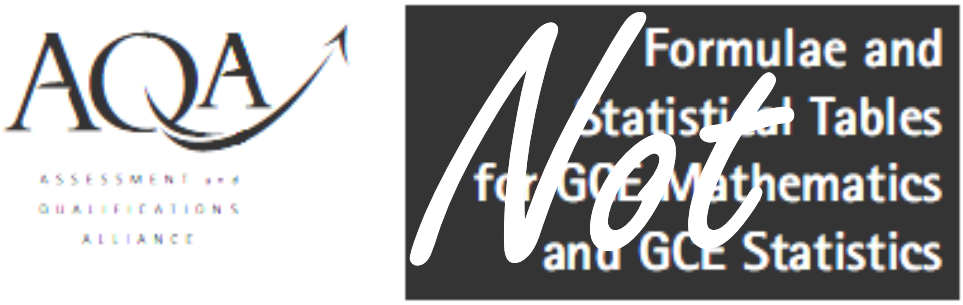

The Not-Formula Book

Decision 1

Everything you need to remember

that the formula book won't tell you

The Not-Formula Book for D1

Everything you need to know for Decision 1 that *won't* be in the formula book

Examination Board: AQA

Brief

This document is intended as an aid for revision. Although it includes some examples and explanation, it is primarily not for learning content, but for becoming familiar with the requirements of the course as regards formulae and results. It cannot replace the use of a text book, and nothing produces competence and familiarity with mathematical techniques like practice. This document was produced as an addition to classroom teaching and textbook questions, to provide a summary of key points and, in particular, any formulae or results you are expected to know and use in this module.

Contents

Chapter 1 – Minimum connectors

A **graph** is a mathematical diagram giving information about connections between points. Points are known as **vertices** or **nodes** and the connecting lines as **edges** or **arcs**. The edges may have a weighting which could represent, for instance, the distance or the travel time between points.

A **minimum spanning tree** of a network always has one edge fewer than the number of vertices in the graph (trivially). It may not be unique, but is the shortest length of arcs (edges) required to link every single node (vertex).

Kruskal's algorithm:
To find a minimum spanning tree for a network with n edges.

Step 1: Choose the unused edge with the lowest value.
Step 2: Add this edge to your tree.
Step 3: If there are n-1 edges in your tree, stop. If not, go to step 1.
NOTE: Ensure you do not create a cycle – if choosing an arc would produce a cycle, don't choose it.

Prim's algorithm:
To find a minimum spanning tree for a network with n edges.

Step 1: From a start vertex draw the lowest valued edge to start your tree. (Any vertex can be chosen as the start vertex; however, it will always be given in an exam question.)
Step 2: From any vertex on your tree, add the edge with the lowest value.
Step 3: If there are n-1 edges in your tree, you have finished. If not, go to step 2.
NOTE: Ensure you do not create a cycle – if choosing an arc would produce a cycle, don't choose it.

Note: Provided all edges have a different weighting, Kruskal's and Prim's both produce the same result, so make a note of the order in which you choose edges to allow the examiner to verify your choice of algorithm. Recall that Kruskal's algorithm adds edges regardless of their connectedness to the previously selected edges. Prim's limits your choice to connected edges.

Matrix form of Prim's algorithm:
Used in situations where the information is too complex to easily draw in diagram form, or when the algorithm is to be applied by a computer program.

Step 1: Label the column corresponding to the start vertex with a 1. Delete the row corresponding to that vertex.
Step 2: Ring the smallest available value in any labelled column.
Step 3: Label the column corresponding to the ringed vertex with a 2, etc. Delete the row corresponding to that vertex.
Step 4: Repeat steps 2 and 3 until all rows have been deleted.
Step 5: Write down the order in which edges were selected and the length of the minimum spanning tree.

Chapter 2 – Shortest path problem (Dijkstra's algorithm)

The minimum spanning tree of the previous chapter is used for connecting vertices with the minimum of distance/cost (eg installing cable for broadband internet), but doesn't give an optimal route from one vertex to another. For this type of analysis, used for planning a route via a road network between two cities, for instance, Dijkstra's algorithm is required.

