) + 1$, and that (if the graph is not Hamiltonian), given a cycle C in G we can find a path with $|P| = |C| + 1$ and $e(P) = e(C)$. This will lead to a contradiction.

We start with the easy part, going from a cycle to a path, assuming connectivity.

Lemma 5.3. *Let G be a connected non-Hamiltonian graph. Then the length of the longest path in G is at least the length of the longest cycle.*

Proof. Let C be a longest cycle, with length ℓ, noting that $\ell < n$ by assumption. Since G is connected there is some $xy \in E(G)$ with x a vertex of C and y a vertex not on C. But then C and the edge xy together contain a path of length ℓ. □

[8] For those interested: we write P for the class of 'decision problems' (where there is some input, here a graph G, and the output is always yes or no) for which there is a polynomial time algorithm. We write NP for, roughly speaking, the class of decision problems where there is a polynomial time algorithm for *verifying* a proposed solution. This includes all the problems we consider here, in particular whether H is Hamiltonian – we just check whether the vertices of a proposed Hamilton cycle match those of G, the edges are all present. It is an extremely important open question whether in fact $P = NP$, i.e., all NP problems can be solved in polynomial time. Most mathematicians firmly believe that the answer is no. Within NP there is large subclass of problems called 'NP-complete', meaning that if you can solve (find a polynomial time algorithm for) any one such problem, then you can find one for any problem in NP, so $P = NP$. Q2 is an example of such a problem.

We are now ready to prove Ore's Theorem (and hence Dirac's Theorem).

Proof of Theorem 5.2. First we show that G is connected. If not, there are vertices x and y not joined by a path in G. But then x and y are certainly distinct and non-adjacent, so $d(x) + d(y) \geqslant n$. The neighbours of x and of y all lie in $V(G) \setminus \{x, y\}$, a set of size $n - 2 < n$. Thus there is at least one vertex w (in fact at least two) that is a neighour of both. So x and y are joined by the path xwy.

Suppose that G is not Hamiltonian. Let $P = v_0 v_1 \cdots v_\ell$ be a longest path in G. Note that $\ell \geqslant 2$ since G is connected and $|G| \geqslant 3$.

If G contains a cycle of length $\ell + 1$ then either G is Hamiltonian, or Lemma 5.3 applies giving a path of length at least $\ell + 1$, a contradiction. So G contains no cycle of length $\ell + 1$, and in particular $v_0 v_\ell \notin E(G)$.

By assumption $d(v_0) + d(v_\ell) \geqslant n$. Since P is a longest path, all neighbours of v_0 and of v_ℓ are on P. Let $A = \{i : v_0 v_i \in E(G)\}$, and let $B = \{i : v_{i-1} v_\ell \in E(G)\}$. Clearly $|A| = d(v_0)$. Also, $|B| = d(v_\ell)$, since there is one element $(j + 1)$ for each j such that v_j is a neighbour of v_ℓ. So $|A| + |B| \geqslant n$.

Now A and B are subsets of $\{1, 2, \ldots, \ell\}$, a set of size $\ell < n$ (recall that P has $\ell + 1$ vertices). Thus they cannot be disjoint, and there is some i such that $v_0 v_i$ and $v_{i-1} v_\ell$ are edges of G. But now $v_0 v_i v_{i+1} \cdots v_\ell v_{i-1} v_{i-2} \cdots v_1$ is a cycle in G with $\ell + 1$ vertices, a contradiction. □

A question for you to think about: is the proof above algorithmic? We can't hope for a (polynomial time) algorithm for finding a Hamilton cycle in any graph that has one, but that is not the situation here: we are looking at a very restricted class of graphs, namely those with $\delta(G) \geqslant n/2$.

6 Shortest Paths

Let G be a graph. For x, y vertices of G, the *graph distance* $d_G(x, y)$ between x and y is simply the length of a shortest x–y path in G, or ∞ if there is no such path.

Certainly $d(x, x) = 0$ and $d(x, y) > 0$ if $x \neq y$. Also, $d(x, y) = d(y, x)$. Furthermore, d_G satisfies the triangle inequality: $d_G(x, z) \leqslant d_G(x, y) + d_G(y, z)$, since (if the right-hand side is finite) there are x–y and y–z paths of lengths $d_G(x, y)$ and $d_G(y, z)$, and combining them gives an x–z *walk* of length $d_G(x, y) + d_G(y, z)$. But the shortest x–z walk is a path, so there is an x–z path of at most this length. Thus d_G is a metric on V_G. Although we will not use this here, in general these examples $(V(G), d_G)$ are an important class of metric spaces.

