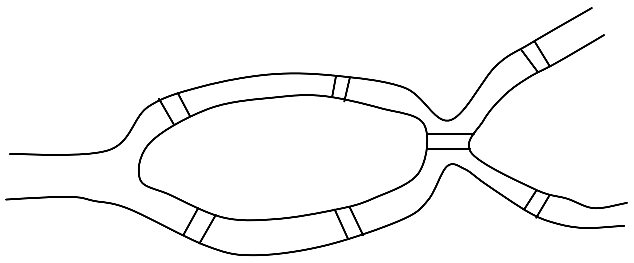
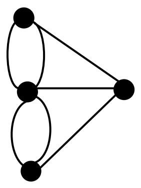
7. GRAPH ALGORITHMS

§7.1. Eulerian Graphs

The Swiss mathematician Leonard Euler (pronounced "Oiler") once asked whether it was possible to go for a walk around the city of Konigsburg, crossing each of the 7 bridges exactly once.



There are 4 pieces of land and 7 bridges. We can represent them by a graph with 4 vertices and 7 edges as follows.

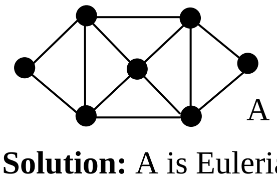


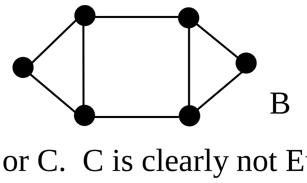
This is a graph with multiple edges and we have chosen to focus on graphs with at most one edge connecting any two vertices. However the reason there is no such path extends to graphs in general, whether or not there are multiple edges.

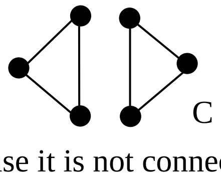
With a walk we must enter each vertex and leave each vertex so the degree of each vertex must be even. Euler did not make it clear whether one had to return to the starting point. If not, the degree of these two vertices could be odd. But the graph in question has all 4 vertices with odd degree. That is all 4 vertices have to be the start or end of a walk. The only way we could achieve the goal would be to take a helicopter ride at some point. If we had to return to where we started, two helicopter rides would be necessary. This is not what Euler had in mind.

An **Eulerian cycle** in a graph (we are back to undirected graphs with no multiple edges) is one that passes along every edge exactly once. An **Eulerian graph** is one that has an Eulerian cycle. (Remember to pronounce Eulerian as "Oil-air-ian".)

Example 1: Which of the following graphs are Eulerian?







Solution: A is Eulerian, but not B or C. C is clearly not Eulerian because it is not connected. But being connected is clearly not enough.

Theorem 1: An undirected graph is Eulerian if and only if it is connected, and every vertex has even degree.

Proof: Suppose G is Eulerian. Clearly G must be connected. Moreover, since we must enter and exit every vertex along different edges the degree of every vertex must be even.

Now suppose that G is connected and every vertex has even degree. We prove the theorem by induction on the number of edges. Suppose G has E edges and the theorem is true for graphs with fewer than E edges.

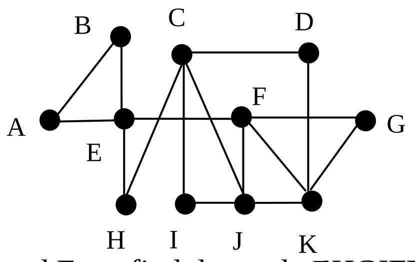
G clearly has a cycle. Simply start at some point and move to another vertex. Since every vertex has degree at least 2 we can continue the path until we repeat a vertex. That portion of the path between the two visits to that vertex will be a cycle.

Remove the edges of this cycle and remove any vertices that are now isolated (these would have had degree 2 in the original graph), The resulting graph will have every vertex of even degree (we have removed 2 edges from each vertex visited in this cycle). But it may no longer be connected. Let C_1, C_2, … , C_k be these components.

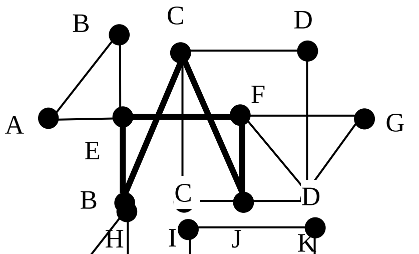
However each component is connected and will have every one of its vertices with even degree. Moreover these components will have fewer edges than G. Hence there is an Eulerian cycle in each of these components.

Now, start at one of the vertices in the original cycle. This vertex will be in one of the C_i. Now trace out the Eulerian cycle in this component. Continue around the original cycle. However, every time you enter a new C_i for the first time trace out the corresponding Eulerian cycle. Finally you will return to your starting point. This will be an Eulerian cycle for the graph G.