Dijkstra's algorithm
Used for finding the shortest path through a network.

Step 1: Label the start vertex as 0.
Step 2: Box this number (permanent label).
Step 3: Label each vertex that is connected to the start vertex with its distance (temporary label).
Step 4: Box the smallest number.
Step 5: From this vertex, consider the distance to each connected vertex.
Step 6: If a distance is less than the distance already in this vertex, cross out the distance and write in the new distance. If there was no distance at the vertex, write down the new distance.
Step 7: Repeat from step 4 until the destination vertex is boxed.

Note: At step 4, the 'smallest number' refers to the smallest available temporary label – this does *not* need to be directly connected to the latest vertex to be permanently labelled.

Note: in some networks, called directed networks, sometimes edges will be unidirectional (eg, one-way streets). To apply Dijkstra's algorithm in these cases, only consider edges leading away from the vertex.

If there are **multiple start points**, then we apply Dijkstra's algorithm from the end point until we have reached each of the starting points. In this way we can find the shortest route.

Limitations: If we use Dijkstra's algorithm on a network containing an edge that has a negative value it does not work.

To find the **route** of the shortest path using Dijkstra's algorithm:
As well as listing temporary values, we put a letter after each value, which indicates the preceding vertex on the route. We find the route by backtracking through the network from the finishing point.

Chapter 3 – Chinese postman problem

Dijkstra's algorithm finds the shortest path between two vertices. The Chinese postman problem is an attempt to find the shortest path which traverses *every edge*. The obvious application would be a postman who needs to traverse each road in a given area.

A **traversable** graph is one that can be drawn without taking pen from paper and without retracing the same edge. In such a case the graph is said to have an **Eulerian trail**.

An **Eulerian trail** uses all the edges of a graph. For a graph to be Eulerian all the vertices must be of even order.

If a graph has two odd vertices then the graph is said to be **semi-Eulerian**. A trail can be drawn starting at one of the odd vertices and finishing at the other odd vertex.

Chinese postman algorithm
Used for finding the shortest route which traverses all edges of a network.

To find a minimum Chinese postman route we must walk along each edge at least once and in addition must walk along the least pairings of odd vertices on one extra occasion.

Step 1: List all odd vertices.
Step 2: List all possible pairings of odd vertices.
Step 3: For each pairing find the edges that connect the vertices with the minimum weight.
Step 4: Find the pairings such that the sum of the weights is minimised.
Step 5: On the original graph add the edges that have been found in step 4.
Step 6: The length of an optimal Chinese postman route is the sum of all the edges added to the total found in Step 4.
Step 7: A route corresponding to this minimum weight can then be found by inspection.

Note: This algorithm readily calculates the *length* of the shortest path. The more complex the network, the harder it is to apply step 7 – identifying the *route*.

Before attempting to find the route for the Chinese postman algorithm, first use this method to **identify which vertices appear, and how often**:

Step 1: On the original diagram add the extra edges to make the graph Eulerian.
Step 2: List the order of each vertex. At this stage each vertex will have an even order.
Step 3: The number of times each edge will appear in a Chinese postman route will be half the order of its vertex, with the exception being vertex A (the start/finish vertex), as this will appear on one extra occasion.

Note: There may be problems set where the start and finish vertices do not have to be the same. In a semi-Eulerian graph (exactly 2 odd vertices), the minimum route length therefore will simply be the sum of the edge lengths.

Chapter 4 – Travelling salesman problem

Where the Chinese postman was trying to minimise the distance travelled while traversing every *edge*, the travelling salesman needs to minimise the distance travelled while visiting every *vertex*.

A minimum spanning tree links all vertices, but a **tour** for the travelling salesman must start and finish at the same vertex, meaning that, rather than containing one fewer edges than vertices, it contains the same number.

The optimal tour is defined as a tour of the graph with the shortest path. There is no algorithm for calculating this, and brute force calculations would take prohibitively long, even by computer. Therefore an upper bound is a useful tool.