How do we actually find the graph distance between x and y? The answer is simple: working outwards from x we can easily find the graph distance from x to every other vertex: x itself is the unique vertex at distance 0, the neighbours of x are

at distance 1. A vertex v is at distance 2 if and only if v is a neighbour of some w at distance 1, and v itself is not at distance 0 or 1.

More generally, $d_G(x,v) = d+1$ if and only if (i) v has a neighbour w with $d_G(x,w) = d$ and (ii) it is not the case that $d_G(x,v) \leqslant d$. This gives us an easy algorithm: start with $D(x) = 0$ and $D(y)$ 'unassigned' for all other vertices. In round $d \geqslant 0$, for each vertex v with $D(y) = d$, for each neighbour w of v with $D(w)$ not assigned, set $D(w) = d+1$. The algorithm stops if in the entire round no vertex is assigned $d+1$. At the end, $D(v) = d_G(x,v)$, or is unassigned if there is no x–v path.

In particular, this algorithm efficiently finds all vertices reachable from x (the component containing x), and so tests whether G is connected.

What about the following more general question: let G be a connected (for simplicity) graph in which each edge e has a *length* $\ell(e) > 0$. We will call this a *length function*.[9] For a path P in G, the *ℓ-length* of P is just $\ell(P) = \sum_{e \in E(P)} \ell(e)$. We write $d_\ell(x,y)$ for $\min\{\ell(P) : P \text{ is an } x\text{–}y \text{ path}\}$, and say that P is an *ℓ-shortest* x–y path if it achieves the minimum.

It's easy to see that d_ℓ is a metric on $V(G)$. How do we find $d_\ell(x,y)$ for two vertices of G? The answer is Dijkstra's Algorithm. To understand this algorithm I personally think about a huge number of ants that start at vertex x at time zero and wander along all possible edges at rate 1. R will be the set of vertices the ants have already reached. For each $v \in U$, $D(v)$ will be a 'projected ant arrival time' at v, based on ants that are already crawling along an edge to v.

Dijkstra's Algorithm. Initially let $R = \emptyset$ and $U = V(G)$. Let $D(x) = 0$ and for each vertex $y \neq x$ let $D(y) = \infty$.

While there is a vertex $v \in U$ with $D(v)$ finite, do the following: pick such a vertex v with $D(v)$ minimal. Move v from U to R, and then for each neighbour w of v which is in U, if $D(v) + \ell(vw) < D(w)$, then set $D(w) = D(v) + \ell(vw)$.

When we change $D(w)$ as above, we say that v *updates* w.

Theorem 6.1. *Let G be a connected graph in which each edge has a positive length $\ell(e)$, and let $x \in V(G)$. Then after running Dijkstra's algorithm we have $D(v) = d_\ell(x,v)$ for every $v \in V(G)$.*

Proof. First note that during the algorithm, $D(v)$ only ever decreases. Moreover, if/when v is moved from U to R, than at this point $D(v)$ is finite, and from this point on $D(v)$ does not change.

We first claim at at the end of the algorithm $U = \emptyset$. Suppose not. Then (since $x \in R$) neither U nor R is empty, so these complementary sets partition $V(G)$. Since G is connected, there is an edge yy' of G with $y \in R$ and $y' \in U$. But then at the

[9] It's the same as a cost function; we just use a different word to guide our intuition in this different context.

time y was moved to R we have that $D(y)$ was finite, and $D(y')$ (if not already finite) would have been updated to a finite value. So the final value of $D(y')$ is finite. But then the algorithm would not have stopped.

We now claim that for any vertex v, when v is added to R, then $D(v) = d_\ell(x, v)$; this implies the result since $D(v)$ does not then change. We show this step-by-step, considering the vertices in the order they are added to R.

Firstly, x is added right at the beginning, and $D(x) = 0 = d_\ell(x, x)$.

So now suppose we are about to add $v \neq x$ to R, and that for all y currently in R we do have $D(y) = d_\ell(x, y)$. Since $D(v)$ is finite, at least one other vertex has updated v. Let y be the *last* vertex to have done so. Then $y \in R$, and we have

$$D(v) = D(y) + \ell(yv) = d_\ell(x, y) + \ell(yv) \geqslant d_\ell(x, v).$$

To see the reverse inequality, let P be an x–v path with $\ell(P)$ minimal. Since $x \in R$ and $v \in U$, going along P in order there is some pair of consecutive vertices yy' with $y \in R$ and $y' \in U$. Let P' be the initial segment of P up to y'. Then

$$\ell(P) \geqslant \ell(P') \geqslant d_\ell(x, y) + \ell(yy') = D(y) + \ell(yy') \geqslant D(y'),$$

where the equality is because $y \in R$ (i.e., our inductive assumption), and the last inequality is because when y was added to R it would have updated $D(y')$ for its neighbour $y' \in U$ to $D(y) + \ell(yy')$, unless $D(y')$ was already at most this. And after this update, $D(y')$ can only decrease.

By the choice of v, we have $D(v) \leqslant D(y')$, so it follows that $D(v) \leqslant \ell(P) = d_\ell(x, v)$, and hence $D(v) = d_\ell(x, v)$ as claimed. □

The main point of Dijkstra's algorithm is to find the distance between a given pair of vertices. It so happens that at the same time it finds the distances from one given vertex x to all others. More than that: it finds a tree structure within G containing efficient routes between x and all other vertices.

Definition. An *ℓ-shortest paths tree in G with root x* is a spanning tree T of G such that for each $v \in V(G)$ the unique x–v path in T has length $d_\ell(x, v)$.

When we run Dijkstra's algorithm every vertex $w \neq x$ is updated at least once (otherwise we would have $D(w) = \infty$ and it would remain in U). The *parent* of w is the *last* vertex v to update w. In particular, since $D(v)$ was already final at the point it updated w, and $D(w)$ is not updated again, if v is the parent of w then we have $D(w) = D(v) + \ell(vw)$ and hence, by Theorem 6.1,

$$d_\ell(x, w) = d_\ell(x, v) + \ell(vw). \tag{1}$$

Lemma 6.2. *Let T be the graph on $V(G)$ with an edge wv for each $w \neq x$, where v is the parent of w. Then T is an ℓ-shortest paths tree in T rooted at x.*

Proof. Let T_R correspond to T part way through the algorithm, so the vertex set is the current set R, and there is an edge wv from each $w \in R \setminus \{x\}$ to its parent. Note that $T_{\{x\}}$ is a tree. When we add a new vertex w to T_R its parent v is a vertex already present, so we add a new vertex and one edge to an existing vertex, and it follows that T_R remains a tree; hence T is a tree.

Let P_w denote the unique x–w path in T. If v is the parent of w, then P_w is just P_v with the edge vw added at the end. So $\ell(P_w) = \ell(P_v) + \ell(vw)$. It follows from (1), induction and the fact that $\ell(P_x) = 0 = d_\ell(x, x)$ that $\ell(P_v) = d_\ell(x, v)$ for all v. □

To comment briefly on complexity: Dijkstra's algorithm runs for $n = |G|$ steps (as each vertex is moved from U to R). Within each step we consider at most n vertices w to update, and choosing the next vertex takes at most n steps. This gives a running time of $O(n^2)$. This can (with complications that we will not be consider) be improved to something like $O(m + n \log n)$ where $m = e(G)$. So the algorithm is pretty efficient. For us all that matters is that it is polynomial, unlike 'consider all possible paths and find the shortest', for example.

7 Matchings and coverings

A *matching* in a graph G is just a set M of vertex-disjoint edges of G, which we often think of as a pairing up of *some* of the vertices of G. The matching is *perfect* if it includes all vertices of G.

Definition. A graph G is *bipartite* if we can write $V(G) = A \cup B$ where A and B are disjoint and every edge of G has one end in A and the other in B. In this case we say that (A, B) is a *bipartition* for G.

Note that a bipartite graph may have more than one bipartition. But often the bipartition is given in advance. For example, A may be a set of tasks that need completing, and B a set of people, and an edge ab may represent the information that person b is capable of doing task a. So it's not that we start with the graph and try to find A and B; rather we know the bipartition in advance from the situation, and the only possible edges that make sense are ones joining A to B.

