# A Simple Rule-Based Assembly Line Sequencer

**Ian Hogan**[1]

**Abstract**

This article describes a simple rule-based assembly line sequencer which was developed and implemented at the Nissan Australia manufacturing plant in the late 1980's. The aim of this article is to illustrate how a simple algorithm resulted in a major improvement in assembly line sequencing. It also shows how implementing this algorithm in a database language integrated the results easily into routine operations. Finally, it briefly covers related work in minimising colour changes in the paint-shop and sub-assembly planning.

**Keywords**: Heuristic Algorithm, Production Sequencing, Rule-based systems

## The Problem

This work was part of a larger project to automate the production planning and scheduling tasks for the Nissan Australia vehicle assembly plant at Clayton, Victoria. The production plan, which was driven by dealer demand, determined the mix of models scheduled to start assembly each day. One requirement of the project was to produce the build list for the day – that is, what vehicle to commence building in what order.

Prior to the development of the algorithm described below, the sequence was produced by hand. Essentially a set of job cards was produced, each of which listed the vehicle serial number and its model code, options, colour and trim. A production supervisor would then "shuffle" the cards according to a set of explicit and implicit rules. These rules were different for each of the two body assembly lines. The rules related to the work rates of the various sub-assembly areas and the work content of the different vehicle models.

For example, on the Pintara / Skyline body assembly line, rules included that every third vehicle should be a station-wagon, automatic and manual vehicles should be evenly distributed and that top range vehicles should be well spaced throughout the assembly line, rather than being bunched together.

## The Algorithm

The basis of this heuristic algorithm was to categorise the vehicles according to a hierarchical set of rules. This can be viewed as a tree with the branch nodes corresponding to the rules and the leaf nodes corresponding to the different vehicle sub-categories. In addition to the rules specifying the branching decisions, a branch *order* and *frequency* were specified. This is summarised in Figure 1, and the following notes:

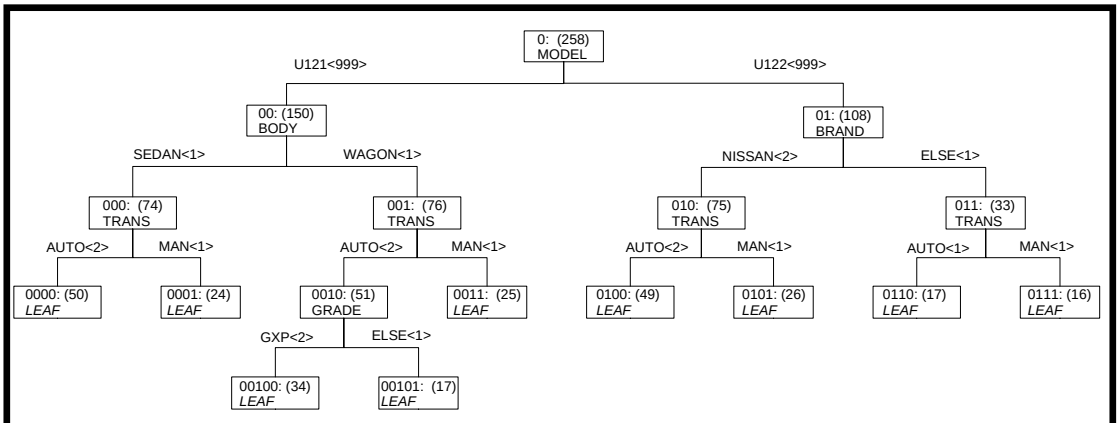


***Figure 1: Example Rule-Set.***

[1] THINKronicity P/L, Melbourne, Victoria, Australia (Email: Ian.Hogan@THINKronicity.com.au)

- The set of leaf nodes constitute a disjoint partition of the set of vehicles to be called up for each day's production – that is, each vehicle will meet the criteria for exactly one leaf node.

- The number in the top left corner of each node is the node id, and corresponds to the path from the root node to that node.

- The number in brackets indicates the number of vehicles meeting the criteria for that path.

- The word in the node is either LEAF for a leaf node, or the name of a vehicle attribute used for the decision rule.

- The word on the arc between nodes indicates the value of the vehicle attribute corresponding to that path, or the word ELSE for values not matching the ones specified on the other branches.
- The number in angle brackets is the branch frequency – that is, the number of consecutive traversals of that branch permitted.

To select each vehicle, the program traversed the tree from the root to a leaf node in a recursive fashion. Pseudo-code for the algorithm is shown in Figure 2.

A worked example of the algorithm is shown in Appendix I, so its working can be fully understood. For simplicity a single vehicle model is used in the example. Appendix II shows the Python program used to elaborate the sequence for Appendix I.

**Implementation Issues**

The main challenge in implementing this algorithm was the language used for the production planning system – Adabas Natural – which was designed for business databases. For example, at that time Natural only had single dimensioned arrays and as I recall did not support recursive functions. However once the algorithm was coded, execution time was only a matter of seconds to sequence a day's production.

```
Main:
        SequenceNumber = 1
        Repeat
                RecordVehicle(SequenceNumber, GetNextVehicle(root-node))
                 SequenceNumber  = SequenceNumber + 1
        Until root-node.UnAllocatedCount = 0

GetNextVehicle(node):

                If node.UnAllocatedCount=0 Then:
                        Return GetNextVehicle(node.parent)

                ElseIf node.Type = Leaf Then
                        Decrement(node.UnAllocatedCount)
                        Return node.NextVehicle

                Else:
                        Return GetNextVehicle(GetNextActiveBranch(node))

GetNextActiveBranch(node):
        If node.ActiveBranch.UsedCount < node.ActiveBranch.RepeatCount Then:
                Increment(node.ActiveBranch.UsedCount)
        Else:
                node.ActiveBranch =  (node.ActiveBranch+1) Mod node.BranchCount
                node.ActiveBranch.UsedCount = 1

        Return node.ActiveBranch

RecordVehicle(SequenceNumber , VehicleId):
        Insert  (SequenceNumber , VehicleId) Into VehicleCallupTable
```

***Figure 2: Pseudo-Code for the Algorithm***

This fast execution time meant that, if required, the rules could be adjusted and the effect on the production sequence could be seen very quickly. This was an important issue at the start of the process when the rules were being developed. The rules editor presented the user with a view of a single node in the decision-tree, and allowed them to adjust the parameters of that node, and easily navigate to parent or sub-nodes.

Once a sequence schedule was built for a day, the production management staff were able to view the sequence on-screen, and adjust it if they felt that fine-tuning was needed. Once they were satisfied with the sequence, they printed bar-coded cards and job-schedule reports which were issued to the factory floor staff. The daily sequencing process is illustrated below in Figure 3.

Automating this task reduced what had been a two hour or more manual task to about 15 minutes work or less.

**The Acid Test**

One of the interesting elements of this task was seeing how the end-users would work with the decision-tree to create the required build-sequence. The main user told me she had never heard of the word *node* before, other than the thing you blew when you "*had a bad cold in da node*"! The challenge was really apparent when I left the company and moved interstate shortly after the implementation of this work. However, I re-joined the company about 8 months later after returning from interstate, and found that the production planning staff had taken to the system very well, and had adjusted the rules to suit the needs of the factory with different models coming in. Having been closely involved in the design and implementation of the system, they had a strong sense of ownership and engagement.