Any tour that exists is an **upper bound**. The best upper bound is the **lowest upper bound**.

Nearest-neighbour algorithm
Used to find an upper bound for the optimal tour.

Step 1: Choose a start vertex.
Step 2: From your current vertex go to the nearest unvisited vertex.
Step 3: Repeat step 2 until all the vertices have been visited.
Step 4: Return to the start vertex.

Note: By starting at different vertices, different upper bounds can be generated.

Just as an **upper bound** for the length of an optimal tour can be found, a **lower bound** can be calculated. Since it only shows that the shortest tour cannot be below this value, it does not necessarily mean that any tour exists with a value as low as this lower bound. Conversely to the upper bound, the best lower bound is the **greatest lower bound**.

Lower bound algorithm
Used to find a lower bound for the optimal tour.

Step 1: Delete a vertex and all edges connected to the vertex.
Step 2: Find a minimum spanning tree for the remaining network.
Step 3: Add the two shortest edges from the deleted vertex.

Note: You can use either Kruskal's or Prim's to find the minimum spanning tree, required in step 2. An **incomplete network** is one in which not all vertices are linked by edges. This would result in a contradiction should we apply our upper bound or lower bound algorithms.

If any network is **incomplete** then before any upper or lower bounds are obtained, the network must be made complete. Ensure that the distances between all pairs of vertices are represented by a single edge of minimum length.

Chapter 5 – Graph theory

A **graph** consists of a finite number of points connected by lines. Points are normally called **vertices** or **nodes**. Lines are called **edges** or **arcs**.

A **network**, or **weighted graph**, is a graph whose edges are assigned a number. This can represent values such as the distance, the time or the cost relating to travelling between two points.

Two vertices are **connected** if there is an edge between them. A *graph* is **fully connected** if all pairs of vertices are connected, and **connected** if there is a route between all pairs of vertices.

A **simple graph** contains no loops (edges which go from a vertex back to the same vertex) and no duplicate edges (two different edges connecting the same pair of vertices).

The **degree** or **order** of a vertex is the number of edges connected to it.

A **directed graph** or **digraph** is one which has a specified direction for its edges.

A **complete graph (K_n)** is one in which every vertex is connected by *exactly one* edge to each of the other vertices (if connected by more than one edge it is merely fully connected).

A **bipartite** graph has two sets of vertices and the edges only connect vertices from one set to the other.

A **trail** is a sequence of edges of a graph such that the second vertex of each edge is the first vertex of the next edge, with no edge included more than once.

A **path** is a trail such that no vertex is visited more than once (except that the first vertex may be the last).

A **cycle** is a closed path with at least one edge.

A **Hamiltonian cycle** is a cycle that visits every vertex of a graph.

An **Eulerian trail** is a trail that traverses every edge of a graph exactly once. (If a graph possesses an Eulerian trail then the graph itself is called Eulerian. If the graph possesses a non-closed trail that uses all of its edges exactly once then the graph is called **semi-Eulerian**.)

A **tree** is a connected graph with no cycles.

Any tree that connects all the vertices of a graph is called a **spanning tree** for that graph. For a connected graph with n vertices each spanning tree has exactly $n - 1$ edges.

A **minimum spanning tree** is a spanning tree of minimum weight for a network.

A graph may be represented by a matrix, which is called an **adjacency matrix**. Each row and column represent a vertex of a graph and the number in the matrix gives the number of edges joining the pair of vertices.

Chapter 6 – Matchings

Matchings describe how bipartite graphs can be linked.

Pairing vertices of one of the sets to vertices of the other set in such a way that no two edges have a common vertex is called a **matching**

A **maximum matching** is a matching that contains the same number of edges as there are vertices on either of the two sets.

A **complete matching** is a matching that contains the same number of edges as there are vertices on either of the two sets.

Note: Although from any set of information a maximum matching is always possible, a complete matching is not always possible.

Alternating path algorithm
Used to improve a matching to find a maximal matching.

