Graph Theory. Chapter 4.

Wandering.

Here is an algorithm, due to Tarry, that constructs a walk in a connected graph, starting at any vertex v_0, traversing each edge exactly once in each direction, and ending at v_0. If postman Pat is to visit every house on his patch without constantly crossing the road, so that he has to go once down each road in each direction, he has to solve this problem to avoid unnecessary travel.

This algorithm can be described most clearly in a way that is very far removed from pseudo code, though any programmers among you should have no problem in turning this into code.

I shall refer to the graph as a maze to emphasise the more visual aspect.

We need to consider the edges that come in to any vertex as being arranged in some circular order. We shall suppose that the graph has been drawn on paper, and that we take the edges at a vertex in clockwise order. The computer will have the edges with one end at any given vertex stored in some order; so in this case the edges at a vertex again come in a natural circular order, the edge after the last edge in the list being the first edge in the list.

We enter the maze with a collection of white stones and a collection of black stones, one for each vertex. Starting at v_0, choose any edge that starts at v_0, and put a white and black stone at the begining of this edge. Now go down this edge to the next vertex.

When we come to a vertex there are two possibilities.

(a) There is a black and white stone at this vertex. Move the white stone round to the next edge; that is to say, the next edge in the clockwise direction from its present position. Go down the edge now marked by this stone, picking up the black and white stone as you go if they now both mark the same path; except that if you are at the start vertex, and have just picked up the two stones, the walk is complete, and you halt.

(b) The vertex we come to has no stones. Now put down a black stone to mark the edge we entered by, put down the white stone to mark the next edge round from this edge, and go down the edge marked by the white stone.

It remains to prove that this procedure is correct. It turns out that the black stones are not really needed. If we don't have any black stones, there are two adjustments to be made. The halting criterion remains essentially the same. When we are at the start vertex, and move the white stone, if it now marks the edge we first went down, rather than going down this edge again, we halt. With the other vertices, when we exit from the edge with which we first come to that vertex, that is to say, when the white stone now marks the edge that would have been marked by a black stone, we do not need to notice that this has happenned, we just leave the white stone where it is. It will follow from the proof that we never come to this vertex again anyway, so picking up the stones is simply a matter of being tidy.

We have two things to prove.

1. The algorithm will not take us twice down the same edge in the same direction. This will also imply that the algorithm will teminate; we don't keep going round in circles, as with the three men in a boat, when they got lost in the Hampton Court maze.

2. The algorithm takes us at least once down each edge in each direction.

Let us prove 1. If there is an edge that we go down twice in the same direction there is a first such edge, let this edge be e. Suppose we go down e from vertex u to vertex v, and that this is the first time we have been down an edge twice in the same direction. Suppose that u has valency d. The algorithm ensures that we only leave u for the second time when we have already left u along all the other edges, so we are leaving u for the $d+1$-st time. This means that we have come to u twice along the same edge. But this contradicts the choice of e as the first duplicated edge, and hence 1 is proved.

Now prove 2.

Suppose that 2 is false. Call a vertex v 'saturated' if, when the algorithm has completed, we have gone down every edge with one end at v once in each direction. Suppose that there is an unsaturated vertex on the path defined by the algorithm. Then there is a first unsaturated vertex to be visited, call it v. Now v is not v_0, because the algorithm continues until v_0 is saturated. Suppose that the first time we come to v it is along an edge e from w to v. So by the choice of v as the first unsaturated vertex to be visited, w is saturated. However, the edge from w to v was the edge marked by the black stone at v, so since v is unsaturated we never go down the edge e from v to w. But this makes w unsaturated, and we came to w before we came to v. This contradicts the assumption that v was the first unsaturated vertex to be visited.

It follows that the path defined by the algorithm only visits saturated vertices. But if there are saturated and unsaturated vertices there must be an edge joining a saturated edge to an unsaturated edge, and since one end of this edge is saturated, the path must take us along this edge, visiting the unsaturated edge.