In a bipartite graph G, each edge of a matching M pairs some $a \in A$ with some $b \in B$, so a perfect matching is only potentially possible if $|A| = |B|$. In general, it is an important question whether a perfect matching exists. More generally, we may ask when G contains a matching of size $|A|$, noting that every such matching necessarily includes every vertex in A. We call such a matching a *complete matching from A to B*, noting that when $|B| > |A|$, the same matching is not complete from B to A.

For $S \subset A$ let

$$N(S) = \{b \in B : \exists a \in S \text{ with } ab \in E(G)\}$$

be the *neighbourhood* of S, which is just $\bigcup_{a\in S}\Gamma(a)$. If G contains a complete matching from A to B, then clearly

$$|N(S)| \geqslant |S| \text{ for every } S \subset A.$$

Indeed the partners of the vertices in S are distinct vertices in $N(S)$. This condition is known as *Hall's Condition*. It turns out that this simple, trivially necessary condition is also sufficient.

Theorem 7.1 (Hall)**.** *Let G be a bipartite graph with bipartition (A, B). Then G contains a complete matching from A to B if and only if Hall's Condition holds in G.*

Proof. We have already shown that Hall's Condition is necessary. For sufficiency we argue by induction on $n = |A|$.

If $n = 1$, the result is trivial. For the induction step, let $n \geqslant 2$ and suppose that the result holds for all bipartite graphs with $|A| < n$. Consider a bipartite graph G with $|A| = n$ and assume that Hall's condition holds in G. There are two cases.

(a) Suppose first that $|N(S)| > |S|$ for each $\emptyset \neq S \subsetneq A$. Let xy be any edge of G with $x \in A$ and $y \in B$. Form G' by deleting the vertices x and y from G. Then G' satisfies Hall's condition (since if $\emptyset \neq S \subseteq A \setminus \{x\}$ then $|N'(S)| \geqslant |N(S)| - 1 \geqslant |S|$), and so by induction G' contains a complete matching from $A \setminus \{x\}$ to $B \setminus \{y\}$. Now adding the edge xy gives the required matching.

(b) If case (a) does not hold then $|N(S)| = |S|$ for some $\emptyset \neq S \subsetneq A$. The bipartite subgraph induced[10] by $S \cup N(S)$ still satisfies Hall's condition, so by induction there is a complete matching M_1 from S to $N(S)$.

Now consider $T = A \setminus S$ and $U = B \setminus N(S)$. We shall see that the bipartite subgraph H induced by $T \cup U$ also satisfies Hall's condition. For each $A \subseteq T$ we have

$$\begin{aligned} |N_H(A)| = |N(A) \cap U| = |N(A) \setminus N(S)| &= |N(A \cup S) \setminus N(S)| \\ &= |N(A \cup S)| - |N(S)| \geqslant |A \cup S| - |S| = |A|, \end{aligned}$$

since $|N(A \cup S)| \geqslant |A \cup S|$ and $|N(S)| = |S|$. So Hall's condition holds in H, and by induction there is a complete matching M_2 from T to U. Then $M_1 \cup M_2$ is the required complete matching from A to B. □

This proof is very nice, but from an algorithmic point of view, it is not very helpful, since there are $2^{|A|}$ subsets S to check to even decide which case we are in. Building matchings greedily also doesn't not work very well in general, so we need some other way of enlarging a matching. The key to this are the notions of alternating and augmenting paths.

[10] The subgraph H of G *induced* by W has vertex set W, and the edge set is all edges of G with both ends in W.

Definition. Let G be a graph, let M be matching in G, and let P be a path in G.

(a) We say P is *M-alternating* if every other edge of P is in M.

(b) We say P is *M-augmenting* if P has at least one edge, is M-alternating and its end vertices are not in any edge of M.

To spell out the meaning of (a), P is M-alternating if, as we go along P, the next edge is in M if and only if the previous edge is not in M. The first edge may or may not be in M.

For convenience we call a vertex *free* or *M-free* if it is not in any edge of M.

Lemma 7.2. *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. Suppose P is an M-augmenting path. Since the first and last vertices are free, the first and last edges are not in M, so P contains one more edge not in M than edges in M. Thus we can find a larger matching by 'flipping' P: for each edge of P, if it is in M remove it from M, and if it is not in M, add it to 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 a path (which might be a single edge) or a cycle. Except for single-edge components corresponding to an edge $e \in M \cap M^*$, the edges in any component alternate between M and M^*. In particular, any cycle has the same number of edges in M and in M^*. Since $|M^*| > |M|$, we can find a component with more edges of M^* than M (which might just be a single edge in $M^* \backslash M$); this is an M-augmenting path in G. □

Lemma 7.2 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. There is a general algorithm for this (Edmond's Algorithm), but it is quite tricky, so we will just consider the much easier bipartite case.