The principle is to start with a vertex in one set that is not in the initial match and alternate between the two sets until we finish at a vertex in the other set that was not in the initial match. This is called an ***alternating path****.*

Step 1: From your initial matching, find a vertex on the left-hand subset not in the initial match and connect this vertex to a vertex on the right-hand set.
Step 2: If the right-hand vertex is not in the initial matching then add this to the initial matching and repeat step 1. If the right hand vertex is in the initial matching go to step 3.
Step 3: Add the new edge to the matching and remove from the initial matching the edge linking this vertex to the left-hand side. Repeat step 1 using the vertex that has just been removed from the matching.
Step 4: Continue in this way until you have a complete matching or there is no further improvement that can be made.

Chapter 7 – Sorting algorithms

These algorithms are examples of different types of sorting methods – each has advantages and disadvantages.

One method of sorting numbers into numerical order is the **bubble sort**. The algorithm is based on comparing successive pairs of numbers. Initially the largest number is located at the end of the list, then the next largest, etc. until the whole list is in order.

Note: Bubble sort starts by comparing items 1 and 2 and swapping their order if necessary. Then it compares the (new) items 2 and 3, and so on. It generally requires a number of sweeps through the list before every value is in its proper place, and this can be verified by checking that the results of two sweeps are identical.

A **shuttle sort** initially compares the first and second numbers, and orders them correctly. It then introduces the third number into the sort and locates that in the correct position, then the fourth into the correct position and so on until the list is fully correct.

Shell sort algorithm
Used to order sets of numbers too large to be efficiently handled by the bubble or shuttle sort.

Step 1: Divide a list into *n/2* sublists, ignoring any remainders (a sublist is part of a list).
Step 2: Shuttle sort each sublist.
Step 3: Merge the sorted sublists.
Step 4: Divide the number of sublists by 2 and repeat steps 2 and 3 until there is only one sublist.

Quick sort algorithm
Used to order sets by sorting into sets larger or smaller than a given 'pivot' number.

Step 1: Use the first number in each list as the pivot.
Step 2: Create new sublist(s) by writing the numbers lower than the pivot to the left and higher numbers to the right (do not change the order of the numbers in the sublist).
Step 3: Go to step 1 and repeat the process on each sublist until each sublist has only one element.

Note: Sometimes a sublist will be empty.

Chapter 8 – Algorithms

Any **algorithm** must have the following properties:

1) There must be a finite number of instructions.
2) Each stage must be defined precisely.
3) Each instruction must be precise.
4) Each answer must depend only on the values input to solve a particular problem.
5) The algorithm must work for any set of values that are input.

When using **flow charts**:

1) Oval boxes are used for starting, stopping, inputting or outputting data.
2) Square boxes are used for calculations or instructions.
3) Diamond-shaped boxes are used for decisions.

When an algorithm is written as a **series of instructions**:

1) The instructions are written in a simplistic form and are prefixed with line numbers.
2) The line numbers themselves do not affect the working of an algorithm;
   they are an indication as to the order of the instructions.

Note: By convention, line numbers are written in multiples of 10, so that amendments or corrections can be inserted without needing to renumber the lines.

Print indicates the production of an output value.

Goto can be used to jump to a particular line depending on given conditions.

Chapter 9 – Linear programming

Linear programming problems in industry often have many constraints and variables, and require a large number of equations. We will be dealing with problems in only two dimensions, and displaying inequalities on standard co-ordinate axes as a graph.

When an inequality is displayed on a graph, the **feasible region** is indicated by being **unshaded**, the **excluded region** by the **shaded** part of the graph.

A **linear programming** problem must first be **formulated** by defining an **objective function**.

Eg:

$$\begin{aligned} &Maximise \quad P = 5x + 12y \\ &subject\ to \quad 2x + y \leq 20 \\ &x + 2y \leq 12 \\ &x + 3y \leq 15 \\ &x \geq 0 \\ &y \geq 0 \end{aligned}$$

The **feasible region** can be illustrated on a graph as shown, and the objective function can be drawn as a family of lines – increase P until the line is on the point of leaving the feasible area. This point will give the x and y values which maximise P: