OCR Decision 1 Module Revision Sheet

The D1 exam is 1 hour 30 minutes long. You are allowed a graphics calculator.

Before you go into the exam make sure you are fully aware of the contents of the formula booklet you receive. Also be sure not to panic; it is not uncommon to get stuck on a question (I've been there!). Just continue with what you can do and return at the end to the question(s) you have found hard. If you have time check all your work, especially the first question you attempted... always an area prone to error.

J.M.S.

Algorithms

- An **algorithm** is defined to be a *finite* sequence of instructions for solving a problem.
- Flow diagrams
- The **Bubble sort** algorithm orders numerical data in descending order.

  1. If there is only one number in the list then stop.
  2. Make one pass down the list, comparing numbers in pairs and swapping as necessary.
  3. If no swaps have occurred then stop. Otherwise, ignore the last element of the list and return to 1.

  Note that the first pass will guarantee that the largest number is at the bottom, which is why we can then ignore it.

- The **Shuttle sort** algorithm also orders numerical data in descending order. It is rather better than bubble sort on average.

  1. Compare the 1st and 2nd numbers in the list and swap if necessary.
  2. Compare the 2nd and 3rd numbers in the list and swap if necessary. If a swap has occurred, compare the 1st and 2nd numbers and swap if necessary.
  3. Compare the 3rd and 4th numbers in the list and swap if necessary. If a swap has occurred, compare the 2nd and 3rd numbers, and so on up the list.

  And so on through the entire list.

- First fit

  1. Place each object in turn in the first available space in which it will fit.

  Nice and simple...

- First fit decreasing

Graphs

- Rather a lot of highly tedious definitions to know I'm afraid. A **graph** is made up of **nodes/vertices** and connected by **arcs/edges**.
- (There can exist **loops** where an arc comes back to the same node it started at and **multiple arcs** where more than one arc connects two nodes, but usually you won't have to worry about these.) A **simple** graph is one without loops and multiple arcs.
- A **subgraph** is any 'subset' of a given graph. I don't know if a graph is its own subgraph. I also don't know if no graph is a subgraph of a given graph.
- A **complete graph** is one where every node is connected to every other node by exactly one arc. The notation K_n is used for the complete graph of n nodes. Every simple graph with $\leqslant n$ nodes is a subgraph of K_n. The number of arcs in K_n is $\frac{n(n-1)}{2}$; make sure you know why.
- A **bipartite graph** is one where a set of r nodes are connected to s nodes such that no nodes in r are connected to one another, similarly for s. A **complete bipartite graph** is one where every node in r is connected to every node in s and is denoted $K_{r,s}$. The number of nodes in $K_{r,s}$ is $r + s$ and the number of arcs is rs.
- A **trail/route** is a 'journey' taken from node-to-node in a graph. Repetitions of nodes are allowed.
- A **path** is a trail where no node is passed more than once.
- A **closed trail** is one where the start and end nodes are the same. Repetition of nodes allowed.
- A **cycle** is a closed trail where repetitions of nodes are not allowed (except the start and end nodes).
- The **order** of a node is the number of arcs that lead away from the node. They can be **odd** and **even** order.
- A **connected graph** is one where a path (not necessarily direct) can be found from any two nodes.
- An **Eulerian graph** is a connected graph with a closed trail containing every *arc* exactly once. This occurs if and only if *every node is of even order.*

  Informally this means you can draw the shape without taking your pen off the page, not going over previously drawn lines (arcs), with your pen ending up in the same place (node) you started.
- A **semi-Eulerian graph** is a connected graph which has a trail (crucially not closed) that contains every arc exactly once. This occurs if and only if *exactly two nodes have odd order.*

  Informally this means you can draw the shape without taking your pen off the page, not going over previously drawn lines (arcs), with your pen ending up in a different place (node) to where you started.
- A **planar graph** is one which can be drawn in a plane (2D Surface) such that arcs only meet at nodes; i.e. the arcs don't cross one another. It should be noted that some graphs can appear non-planar, but can be re-drawn to be planar.

Euler's relationship relates the number of of regions R, nodes N, and arcs A for any *connected, planar* graph. It states

$$R + N = A + 2.$$

Note that the region outside the graph is counted as a region.

- A **tree** is a connected graph with no cycles. Any *connected* graph contains at least one subgraph which is a tree. This is very important later on for Prim & Kruskal.

Networks

- Arcs in a graph can sometimes be given a value or **weight** (i.e. a 'cost' or 'time' or value to move along the arc). A graph with weights is called a **network**.
- A **digraph** is a directed graph where one can only move in specified directions around the network.
- **Matrices** can be useful in representing the weights to get between different nodes. For a non-directed graph there should always be reflectional symmetry around the leading diagonal of the matrix. For a digraph you must be careful to show the 'From' and 'To'.

Prim's Algorithm

- Prim's algorithm can be applied to a network to discover the minimum spanning tree T. For a graph follow:

  1. Select any node to be the first node of T.
  2. Consider the arcs which connect nodes in T to nodes outside T. Pick the one with minimum weight. Add this arc and the extra node to T. (If there are two or more arcs of minimum weight, choose any one of them.)
  3. Repeat 2 until T contains every node of the graph.

- For a matrix Prim's algorithm works like this:

  1. Select any node to be the first node of T.
  2. Circle the new node of T in the top row, and cross out the row corresponding to this new node.
  3. Find the smallest weight left in the columns corresponding to the nodes of T, and circle this weight. Then choose the node whose row the weight is in to join T. (If there are several possibilities for the weight, choose any one of them.)
  4. Repeat 2 and 3 until T contains every node.

Kruskal's Algorithm

- Like Prim, Kruskal's algorithm discovers the minimum spanning tree. Formally it states:

  1. Choose the arc of least weight.
  2. Choose from those arcs remaining the arc of least weight which does *not* form a cycle with already chosen arcs. (If there are several such arcs, choose one arbitrarily.)
  3. Repeat 2 until $n - 1$ arcs have been chosen.

Informally you look at the network and pick out the lowest weighted arc and add it to your set. Ensuring that you avoid any cycles you keep picking the lowest weighted arcs until you have a spanning tree.

There is no matrix method that I know of for Kruskal. If presented with a matrix then draw out the network.

Dijkstra's Algorithm

- **Dijkstra's** algorithm finds the shortest path from a node to other nodes in a network. The moment you create a permanent label you know that the shortest route from the starting node to that node has been found. Formally it states:

  1. Label the start node with zero and box this label.
  2. Consider the node with the most recently boxed label. Suppose this node to be X and let D be its permanent label. Then, in turn, consider each node directly joined to X but not yet permanently boxed. For each such node, Y say, temporarily label it with the lesser of $D +$ (the weight of arc XY) and its existing label (if any).
  3. Choose the least of all temporary labels on the network. Make this label permanent by boxing it.
  4. Repeat 2 and 3 until the destination node has a permanent label.
  5. Go backwards through the network, retracing the path of shortest length from the destination node to the start node.

- Here you should make use of these boxes to keep track of your progress; an examiner will also use these to mark your exam.

8	4
$\cancel{6}$ $\cancel{5}$ 4	

This means that this node was the eighth node to be made permanent and it's shortest distance from the starting node was 4. Whilst discovering that 4 was the best there were temporary distances of 6 and 5 that were improved upon.

Travelling Salesperson

- A **Hamiltonian cycle** is defined as a tour which contains every node precisely once. It is often useful to find the Hamiltonian cycle of least total weight. This is a hard problem of factorial order.

- The **nearest neighbour** algorithm finds a reasonably good Hamitonian cycle. It does *not* necessarily find the best.

  1. Choose any starting node.
  2. Consider the arcs which join the previous chosen node to not-yet-chosen nodes. From these arcs pick one that has minimum weight. Choose this arc, and the new node on the end of it, to join the cycle.
  3. Repeat 2 until all nodes have been chosen.
  4. Then add the arc that joins the last-chosen node to the first-chosen node.

Informally you start at a node and run to the next node along the arc of least weight avoiding nodes you've been to before and keep doing that until you run out of nodes. Then go back to the start.

- A lower bound can be found with the **Lower Bound** algorithm.

  1. Choose an arbitrary node, say X. Find the total of the two smallest weights of arcs incident at X.
  2. Consider the network obtained by ignoring X and all arcs incident to X. Find the total weight of the minimum connector for this network.
  3. The sum of the two totals is a lower bound.

- The **Tour Improvement** algorithm can be used to improve a discovered Hamiltonian tour.

  1. Let $i = 1$.
  2. If $d(V_i, V_{i+2}) + d(V_{i+1}, V_{i+3}) < d(V_i, V_{i+1}) + d(V_{i+2}, V_{i+3})$ then swap V_{i+1} and V_{i+2}.
  3. Replace i by $i + 1$.
  4. If $i \leqslant n$ then go back to 2.

Route Inspection

- The **Chinese Postman** algorithm is a method to find the least-weight closed trail containing every arc (note arc and *not* node). It is so called because a postman needs to travel along roads (arcs) when delivering letters. The algorithm states

  1. Find all nodes of odd order.
  2. For each pair of odd nodes find the connecting path of minimum weight.
  3. Pair up all the odd nodes so that the sum of the weights of the connecting paths from 2 is minimised.
  4. In the original graph, duplicate the minimum weight paths found in 3.
  5. Find a trail containing every arc for the new (Eulerian) graph.

- If you have four odd order nodes $\{A,B,C,D\}$ then there are $3 \times 1 = 3$ comparisons to make: $AB - CD$, $AC - BD$, $AD - BC$.

  If you have six odd order nodes $\{A,B,C,D,E,F\}$ then you have $5 \times 3 \times 1 = 15$ comparisons to make: $AB-CD-EF$, $AB-CE-DF$, $AB-CF-DE$, $AC-BD-EF$, $AC-BE-DF$, $AC-BF-DE$, $AD-BC-EF$, $AD-BE-CF$, $AD-BF-CE$, $AE-BC-DF$, $AE-BD-CF$, $AE-BF-CD$, $AF-BC-DE$, $AF-BD-CE$, $AF-BE-CD$.

  If you have $2n$ odd order nodes then you need to make $(2n-1) \times (2n-3) \times \ldots 3 \times 1$ comparisons. In an exam situation I can't believe they would make you do more than four odd ordered nodes.

Linear Programming

- The **Simplex** algorithm solves...