Let M be a matching in a bipartite graph with bipartition (A, B). Let A^* and B^* be the sets of M-free vertices in A and B, respectively. Any M-augmenting path P has odd length, so one end is in A, the other in B; the ends are free vertices by definition, so (after reversing P if necessary) P is an ab-path for some $a \in A^*$ and $b \in B^*$. As we follow P from a^*, we cross from A to B along an edge not in M. Then we cross from B to A along an edge in M, and so on. This gives the following alternate description of M-augmenting paths:

Put a direction on each edge of 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. Then an M-augmenting path is equivalent to a directed path from A^* to B^*, i.e., a path that respects directions of edges. How do we find such a path, or check that none exists? Using the following simple search algorithm; this works in any directed graph (where each edge has an orientation); not just bipartite ones.

Search algorithm: let G be a graph in which each edge is one-way, and let $S \subset V(G)$. Start with $R = S$. Repeat the following step: if there is any edge xy of G directed from some $x \in R$ to some $y \notin R$ then add y to R, otherwise stop.

Lemma 7.3. *let G be a graph in which each edge is one-way, and let $S \subset V(G)$. At the end of the search algorithm above, R is precisely the set of vertices which can be reached from S, i.e., $R = \{y : \exists$ a directed x–y path$\}$.*

Proof. It's easy to check by induction on when a vertex is added to R that $y \in R$ implies y reachable from S. On the other hand, if $x = v_0 \to v_1 \to \cdots v_t = y$ is a directed path with $x \in S$, then at the end of the algorithm $v_0 \in R$ (since it is in S), and $v_{i-1} \in R$ implies $v_i \in R$ (otherwise we would not have stopped). So $v_t \in R$. □

Note that during the search algorithm, we can remember the 'parent' x of each vertex y added to R, so when we add y to R then working backwards step-by-step (from y to its parent, to their parent etc) we can actually find a path from some $s \in S$ to y.

Putting the pieces together leads to the *Hungarian Algorithm* to find a largest possible mathcing in a bipartite graph. [Actually, Kuhn's Hungarian Algorithm is more complicated, and deals with the weighted case.]

Hungarian Algorithm: Let G be a bipartite graph with bipartition (A, B). Start with $M = \emptyset$. Repeat the following: let A^* and B^* be the M-free vertices in A and B, and orient the edges from G as above (from A to B if $e \notin M$ and from B to A if $e \in M$). Run the search algorithm with $S = A^*$. If $R \cap B^* = \emptyset$ then stop. Otherwise, there is some $y \in R \cap B^*$. Find a directed path from some $x \in A^*$ to y (from the search algorithm). This is an augmenting path, which we use to augment M to a larger matching, and repeat.

By the observation above concerning M-augmenting paths being equivalent to directed paths from A^* to B^*, and Lemma 7.2, when the algorithm stops M is a matching of maximum size.

What is the running time of this algorithm? We won't analyze it precisely, but when $|G| = n$ and $e(G) = m$, the main algorithm has at most $n/2$ steps, since M gets larger in each step. Each step calls the search algorithm. This itself has at most n steps (since R grows in each), and each step takes at most m simple checks, checking all edges in the graph. So we obtain a crude bound of $O(n^2 m)$, so certainly polynomial.

Even if we aren't concerned with algorithms, the theory behind this algorithm will give us a nice non-algorithmic result, concerning matchings and covers.

Definition. A *cover* for a graph G is a subset C of the vertices such that every edge contains at least one vertex of C.

If M is any matching and C is any cover, then $|M| \leqslant |C|$. To see this, define an injective map $f : M \to C$, where $f(e)$ is any vertex of $e \cap C$.

Suppose that we had found a matching M and a cover C such that $|M| = |C|$. Then we would know that M was a maximum size matching and C was a minimum size cover. This is an example of 'weak duality' and suggests the question of whether equality holds. The answer to the question is 'no' in general, just consider K_3.

Theorem 7.4. *(**König's Theorem (1931)**) 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, and 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 M-free 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 $b \in B \setminus R$. If $ab \notin M$ then since we can reach a, we can reach b via ab, a contradiction. On the other hand, if $ab \in M$ then $a \notin A^*$. Since $a \in R$ we can reach a from A^* along some non-trivial directed path. The last edge of this path must be of the from $b'a$ (since G is bipartite). So from our direction rule $b'a \in M$. But then $b = b'$ since $ab \in M$ and M is a matching. Hence we reach a via b, contradicting $b \notin R$. Thus C is a cover.

It remains to show that $|C| = |M|$. It suffices to show that every vertex in C is contained in some edge of M, and that no edge of M has both ends in C. (This will show $|C| \leqslant |M|$, and we noted previously that $|M| \leqslant |C|$ is immediate from the definitions; alternatively, the $f : M \to C$ defined before is now a bijection.)

Recall that $C = (A \setminus R) \cup (B \cap R)$. If $a \in A \setminus R$, then $a \notin A^*$ (since $A^* \subset R$) so a is in some edge of M. And if $b \in B \cap R$, then b is in an edge of M, as otherwise $b \in B^* \cap R = \emptyset$ which gives a contradiction.

Finally suppose for a contradiction that ab is an edge in M with both ends in C. Then $a \in A \setminus R$, $b \in B \cap R$. We can then reach a via b, contradicting $a \notin R$.

We conclude $|C| = |M|$. □

Although this is not the simplest route to Hall's Theorem overall, having proved König's Theorem we can easily deduce Hall's Theorem.

Proof of Theorem 7.1. As noted before that Hall's condition is necessary is easy. For sufficiency, suppose that every $S \subset A$ satisfies $|N(S)| \geqslant |S|$. Let C be any cover of G. By König's Theorem, it suffices to show $|C| \geqslant |A|$; then the minimum size of a cover is $|A|$, and so is the maximum size of a matching.

Let $S = A \setminus C$. Note that by definition of a cover, we have $N(S) \subset B \cap C$. Then

$$|C| = |A \cap C| + |B \cap C| = |A \setminus S| + |B \cap C| \geqslant |A| - |S| + |N(S)| \geqslant |A|.$$

□

8 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 they find the shortest route? (The problem was first posed by the Chinese mathematician Kwan Mei-Ko in 1960 and is named in his honour.)

Let G be a connected graph, and 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 $\ell(e) > 0$ be the length of e. The length of W is $\ell(W) = \sum_{e \in W} \ell(e)$. We want to find a shortest postman walk.

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 $\ell(G^*)$ is as small as possible. Note that such an extension is not a simple graph, but rather is a multigraph.

We now describe *Edmonds' algorithm* (1973). We *assume* that we have access to an algorithm for finding a 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' Postman Walk Algorithm.

1. Let X be the set of vertices with odd degree in G. Using Dijkstra's Algorithm, for each $x \neq y$ in X find an ℓ-shortest x–y path P_{xy}.

2. Let K be the complete graph with vertex set X. Define a weight function w on the edges of K by $w(xy) = \ell(P_{xy})$. Find a perfect matching M in K with minimum w-weight. Note that this perfect matching step makes sense as $|X|$ is even, by the handshaking lemma.

3. Let G^* be the Eulerian extension of G obtained by copying all edges of P_{xy} for all $xy \in M$. Find an Euler Tour W in G^*. (Fleury's algorithm yields such a tour.) Interpret W as a postman walk in G.

To analyze this algorithm we need a simple lemma.

Lemma 8.1. *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 vertex v of odd degree, and let C be the component of H containing v. By the handshaking lemma, C contains another vertex w of odd degree, so C (and hence H) contains a v–w path. □

Theorem 8.2. *Edmonds' Algorithm finds a minimum length postman walk.*

Proof. Let W^* be a minimum length postman walk. It suffices to show that Edmonds' 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 repeated 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 8.1 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 $P_1, \ldots, P_k$ be the paths found, noting that they are edge-disjoint by definition.

Writing $\ell(G)$ for the sum of the lengths of the edges in G, we have

$$\ell(W^*) - \ell(G) = \ell(H) \geqslant \sum_{i=1}^{k} \ell(P_i) \geqslant \sum_{i=1}^{k} d_\ell(x_i, y_i)$$

where x_i and y_i are the ends of P_i. But the final sum is just $\sum w(xy)$ over the edges of a matching in K, and Edmonds' Algorithm finds a walk with length $\ell(G)$ plus the *minimum* weight of a matching in K. □