Thus 2 is also true; and this proves that the algorithm is correct.

Eulerian walks

A very famous problem goes as follows. Under what conditions on a connected graph is it possible to construct a walk that contains every edge exactly once, ending where it starts? That is to say, if the postman is prepared to keep crossing the road, and hence only needs to go down each road once, when is it possible to organise the route so that, starting at any given point, this can be done?

The original version of this problem was a special case, the 'bridges of Königsberg' problem, and was solved by Euler. The answer is really rather easy; unlike most of the problems that Euler solved. A walk in a graph that goes along every edge, ending where it starts, is generally called an 'Euler cycle'. A graph with an Euler cycle is said to be 'Eulerian'. As an Euler cycle is not a cycle according to our definition, I shall use the term 'Euler walk'.

Lemma. *In an Eulerian graph every vertex has even valency.*

Proof. Any walk that ends where it starts must come to every vertex as often as it leaves it. If an Eulerian walk comes to a vertex k times and hence leaves it k times the vertex must have valency $2k$.

It remains to prove that if every vertex of a connected graph has even valency then the graph is Eulerian. We shall do this constructively. That is to say, we shall give an algorithm to find an Eulerian walk, and prove that the algorithm is correct.

The first idea in the algorithm is to delete edges from the graph as we pass along them. This stops us from going along the same edge twice.

The idea is to ensure, before we go along an edge, and delete it, that the deleted graph will not then contain two proper components, where a component is proper if it contains an edge or the start vertex v_0.

Note that the sum of the valencies of all the vertices in any graph is even, being twice the number of edges. It is easy to see that as we construct our walk, deleting edges as we go, the deleted graph will have all vertices of even valency if we are at v_0, since we have then left each vertex as often as we have come to it, or there will be just two vertices of odd valency, namely v_0 (which we have left once more than we have come to it) and the vertex we are at (which we have come to once more than we have left).

Suppose we try to construct an Euler walk at random, just constructing any walk, deleting edges as we use them. If we fail we will eventually come to a situation in which we can go no further, but have not gone down every edge. When this happens the vertex we are at will have valency 0. Since 0 is an even number we are now at the start vertex. It is very easy to construct an example where a carelessly chosen walk gets into this difficulty.

The algorithm will go as follows. We must make sure that the deleted graph does not have more than one proper component, where a proper component is one which either contains an edge, or contains v_0. That is to say, we do not care if the deleted graph has components consisting of a single point that is not v_0. Notice that the vertex at which the walk has got to at any time is in a proper component; for either this vertex has odd valency in the deleted graph, in which case it has an edge attached to it, or it has even valency and is the start vertex.

Now the crucial observation is that, provided that the graph has only one proper component, and an Euler walk has not yet been completed, we can always continue the walk without creating two proper components. Once this has been proved the algorithm is obvious. Just keep extending the walk, one edge at a time, always checking, before the next edge is added, that deleting it will not create two proper components.

Suppose then, for a contradiction, that we have arrived at a vertex v with the deleted graph having only one proper component, but still containing at least one edge, but that we can go no further without creating two proper components. Consider the valency d of v in the deleted graph.

If $d = 0$ then $v = v_0$ as d is even; and v_0 constitutes a proper component of the deleted graph as $d = 0$. Since the deleted graph contains only one proper component, it has no edges, and an Euler walk has been completed, against the hypothesis.

If $d = 1$ then we are at a vertex v that is not v_0, as 1 is odd. Now v is attached to the rest of the proper component of the deleted graph by a single edge; when this edge is deleted there will still be at most one proper component, as we will have simply cut off a single vertex that is not the start vertex.

If $d > 1$ is even then we are at the start vertex, and every vertex in the deleted graph has even valency. If we now delete any edge e joining v_0 to some other vertex v_1 then we have created two proper components. One of these contains the vertex v_1, and the other the vertex v_0. But then each component contains exactly one vertex of odd valency, namely v_0 or v_1, and this is impossible.