**Related Work**

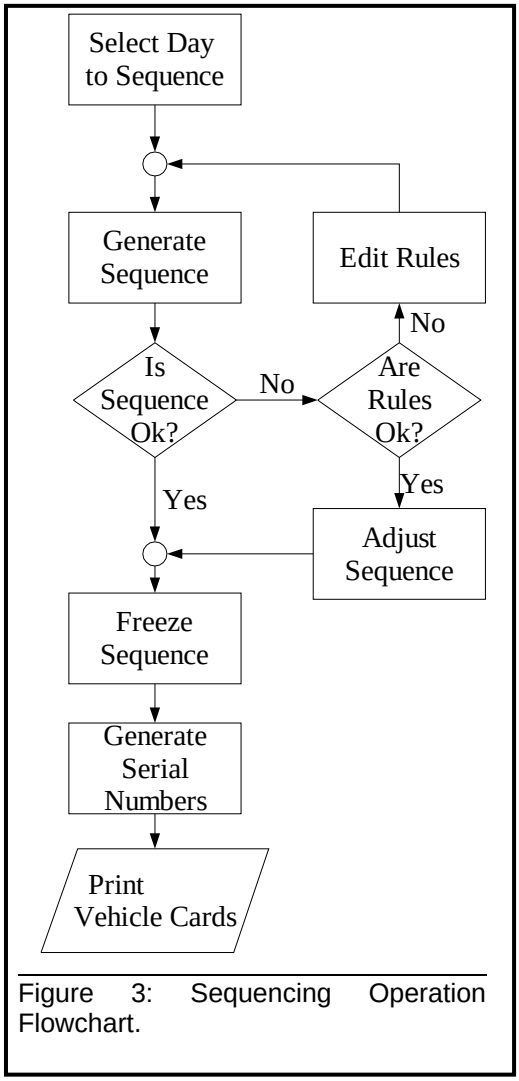


Figure 3: Sequencing Operation Flowchart.

As part of the new production planning system, the opportunity was taken to reduce the frequency of paint colour changes in the spray painting area. As there is both a time and material cost to each paint change, this would improve efficiency in this area. The mechanism to achieve this reduction was devised by the production manager, and is in the "crude but effective" category. As each week's vehicle production was allocated, the vehicles for that week were sorted by paint colour, with alternate weeks going from light to dark and dark to light. As around half of the plant output was white vehicles this made for quite a number of all-white vehicle days in the plant.

As an aside, it was interesting to watch the demeanour of the production manager during the introduction of this paint-sorting feature. On the first day that there were only white vehicles in the plant he was delighted. However as the week progressed and four days later there were still only white cars in the plant, he was beginning to get worried! However investigations revealed that there had been a subconscious bias in the manual planning process, and a backlog for white cars had built up. Once this backlog worked its way out of the system, the process was a great success.

Another aspect of the system was introduction of the bar-code vehicle tags, which were exchanged at the major break-points in the production line. As the cards were brought back to the planning office they were read into the computer system, and hence the progress of each vehicle was tracked through the production line. Note, due to some vehicles being side-lined for special parts, re-work or the need for two-tone paintwork, vehicles do not all progress through the production line at the same rate. This tracking information was then used to drive a 3-day forecast for material supply and sub-assembly areas. This helped reduce delays due to material shortages, and helped with a move towards JIT (Just in time) production.

## Conclusion

From these examples it can be seen that it is possible to dramatically improve production operations just with simple, but appropriate, models and algorithms. The dynamic nature of many production facilities would seem to prohibit the use of traditional optimisation techniques for day-to-day operations. This is not to deny their importance in long-range production and capacity planning.

## Acknowledgements

I would like to thank Nissan Australia for permission to document this work. In particular I would like to thank Mr Bandu Dissanayake[2], not only for his assistance in obtaining this permission, but also for his insightful management during my time at Nissan. I would also like to thank the IT and Production Planning staff at for their support during this development.

2. National Manager – Systems Development, Information Technology, at Nissan Motor Co. (Australia) Pty Ltd.

[1] THINKronicity P/L, Melbourne, Victoria, Australia (Email: Ian.Hogan@THINKronicity.com.au)

## Appendix I: Example Rules and Sequence

The figure below illustrates the sequencing algorithm for a single vehicle type. The arrows show the first 5 steps in the sequencing operation. At the bottom of the branch nodes is a table showing the active branch for the sequencing step.

Table 1 on the next page shows the generated sequence. As can be observed, no swaps are required to achieve the desired goals – alternating sedans and wagons, two automatic and one manual transmission per group of three vehicles and spacing out the non-GXP grades.