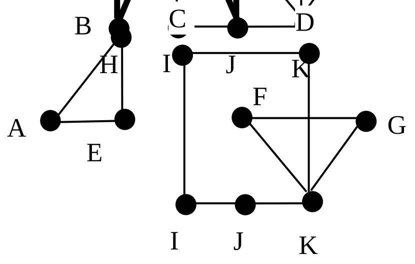
Example 2: Suppose G is the graph:



Start at A and travel the path AEHCJFE. Having repeated E we find the cycle EHCJFE.



Now remove this cycle.

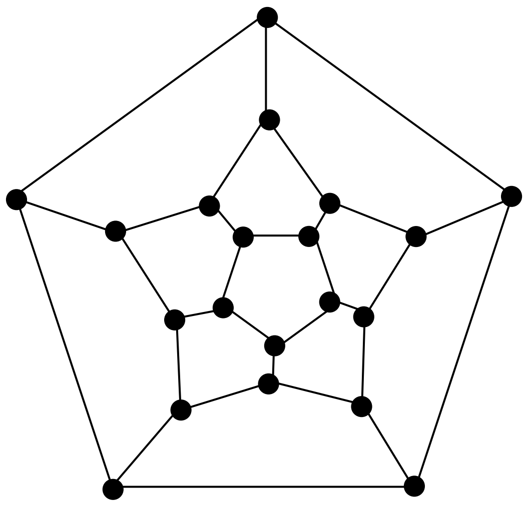


There are 2 components. Each has an Eulerian cycle, for example, ABE and CDKGFKJIC. Combining these with the original cycle, as described in the proof of Theorem 2, we get the Eulerian cycle EABE H CDKGFKJIC JFE.

§7.2. Hamiltonian Graphs

In 1857 the mathematician Sir William Hamilton created a puzzle that, in its day, was almost as popular as the Rubic’s Cube. It was a wooden dodecahedron in which the 20 vertices were labelled with the name of a city. The problem was to make a world tour, starting and finishing at the same city and passing along the edges, so that each other city is visited exactly once.

We do not need a solid dodecahedron to solve the puzzle. The edges form a graph on a sphere that becomes a planar graph when we flatten it out.



A **Hamiltonian path** in an undirected graph is a path that passes through each vertex exactly once. It clearly cannot pass along each edge more than once, but some edges will not occur.

A **Hamiltonian cycle** in an undirected graph is a cycle that passes through every vertex exactly once. (Of course it will return to the vertex where it began.)

A **Hamiltonian graph** is an undirected graph that has a Hamiltonian cycle. Clearly a Hamiltonian graph must be connected.

Hamilton's Dodecahedron Puzzle is equivalent to finding a Hamiltonian cycle in the above graph.

Theorem 2 (Ore): If $\deg(X) + \deg(Y) \geq n$ whenever X is not adjacent to Y in an undirected graph G with n vertices then G is Hamiltonian.

Proof: Suppose G satisfies these conditions but is not Hamiltonian. Add extra edges until you reach a graph H such that H is not Hamiltonian but adding one extra edge (U, V) makes a Hamiltonian graph K..

This edge must be included in every Hamiltonian cycle in K.

Removing this edge will give a Hamiltonian path $(V_1, V_2), (V_2, V_3), \ldots , (V_{n-1}, V_n)$ in H.

Everything that follows refers to the graph H.

Since G is a subgraph of H the assumption of the theorem carries across to H. This is because two vertices being non-adjacent in H implies that they are non-adjacent in G and the degree of every vertex in H is greater than or equal to its degree in G. Also, since H has exactly the same vertices as G, H has n vertices.

Let $S = \{i \mid V_i$ is adjacent to $V_1\}$. Then $\#S = \deg(V_1)$. Clearly $1 \notin S$. Moreover $n \notin S$, for if V_n is adjacent to V_1 then adding this to the Hamiltonian path would give a Hamiltonian cycle.

Let $T = \{i \mid i + 1 \in S\}$. Then $\#T = \#S = \deg(V_1)$. Also $V_n \notin T$.

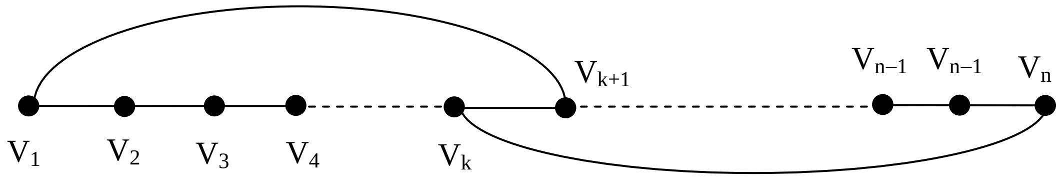
Let $R = \{i \mid V_i$ is adjacent to $V_n\}$. Then $\#R = \deg(V_n)$. Clearly $V_n \notin R$.

Since $n \notin S$, V_1 and V_n are not adjacent and so $\deg(V_1) + \deg(V_n) \geq n$.

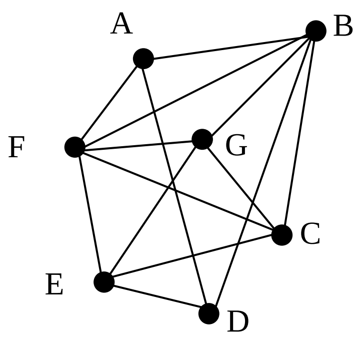
Hence $\#T + \#R \geq n$. Since $\#(T \cup R) \leq n - 1$, $\#(T \cap R) \geq 1$.

Let $V_k \in T \cap R$. Then (V_k, V_n) and (V_{k+1}, V_1) are edges in H.

Then $(V_1, V_2), (V_2, V_3), \ldots , (V_{k-1}, V_k), (V_k, V_n), (V_n, V_{n-1}), (V_{n-1}, V_{n-2}), \ldots (V_{k+2}, V_{k+1}), (V_{k+1}, V_1)$ is a Hamiltonian cycle in H, a contradiction.



Example 3: Let G be the following graph.

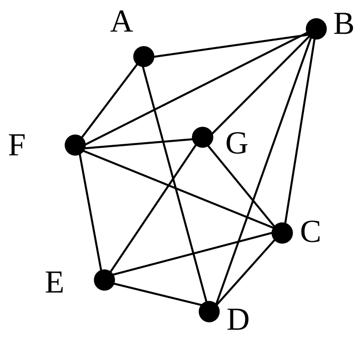


The number of vertices is n = 7. The degrees of the vertices are as follows:

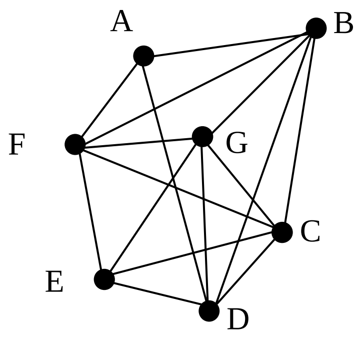
A	B	C	D	E	F	G
3	5	4	3	4	4	5

The pairs of non-adjacent vertices are: AC, AE, AG, BE and CD

At first glance it might appear that G is not Hamiltonian. Suppose this is the case.
Suppose that when we add the edge CD it is still not Hamiltonian. But adding the edge GD as well the graph is clearly Hamiltonian. So in terms of the notation of the above theorem H is



and K is

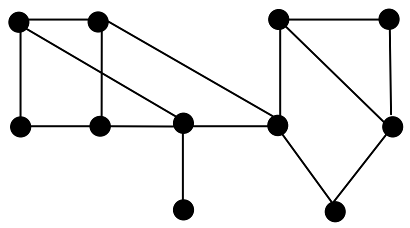


The cycle ABCGDEFA is a Hamiltonian cycle in K and DEFABCGD is a Hamiltonian path in H.
V_1 = D, V_2 = E, V_3 = F, V_4 = A, V_5 = B, V_6 = C, V_7 = G.
S = {2, 4, 5, 6, 7}, T = {1, 3, 4, 5, 6}, R = {1, 2, 3, 6}.
#T = 5 = #S = deg(D) = 5. #R = deg(G) = 5.
So $T \cap R$ = {1, 3, 6}.
Suppose we choose k = 1, so k + 1 = 2. Then we get the Eulerian cycle D GCBAFE D in H.
Suppose we choose k = 3, so k + 1 = 4. Then we get the Eulerian cycle DEF GCBA D in H.
Suppose we choose k = 6, so k + 1 = 7. Then we get the Eulerian cycle DEFABC G D in H.
These are not Hamiltonian cycles for G, but DEFGCBAD is. So G is indeed Hamiltonian, as it ought to be since it satisfies the hypothesis of Theorem 11.

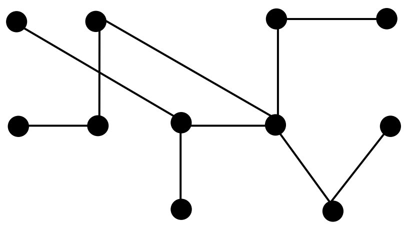
§7.3. Minimal Spanning Trees

Starting with any connected undirected graph we can obtain a tree by removing edges so as to remove cycles. If the resulting tree includes all the vertices we say that it is a **spanning tree**.