If $d > 1$ is odd then we are not at the start vertex, and in the deleted graph all vertices except for v and v_0 have even valencies. If now we delete an edge e joining v to w say, the only vertices in the deleted graph with odd valency will now be v_0 and w (unless $v_0 = w$, in which case all vertices will now have even valency). So, if we have created two proper components by deleting e, then v_0 and w must both be in the same proper component, and v must be in the other component (since removing e separates these components). So every walk in the deleted graph (before e is deleted) that goes from v to v_0 must pass along e, and e is the only edge with v at one end that has this property. So there is only one edge with v at one end whose removal will create two proper components in the deleted graph. But since v has valency at least 3 in the deleted graph, there are at least two edges that we can now go down without creating two proper components.

This completes the proof of the algorithm is correct, and that every connected graph with all vertices of even valency is Eulerian.

Let us now think a little more about this algorithm. When we are at v_0, the above argument has shown that we can delete any edge through v_0 without creating two proper components; and that if we are at any other vertex v, then there is at most one edge through v whose deletion would cause a second proper component to arise. It follows that we can construct an Eulerian walk for fussy tourists. That is to say, when we come to a vertex we can allow the tourists to decide which edge to go down next; subject, of course, to them not choosing an edge they have already been down; except that, whenever there is a choice of edge, the guide may ban one of the possible choices. Even better for the tourists; whenever the walk returns to v_0 they can choose any edge they have not been down before.

This makes the algorithm more efficient. If the program finds that the first edge it tries does cause two proper components to arise, it is then free to pick any other edge through the appropriate vertex that has not been deleted; and if it is at v_0 any edge may be chosen.

As a consequence of the fussy tourist observation, we can deal with Eulerian walks for digraphs. Suppose that we have a connected directed graph. When can we construct an Eulerian walk that goes in the correct direction down each edge?

Clearly a necessary condition is that the in-valency of each vertex should equal its out-valency, where the in-valency is the number of edges that lead into that vertex, and the out-valency is the number that lead out. It is now easy to prove that this condition is sufficient as follows.

Let us use the same algorithm as before. The only problem is that, when we add a new edge to the walk, it must point in the correct direction.

What effect does this restriction have?

If the valency in the deleted graph of the vertex v we are at is 1 then this edge must point away from v; so there is no problem. If the valency is 2 then we are at v_0, since 2 is even. One of these edges points in to v_0 and one points out. Since we are at v_0 there is no restriction in the original undirected algorithm, so we are free to choose the edge pointing in the correct direction. If the valency is greater than 2 then there is more than one edge pointing away from v, and at most one of these is banned; so we can pick one that is not banned, and proceed.

This proves the following result.

Theorem. *Let G be a connected directed graph. Then there is an Euler walk in G if and only if the in-valency of each vertex (that is to say, the number of edges with this vertex as its end) is the out-valency (that is, the number of edges with this vertex as its begining).*

Now consider walks in which we start and end at different vertices.

Lemma. *Let G be a connected graph, and $v \neq w$ be vertices in G. There is a walk in G from v to w that passes exactly once down every edge of G if and only if the valencies of v and w are odd, and all other valencies are even.*

Proof. Consider the graph obtained from G by adding a new edge joining v and w.

Now suppose that we want to walk down every edge at least once in a connected weighted graph G that is not Eulerian, returning to our starting point, and covering as short a distance as possible. If the graph were Eulerian we could simply walk once down each edge, so the total distance would be the sum of the weights of the edges. In general we have to repeat certain edges. These repetitions can be signified by adding in duplicate edges whenever we go down an edge again. With these duplicate edges, the graph has become Eulerian. So construct a subsidiary graph H whose vertices are the vertices of G of odd valency. H is to be a complete weighted graph, the length of the edge joining two vertices in H being the minimal distance between these vertices in G (computed using Dijkstra). Now H has an even number of vertices, and we want to construct a minimal perfect matching in H. That is to say, we divide the vertices of H into pairs, so that the sum of the weights of the edges joining matching pairs is as short as possible. We then translate these edges back into walks in G, and adding an extra copy of each edge arising like this to G, we turn G into an Eulerian graph, and find an Euler walk as before.