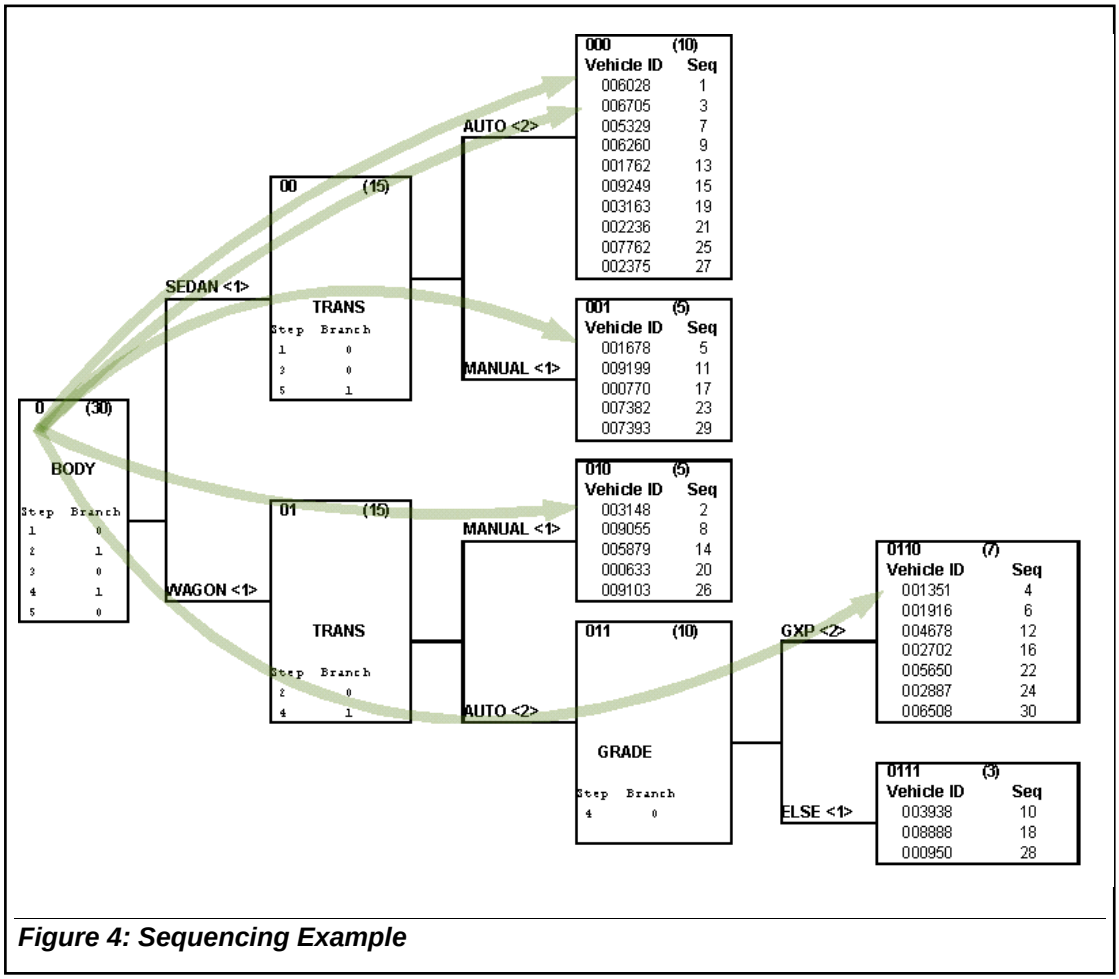


***Figure 4: Sequencing Example***

The Python program used to generate this sequence is shown in Appendix II. Details on the Python language can be found at http://www.python.org/. The distribution used was the ActivePython 2.2.3 (Build 227) from ActiveState (http://www.activestate.com/).

**Table 1 : Generated Production Sequence**

| Sequence | Vehicle ID | Type |
|---|---|---|
| 1 | 006028 | SA |
| 2 | 003148 | WM |
| 3 | 006705 | SA |
| 4 | 001351 | WAG |
| 5 | 001678 | SM |
| 6 | 001916 | WAG |
| 7 | 005329 | SA |
| 8 | 009055 | WM |
| 9 | 006260 | SA |
| 10 | 003938 | WAE |
| 11 | 009199 | SM |
| 12 | 004678 | WAG |
| 13 | 001762 | SA |
| 14 | 005879 | WM |
| 15 | 009249 | SA |
| 16 | 002702 | WAG |
| 17 | 000770 | SM |
| 18 | 008888 | WAE |
| 19 | 003163 | SA |
| 20 | 000633 | WM |
| 21 | 002236 | SA |
| 22 | 005650 | WAG |
| 23 | 007382 | SM |
| 24 | 002887 | WAG |
| 25 | 007762 | SA |
| 26 | 009103 | WM |
| 27 | 002375 | SA |
| 28 | 000950 | WAE |
| 29 | 007393 | SM |
| 30 | 006508 | WAG |