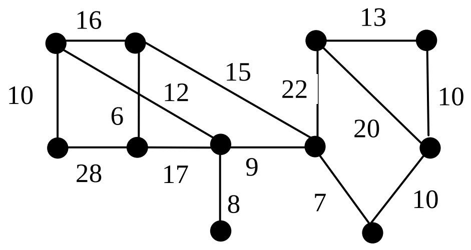
Example 4: The following undirected graph G is connected.



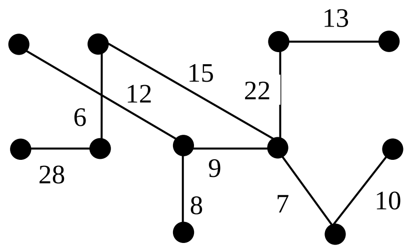
The following is a spanning tree for G.



Let us now make G into a weighted graph.



We transfer these weights to our spanning tree.



The total weight of this spanning tree is 130. Now the vertices in our original weighted graph might represent towns, with the weights being the distances between them. If we want to build roads connecting these towns, as cheaply as possible, we might choose to base the network on a spanning tree. This particular spanning tree would require a total of 130 kilometres of road. But perhaps there is one whose total weight is less than 130.

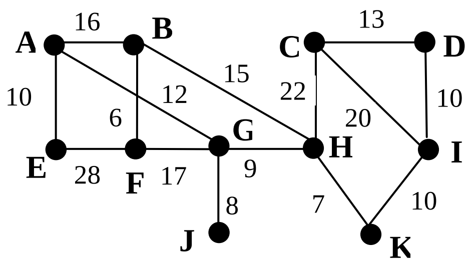
The **weight** of a spanning tree is the sum of the weights of its edges. A **minimal spanning tree** in a weighted undirected graph is a spanning tree of lowest weight. How might we systematically go about finding such a minimal spanning tree?

§7.4. Kruskal's Algorithm

Kruskal's Algorithm does this. We start with an edge of lowest weight. At each stage we examine all the edges which could be added without producing a cycle, and of these edges we choose one of lowest weight. While this seems a good strategy it is not clear that it will guarantee a minimal spanning tree at the end.

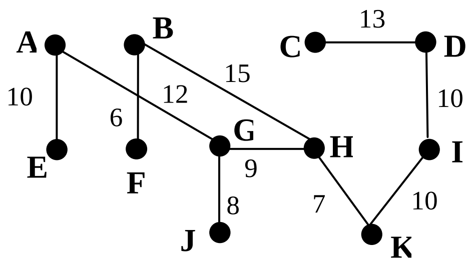
KRUSKAL'S ALGORITHM for finding a minimal spanning tree in a connected, weighted undirected graph.
(1) Create a list of the edges and their weights, in ascending order of weights. Call this list A.
(2) Create an empty list of edges, and their weights, called list B.
(3) Delete any edges from list A which would create a cycle if added to the graph in list B.
(4) If list A is empty go to step (7).
(5) Transfer the first edge from list A to list B.
(6) Go to step (3).
(7) List B will now contain the edges of a minimal spanning tree.

Example 4 (continued): We name the vertices.



LIST A	LIST B
BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	empty
HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6
GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7
GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8
AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9
DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10
IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10
AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10
CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12
BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13
AB16 FG17 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15
empty	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15

This minimal spanning tree is



and its weight is 100, a big improvement on the previous spanning tree. Moreover, by Kruskal's Theorem, we will never be able to do any better.

An important stage in the algorithm is to delete those edges which, when added to the tree in list B, would create a cycle. If we have a diagram to look at we can do this "by inspection". But if we have a graph with many thousand vertices there is no way we would attempt to draw it. The graph will only ever exist as a list of edges, together with their weights. So how would our Kruskal program deal with this?

We can do this by keeping a list of the components in our graph B. Then, as we scan through the edges in list A, we delete those edges whose two vertices lie in the same component. If vertices P, Q were in the same component there would be a path connecting them. Adding the edge PQ would provide a second path from P to Q and hence there would be a cycle. In this case we would delete that edge from list A. Otherwise that edge remains, unless it is chosen to be transferred to list B.

KRUSKAL'S ALGORITHM for finding a minimal spanning tree in a connected, weighted undirected graph.
(1) Create a list of the edges and their weights, in ascending order of weights. Call this list A.
(2) Create an empty list of edges and their weights called list B.
(3) Create an empty list of components.
(4) Delete any edges from list A where both endpoints lie in the same component.
(5) If list A is empty go to step (7).
(6) Transfer the first edge PQ from list A to list B.
(7) If P nor Q is in a component create a new component PR.
(8) If P is in a component and Q is not add Q to that component.
(9) If P, Q are in different components combine these components into one.
(10) Go to step (4).
(11) List B will now contain the edges of a minimal spanning tree.

Example 4 (continued):

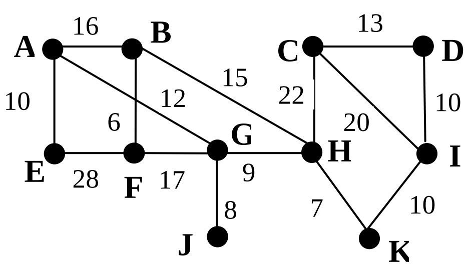
LIST A	**LIST B**	**COMPONENTS**
BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	empty	empty
HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6	BF
GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7	BF HK
GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8	BF HK GJ
AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9	BF GHJK
DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10	BF GHJK AE
IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10	BF GHJK AE DI

AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10	BF DGHIJK AE
CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12	BF ADEGHIJK
BH15 AB16 FG17 CI20 CH22 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13	BF ACDEGHIJK
AB16 FG17 EF28	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15	ABCDEFGHIJK
empty	BF6 HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15	ABCDEFGHIJK

§7.5. Prim's Algorithm

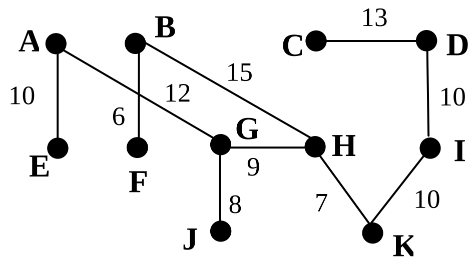
Prim's Algorithm:
(1) Create a list, called list A, containing all the edges except one of lowest weight.
(2) Create a list, called list B, consisting of this edge.
(3) Create a list, called list C, consisting of the endpoints of this edge.
(4) If list A is empty go to step (8).
(5) Remove any edges in list A that have both endpoints in list C.
(6) Of all the edges in list A that have exactly one endpoint in list B, choose one of lowest weight, transfer it to list B and add the other endpoint to list C.
(7) Go to step (4).
(8) List B will consist of the edges in a minimal spanning tree.

Example 4 (continued):



LIST A	**LIST B**	**LIST C**
HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 BH15 AB16 FG17 CI20 CH22 EF28	BF6	BF
HK7 GJ8 GH9 AE10 DI10 IK10 AG12 CD13 AB16 FG17 CI20 CH22 EF28	BF6 BH15	BFH
GJ8 GH9 AE10 DI10 IK10 AG12 CD13 AB16 FG17 CI20 CH22 EF28	BF6 BH15 HK7	BFHK
GJ8 AE10 DI10 IK10 AG12 CD13 AB16 FG17 CI20 CH22 EF28	BF6 BH15 HK7 GH9	BFGHK
AE10 DI10 IK10 AG12 CD13 AB16 CI20 CH22 EF28	BF6 BH15 HK7 GH9 GJ8	BFGHJK
AE10 DI10 AG12 CD13 AB16 FG17 CI20 CH22 EF28	BF6 BH15 HK7 GH9 GJ8 IK10	BFGHIJK
AE10 AG12 CD13 AB16 FG17 CI20 CH22 EF28	BF6 BH15 HK7 GH9 GJ8 IK10 DI10	BDFGHIJK
AE10 CD13 AB16 CI20 CH22 EF28	BF6 BH15 HK7 GH9 GJ8 IK10 DI10 AG12	ABDFGHIJK
CD13 AB16 CI20 CH22 EF28	BF6 BH15 HK7 GH9 GJ8 IK10 DI10 AG12 AE10	ABDEFGHIJK
CI20 CH22	BF6 BH15 HK7 GH9 GJ8 IK10 DI10 AG12 AE10 CD13	ABCDEFGHIJK
	BF6 BH15 HK7 GH9 GJ8 IK10 DI10 AG12 AE10 CD13	ABCDEFGHIJK

This spanning tree is:



This happens to be the same spanning tree as was obtained by Kruskal's Algorithm. Often different spanning trees are found by the two algorithms, but of course the weights will be the same.

§7.6. Graph Colouring

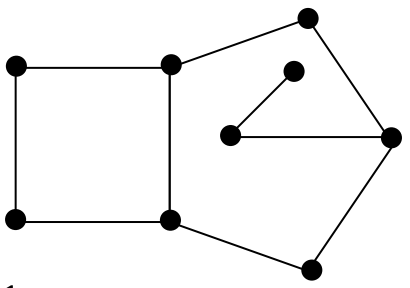
Imagine the task of designing a school timetable. If we ignore the complications of having to find rooms and teachers for the classes we could propose the following very simple model. We suppose that we have a certain number of subjects taken by a certain number of students. We'll suppose that all subjects require just one class. In setting up the timetable we must ensure that if two subjects have at least one student in common we must allocate different times.

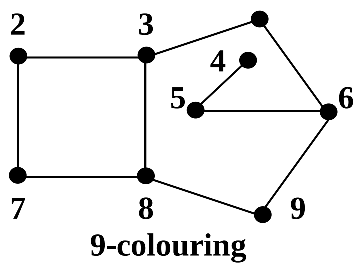
So imagine that the subjects are represented by the vertices of a graph and that we have an edge between subjects that have one or more common students. If we colour the vertices in such a way that adjacent vertices have different colours we can arrange for all subjects that were coloured with the same colour to be held at the same time.

We could simply colour each vertex with a different colour. This would correspond to timetabling each subject at a different time. But there aren't enough hours in the week and it's important to try to use as few time-slots, or as few colours, as possible. This is one application that would benefit from a solution to the problem of colouring graphs with as few colours as possible.

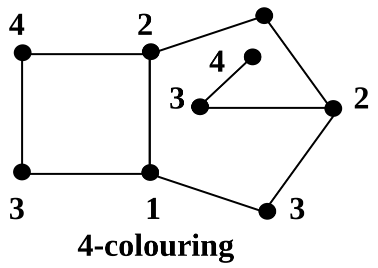
An **n-colouring** of a graph is an assignment of "colours" 1, 2, … , n to the vertices of a graph so that adjacent vertices have different colours. The chromatic number, $\mu(G)$, of a graph G, is the *smallest* n for which an n-colouring exists.

Example 5: For this graph G, $\mu(G) = 3$.

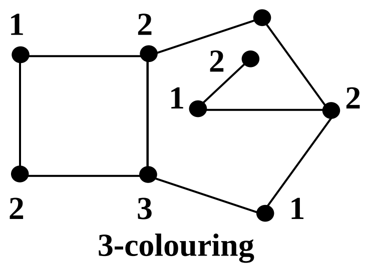




9-colouring



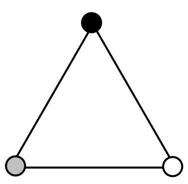
4-colouring

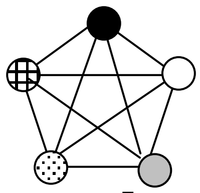


3-colouring

No 2-coloring exists. Around the pentagon we'd have to alternate colours, but because of the odd number of sides we'd end up with two adjacent vertices with the same colour. So 3 is the minimum number of colours.

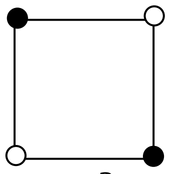
Example 6: The chromatic number of the complete graph K_n is n.
Proof: Since every vertex is adjacent to every other we need to use n distinct colours.

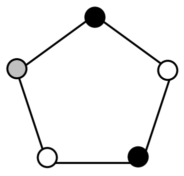

μ = 3


μ = 5

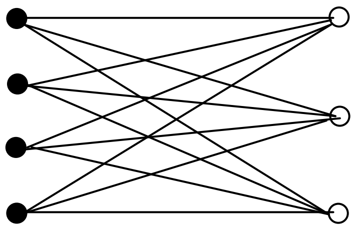
Every graph that contains a subgraph with chromatic number n must itself have chromatic number at least n. So, for example, the chromatic number of any graph that contains a triangle must be at least 3.

Example 7: The chromatic number of an n-sided polygon is: $\begin{cases} 2 \text{ if } n \text{ is even} \\ 3 \text{ if } n \text{ is odd} \end{cases}$

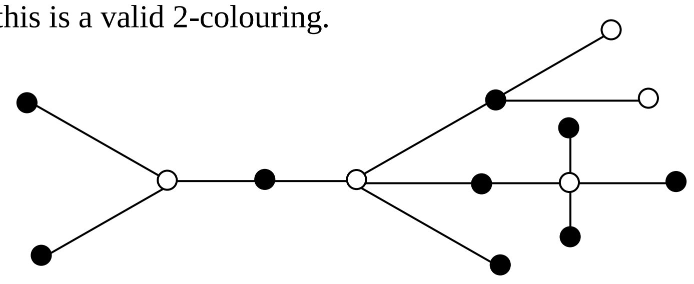

μ = 2


μ = 3

Example 8: The chromatic number of $K_{m,n}$ is 2.
Proof: Colour the m vertices one colour and the n vertices the other colour.


μ(K$_{4,3}$) = 2

Theorem 3: The chromatic number of a tree (connected graph with no cycles) is 2 (except where there's only one vertex).
Proof: Choose a vertex V_0. Each vertex has a unique distance from V_0. Use one colour to colour those vertices whose distance from V_0 is even and the other colour for those vertices whose distance is odd. Clearly this is a valid 2-colouring.