Now let us consider the time complexity of these algorithms.

When we find an Eulerian walk in an Eulerian graph, at each step we need to find whether deleting an edge through the vertex we are at will create two proper components in the deleted graph G'. In fact, at each step we only need to test one edge (or none if the vertex has valency 1 in G'); because if the first edge we test is legal then we take that edge; and if the first is illegal we can take any other. But for each vertex we visit we may have to carry out a connectivity test, and the cost of checking connectivity is $O(E)$ where E is the number of edges; this gives the time complexity of the algorithm as $O(E^2)$. This is reasonably fast.

A more serious problem arises when we construct as short a walk as possible along all the edges of a non-Eulerian weighted graph. In this case we have to find a minimal perfect matching in a complete graph with an even number of vertices. How is this done?

There is a clever algorithm for solving this problem; but let us consider the issues that arise if we can't find a smart algorithm.

Suppose that the graph has 30 vertices. To find the best perfect matching we could consider finding all perfect matchings. So take the first vertex. It could be matched with any one of the other 29. Now take the next unmatched vertex. It could be matched with any of the 27 other unmatched edges, and so forth. So the total number of possibilities is $29 \times 27 \times \cdots \times 3 \times 1$. This number is about 6.1×10^{18}. If I could process 1,000,000 possibilities

in 10 seconds (which would be hard) it would take a million million minutes to complete the calculation. For $2n$ vertices in general, the number of possibilities is $(2n)!/(n! \times 2^n)$. By Stirling's formula, $n!$ is about $n^n e^{-n}$; so putting this in the formula we get $2^n n^n e^{-n}$ as an approximation. This complexity is worse than exponential.

The challenge then is to find a practical way round this difficulty. If we cannot think of a clever algorithm that is bound to lead us rapidly to the correct solution (for example a greedy algorithm like like Prim's algorithm), what can we do?

We need first to put some structure on our search for the best solution.

Construct a rooted tree as follows. The root corresponds to the start of the search. We can now take any of the thirty vertices, say vertex 1, the first vertex, and try matching it to each of the other 29 vertices. These 29 choices correspond to the possible starts to our search, and 29 corresponding nodes are joined by edges to the root. For each of these choices we now pick the first unmatched vertex, and consider the 27 ways of matching this vertex (it may not be matched to either of the vertices matched in the first choice), and for each of these 27 choices we add a new vertex, attached by an edge to the vertex corresponding to the earlier choice. We now have a rooted tree of depth 2, with 29 descendants of the root, each descendant having itself 27 descendants. And so we continue, constructing a tree with of depth 14, where the total number of vertices of depth $1, 2, 3, 4, \ldots$ is $1, 29, 29 \times 27, 29 \times 27 \times 25, \ldots$. At the end, we have a VERY large number of leaves at the bottom of the tree. Each of these leaves corresponds to a perfect matching, and we are required to find a perfect matching of minimum weight. Of course we cannot in practice construct this vast tree.

To make this calculation practical, we have to have some powerful means of discovering that we have made a wrong choice (when this is the case) at an early step in the process.

A very simple approach would be as follows. We start by looking for the best matching we can find following some common sense ideas that should lead to a good solution, though probably not to the best. This is a *heuristic* approach. Well suppose that our heuristics lead to a solution of weight 100. Now we search the tree systematically looking for a better solution. So we make a first choice, pairing v_1 with v_2, say, thus moving to a point in the search tree of depth 1. Then, with these two matched, we match another pair v_3 and v_4 say, moving to a point of depth 2 on the tree. Suppose that, when we get to depth 4, we find that the total cost of all the choices we have made is already greater than 100. Now we know that we have erred, and we try different possibilities for the fourth choice. If none of these is any good, we try changing the third choice, keeping the first two as they were, and so forth. Thus we go up and down the tree, doing a *back track* search. Back-tracking is the process of moving up the tree when we find that we have gone wrong.

There is a great deal to be said about back track searches, and I will say very little. It is hard or impossible to give a satisfactory account of their complexity. In the worst case the heuristics may not help very much, and the cost of the algorithm becomes exponential.

In our case we need not wait until the cost of the choices we have already made exceeds the best weight (100 in the above discussion) that we have found. If the cost so far is 70, and we can see that the weight of the remaining choices must exceed 30, then we have gone wrong, and must back-track. It is easy to find an algorithm that gives a reasonable lower bound for the cost of the remaining choices in a reasonably short space of time. Also, we

want to organise the search so that it makes the most promising choices first, so that if our initial upper bound of 100 is too high we may quickly find a smaller upper bound, as this will speed the search.

It remains to think of some good heuristics.

The simplest way to look for a good matching would be to take the cheapest edge, remove the two vertices and adjacent edges from the graph, and iterate. But suppose, for example, that vertex a is joined to vertex b by a fairly cheap edge, say of weight 10, and that all other edges through a are expensive, say one of weight 100 and all others of greater weight. If b is also joined to some other vertex c with an edge of weight 8 it might be an error to choose this edge of weight 8, as then we cannot match a with b. So it might be best not to keep picking the cheapest edge available, but the most urgent. An edge would be urgent it was the cheapest edge joining one vertex to another, and the urgency would be the cost of the second cheapest edge adjacent to the vertex in question less the cost of the edge in question; thus the edge of weight 10 above would have urgency 90.

Now how would you form a lower bound to the cost of forming a perfect matching? The simplest idea is to take for each vertex v the cost of the cheapest edge with one end in v, add these up, and divide by 2. Why does this work? Consider any matching, and think of the weight as giving the length of the corresponding edge. Cut each edge in the matching into two equal halves. Now each vertex is attached to half an edge, of length at least half the length of the shortest edge adjacent to that vertex. Adding these half lengths gives the wieight of the matching, and this is at least half the sum of the minimum weights.

My program for finding a perfect matching succeded in dealing with a random complete weighted graph on 30 vertices (it is the weights that were random) in about 10 seconds, looking at about 250,000 matchings. The program took me a day to design, implement and check (and it is the first back-track search I have implemented). It should be possible to do a better back-track; but I don't know how much faster I could make it.

De Bruin Sequences

We are given an alphabet of m letters $a_1, \ldots, a_m$, and an integer n, and we wish to find a sequence of letters in this alphabet of length k, where k is as big as possible, so that if we repeat the given sequence indefinitely, and read n successive letters starting from any of the k different points in the original sequence, we will always get a different word.

For example take $m = 2$ and $n = 2$. Writing a and b for a_1 and a_2, we could consider the sequence aba, so $k = 3$. Repeating this to get $abaabaaba\cdots$ we can see that if we start at each of the three possible starting points, we get the 2-letter words ab, ba and aa. These are all different. If we try to go one better by taking $k = 4$ and try $abba$ as our basic sequence. This gives $abbaabbaabba\cdots$, and we have succeeded again, the four words we get are ab, bb, ba, aa. But if we try for $k = 5$ we are doomed to failure, as there are only 4 possible 2-letter words on $\{a, b\}$. There are m^n words of length n on m letters; so in general we want to find an example with $k = m^n$. Such a sequence is a 'de Bruin' sequence. When do they exist? Can we always find one?

To solve this problem, construct a digraph whose vertices correspond to the m^{n-1} words in our alphabet of length $n-1$. From each node v, and each letter a in the alphabet,

there is an edge e_{va} that goes from v to the node corresponding to the word obtained by deleting the first letter in the word corresponding to v and adding a to the end.

This graph is a directed Eulerian graph. Each vertex has in and out valency equal to m; and it is connected as we can get from any word of length $n-1$ to any other word of length $n-1$ in $n-1$ steps, where a step consists of adding a letter at the end and deleting one at the begining.

So what would an Eulerian walk now give us? The graph has m^n edges so the walk has length m^n. If we associate with an edge e_{va} the letter a, we can read off the sequence of letters associated with the Eulerian walk. Let x_i be the letter associated with the i-th edge in this walk. Then the sequence $(x_1, x_2, \ldots, x_{m^n})$ is a de Bruin sequence. For suppose we start at any vertex v, and go round and round our Eulerian cycle. It will be convenient to understand suffices modulo m^n. Then after going along n successive edges, corresponding to letters $x_k, x_{k+1}, \ldots, x_{k+n-1}$, the last edge takes us from the vertex that corresponds to the word $x_k x_{k+1} \cdots x_{k+n-2}$ to the vertex that corresponds to the word $x_{k+1} x_{k+2} \cdots x_{k+n-1}$. Since we never go down the same edge twice as we go round the Euler walk, we never hit the sequence of letters $x_k x_{k+1} \cdots x_{k+n-1}$ twice. So all sub-words of length n starting at different points of the cycle are distinct, as required.

Hamiltonian walks

.

A *hamiltonian walk* in a graph G is a walk that goes to every vertex exactly once, returning to the initial vertex. (So you might say that it goes to the initial vertex exactly twice.) If we have a hamiltonian walk in G starting at one vertex we can easily find one starting at any other vertex; the hamiltonian walk can be taken as a cycle, and you can go round the cycle (in either direction) starting at any vertex. A graph is said to be hamiltonian if it has a hamiltonian walk.

Unlike the case of Euler walks, there is no known efficient method of deciding whether or not a graph G is hamiltonian, or of finding hamiltonian walks if they exist.

It is obvious that a hamiltonian graph must be connected; also that a complete graph is hamiltonian.

Here is an algorithm due to Ore for finding hamiltonian walks in a certain class of graphs.

Theorem. *Let G be a simple graph with n vertices such that if u and v are any two vertices not joined by an edge then $v(u) + v(w) \geq n$, where $v(u)$ is the valency of u. Then there is a hamiltonian walk in G.*

Proof. We will prove the result by constructing a hamiltonian walk. So starting at any vertex v_0, construct a walk never visiting a vertex twice (except that we return to v_0 when every other vertex has been visited), and continuing for as long as possible. If we fail to construct a hamiltonian path in this way, we will get stuck at some vertex u; so every edge with one end at u has the other end at some vertex we have already visited. When this happens, cheat. Add (in red) a new edge joining u to an edge w we have not yet visited (or to v_0 if they have all been visited). Continuing in this style, we get a hamiltonian walk with some red edges. We now need to get rid of these red edges. So let the sequence of vertices visited in the walk be $(v_0, v_1, \ldots, v_n = v_0)$. Suppose that v_j is joined to v_{j+1} by a red edge. (I allow here $j = n - 1$, and take $v_n = v_0$.) It follows that v_j was not joined to v_{j+1} in the original graph, so $v(v_j) + v(v_{j+1}) \geq n$. Place a tick against v_i whenever v_i is joined by an edge in G to v_j, and place a cross against v_i if v_{i+1} is joined by an edge in G to v_{j+1} (where again $v_n = v_0$). The fact that we are considering edges in G means that red edges are not allowed when producing these ticks and crosses. Since $v(v_j) + v(v_{j+1}) \geq n$, the total number of ticks and crosses is at least the total number of vertices. But v_j has neither a tick (v_j is not joined to v_j by an edge) nor a cross (v_{j+1} is not joined to v_{j+1}). So some vertex v_i has both a tick and a cross. Then v_i is joined to v_j and v_{i+1} is joined to v_{j+1}. So now we can consider the walk that follows the original walk, but starting at v_j and going in the direction of decreasing suffices to v_{i+1}, then goes to v_{j+1} along the newly found edge, and then follows the old walk, but now in the direction of increasing suffices, till we get to v_i, and then go along the other new edge to v_j. This is a hamilton walk, whose edges are the same as the edges in the old walk, except that we have replaced edges (v_i, v_{i+1}) and (v_j, v_{j+1}) by (v_i, v_j) and (v_{i+1}, v_{j+1}). The two new edges lie in G; but at least one of the old edges (namely (v_j, v_{j+1})) was red. So we have replaced the old hamiltonian graph with another, the new one having fewer red edges. We can continue in this way until we have no red edges in our walk.

A very important variant of this problem is as follows. Suppose that we have a complete weighted graph G. Clearly G is hamiltonian; but the problem is to find a hamiltonian path of minimal weight. Unfortunately there is no known efficient algorithm that will always solve this problem. One could use a back-track search; but as we have seen, these are likely to be inefficient. As an alternative to a method that may be very slow, but which will give the best solution, we can use a method that is fast, but that may not give the best solution. Here is a simple idea. First construct a hamiltonian walk, using some heuristic to keep its weight small. For example, construct the walk edge by edge, always taking the shortest possible edge to a vertex that has not yet been visited. Now try to improve this walk as follows. Let the vertices be $v_0, v_1, \ldots, v_{n-1}, v_n = v_0$ in the order visited by our walk. Suppose that we can find four distinct vertices $v_i, v_{i+1}, v_j, v_{j+1}$ such that $w(v_i, v_j) + w(v_{i+1}, v_{j+1}) < w(v_i, v_{i+1}) + w(v_j, v_{j+1})$. Then we can find a hamiltonian cycle that is shorter than the original one by changing the walk in exactly the same way that we changed the walk in Ore's algorithm. That is to say, we replace the edges (v_i, v_{i+1}) and (v_j, v_{j+1}) by (v_i, v_j) and (v_{i+1}, v_{j+1}).

Note that I am referring to an edge that joins the vertices u and v by (u, v). Strictly this is invalid, as there might be two edges joining the same pair of vertices. For the purposes of determining whether or not a graph is hamiltonian, we can throw away any such duplicate edges; also any loops. So we may as well assume that the graph is simple, in which case the problem does not arise. A complete graph is simple by definition.

Suppose that the weights on the graph satisfy the triangle inequality. Here is a good heuristic for finding a reasonable solution. First use Prim's algorithm to find a minimal spanning tree. Then use Tarry's algorithm to find a walk that goes exactly once down each edge of this tree in each direction. Then use the triangle inequality to delete repeated vertices one at a time. That is to say, if the walk has successive vertices u, v, w, where v has been visited twice, then delete this v and go directly from u to w.

The travelling sales representative

This is the most famous of graph-theoretic problems, traditionally called 'the travelling salesman's problem'. Given a connected weighted graph, find a walk that visits every vertex, returning to the one where it began, that is of minimal weight.

Given such a walk $(v_0, e_0, v_1, e_1, \ldots, v_n = v_0)$ we can obviously find another walk of the same length that starts and ends at any given fixed vertex v_i and visits every vertex v_j by simply taking the walk to be $(v_i, e_i, \ldots, e_{n-1}, v_n = v_0, e_0, v_1, \ldots, v_i)$; so if we can find the best solution starting at one vertex we have found the best solution starting from any vertex.

Note that we will expect to pass more than once through various vertices.

Given two vertices v and w there may be an edge joining them that is longer than some path from v to w. Using Dijkstra's algorithm, we can easily find the shortest path from v to w, and if d is the weight of this path, we can assume an edge of weight d joining v to w. So we can assume that we have a complete graph, the length of each edge being the Dijkstra distance.

Now the sales rep problem reduces to finding a hamiltonian walk of minimal length in this graph. As this graph satisfies the triangle inequality we may use the above heuristic.