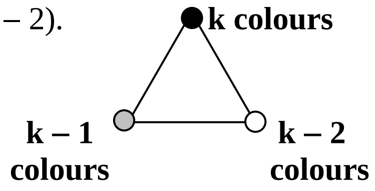
§7.7. The Chromatic Polynomial of a Graph

The **chromatic polynomial** of a graph G is the number of ways of colouring G with k colours (a polynomial in k). It is denoted by **$\Gamma(G)(k)$**. The chromatic number of G is the smallest value of k such that $\Gamma(G)(k)$ is positive.

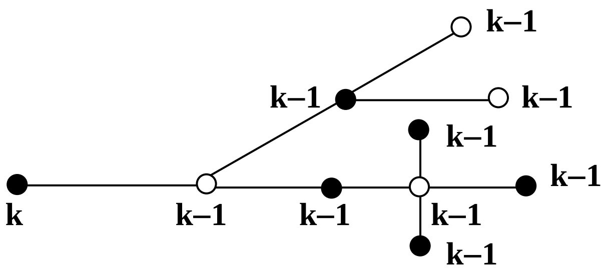
For simplicity, unless we need to specify the parameter k, we'll write $\Gamma(G)(k)$ simply as $\Gamma(G)$.

Theorem 4: $\Gamma(K_n) = k(k - 1) \ldots (k - n + 1)$.
Proof: Order the vertices in some way. There are k colours that can be assigned to the first vertex. Any one of the remaining colours can be assigned to the second vertex, and so on.

Example 9: $\Gamma(K_3) = k(k - 1)(k - 2)$.

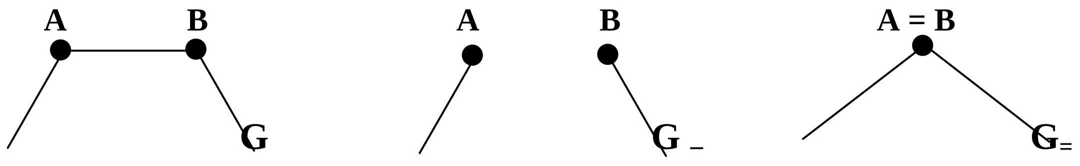


Theorem 5: If G is a tree with n vertices then $\Gamma(G) = k(k - 1)^{n-1}$.
Proof: We use induction on the number of vertices. It's clearly true for n = 1. Suppose the result is true for trees with n vertices and that we have one with n + 1 vertices. Choose any vertex of degree 1 and remove it, together with the associated edge. By induction the resulting tree can be coloured with k colours in $k(k - 1)^{n-1}$ ways. Reinstating the deleted vertex, it's adjacent to only one coloured vertex and so can be coloured in k – 1 ways, giving a total number of colourings of $k(k - 1)^n$.

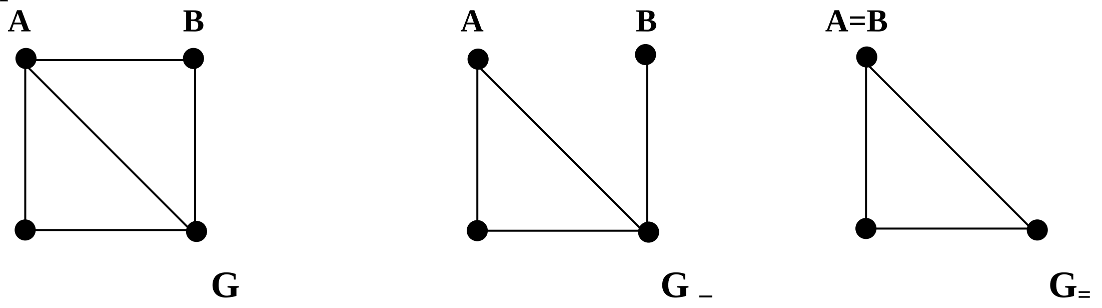


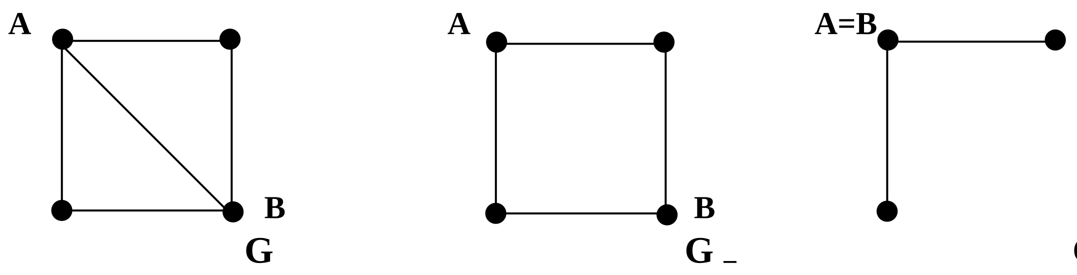
We now show how to compute the chromatic polynomial of a graph. The method is inductive. From a given graph G we produce two simpler graphs G_- and $G_=$ and we use their chromatic polynomials to obtain the chromatic polynomial of G.

Let G be a graph. Select any two adjacent vertices A and B.
Let **G_-** be the same graph with the edge AB deleted.
Let **$G_=$** denote the graph G with vertices A and B identified. (This means they become a single vertex and any edge having either A or B as an endpoint now has, instead, this combined vertex.)



Example 10:

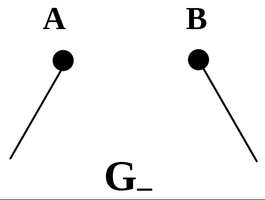
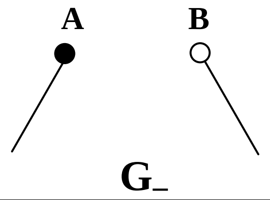
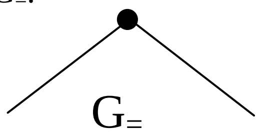
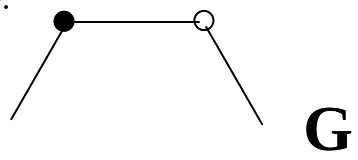




Theorem 9: Let G be a graph. Select any two adjacent vertices A and B and let G_- and $G_=$ be defined as above. Then

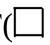
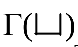
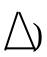
$$\Gamma(G) = \Gamma(G_-) - \Gamma(G_=).$$

Proof: If we k-colour G_- we're free to colour A and B the same colour. But we don't have to. We could just as validly colour them different colours (provided this was consistent with the colourings at the other vertices).

There are two types of colourings of G_- in terms of what happens to A and B	
A, B are given the SAME colour	A, B are given DIFFERENT colours
	
Each of these colourings of G_- gives a valid colouring of $G_=$. 	Each of these colourings of G_- gives a valid colouring of G. 

Hence $\Gamma_{G_-}(k) = \Gamma_{G=}(k) + \Gamma_G(k)$. The result now follows algebraically.

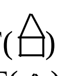
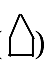
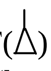
Example 11:

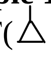
Γ() = Γ() – Γ()

$= k(k-1)^3 - k(k-1)(k-2)$

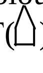
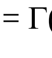
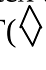
$= k(k-1)[(k-1)^2 - (k-2)]$

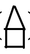
$= k(k-1)(k^2 - 3k + 3)$.

Example 12: Γ() = Γ() – Γ()

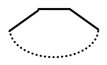
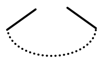
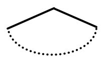
Now Γ() = Γ( ).(k – 1) = $k(k-1)^2(k-2)$

since once the triangle has been coloured the remaining vertex can be coloured any colour, except the colour of the vertex to which it is attached.

and Γ() = Γ() – Γ() = $k(k-1)^4 - k(k-1)(k^2 - 3k + 3)$ from example 7.

Hence Γ() = $k(k-1)^4 - k(k-1)(k^2 - 3k + 3) - k(k-1)^2(k-2)$
$= k(k-1)[(k-1)^3 - (k^2 - 3k + 3) - (k-1)(k-2)]$
$= k(k-1)[k^3 - 3k^2 + 3k - 1 - k^2 + 3k - 3 - k^2 + 3k - 2]$
$= k(k-1)(k^3 - 5k^2 + 9k - 6) = k(k-1)(k-2)(k^2 - 3k + 3)$.
It follows that the chromatic number of is 3. This was pretty obvious without all that calculation, but for a much more complicated graph, where the chromatic number is not so obvious, this could be a useful technique. Also, it could form the basis for a computer program to compute the chromatic number of a graph.

Theorem 6: If C_n is a cycle with n vertices then $\Gamma(C_n) = (k-1)^n + (-1)^n(k-1)$.
Proof: Induction on n. $\Gamma(C_3) = \Gamma(K_3) = k(k-1)(k-2)$ so it holds for n = 3.
Suppose that it holds for n and let $G = C_{n+1}$.

Γ( **) = Γ(**  **) − Γ(**  **)**.

$$\Gamma(C_{n+1}) = k(k-1)^n - \Gamma(C_n)$$

$= k(k-1)^n - (k-1)^n - (-1)^n(k-1) = (k-1)^{n+1} + (-1)^{n+1}(k-1)$.
Hence it holds for n + 1 and so, by induction, for all n.