## Appendix II: Python Sequencing Simulator

```
#=================================================================
# Simple production line sequencer simulator
#
# Written by Ian Hogan, April 2004.
#
# (C) 2004, THINKronicity Pty Ltd.
#=================================================================

# The global NodeList contains the list of nodes in the rule tree
# as a dictionary, where the key is the node id and the value is
# the node object.
gNodeList = dict()

#=================================================================
# Define class to encapsulate rule-node behaviour
#=================================================================
class qNode:
    """ Class qNode encapsulates rule-node behaviour

    Attribute        Meaning
    -----------      ------------------------------------------------
    id               Used to construct the child node id's.  Node id's are
                     a sequence of digits, one digit for each level in the
                     tree.  Hence the current node's parent is just the
                     current node's id with the last digit removed, and the
                     id for a child node is the current node's id with the
                     branch number appended.
    Vehicles         The number of the vehicles in leaf nodes below this node,
                     if this is a branch node, or the number of vehicles in
                     this node if it is a leaf node.
    VehType          If vehType is None, then it is a branch node
                     and BranchFreq indicates the number of branches
                     and their frequencies, otherwise it is a leaf node for
                     that type of vehicle.
    BranchFreq       A list of frequencies for the branches.
    CurBranch        The current active branch from this node, if it is a
                     branch node.
    CurBranchCount   The number of consecutive times the current active branch
                     has been used.
    UnallocatedCount The number of un-allocated vehicles below this branch node
                     or in this leaf node.
    -----------      ------------------------------------------------
    """

    def __init__(self, id, Vehicles, VehType=None, BranchFreq=None):
        """ Initialise the node object and add it to the global nodelist
        """
        # save data passed in
        self.id = id
        self.Vehicles = Vehicles
        self.VehType = VehType
        self.BranchFreq = BranchFreq
        # Initialise status
        self.CurBranch  = 0
        self.CurBranchCount = 0
        self.UnallocatedCount = Vehicles

        # Add to nodelist
        gNodeList[id] = self

    def BranchCount(self):
        """ Get a node's number of branches
        """
        if self.BranchFreq is None:
            return 0
        else:
            return len(self.BranchFreq)

    def ParentId(self):
        """ Get a node's parent's Id
        """
```

```
        return self.id[0:len(self.id)-1]

    def ChildId(self, Branch):
        """ Get a node's child for a specified branch
        """
        return self.id+str(Branch)

    def Parent(self):
        """ Get a node's parent
        """
        if gNodeList.has_key(self.ParentId()):
            return gNodeList[self.ParentId()]
        else:
            return None

    def Child(self, Branch):
        """ Get a node's child for a specified branch
        """
        if gNodeList.has_key(self.ChildId(Branch)):
            return gNodeList[self.ChildId(Branch)]
        else:
            return None

    def IsLeaf(self):
        """ See if this is a leaf node
        """
        return self.BranchFreq is None

    def IsRoot(self):
        """ See if this node is the root of the tree
        """
        return (self.Parent() is None)

    def DropUnallocatedCount(self):
        """ Reduce the unallocated count for the current node and up the tree
        """
        # Recursively step through nodes from current to root,
        # decrementing the unallocated count for each.
        self.UnallocatedCount -= 1
        if not self.IsRoot():
            self.Parent().DropUnallocatedCount()

    def GetNextActiveBranch(self):
        """ Get the next active branch from this node
        """
        # See if current branch consecutive traversal limit has been reached
        if self.CurBranchCount < self.BranchFreq[self.CurBranch]:
            # Limit not reached - count traversal and still use current branch
            self.CurBranchCount += 1
        else:
            # Limit reached - get next branch and count traversal
            self.CurBranch = (self.CurBranch + 1) % self.BranchCount()
            self.CurBranchCount = 1
        # Return child node for the active branch.
        return self.Child(self.CurBranch)

    def GetNextVehicle(self):
        """ Get the next vehicle for a node
        """
        # Any vehicles left in this node?
        if self.UnallocatedCount == 0:
            # No vehicles left - go back to parent
            return self.Parent().GetNextVehicle()
        elif self.IsLeaf():
            # Leaf node - drop count and return vehicle type.
            self.DropUnallocatedCount()
            return self.VehType
        else:
            # Branch node - get vehicle from next active branch.
            return self.GetNextActiveBranch().GetNextVehicle()


#=============================================================
```

## Appendix II: Python Sequencing Simulator

```
# Main code
#============================================================

# Create nodes
Node0    = qNode(id='0',    Vehicles = 30, VehType = None,  BranchFreq = (1,1))
Node00   = qNode(id='00',   Vehicles = 15, VehType = 'S',   BranchFreq = (2,1))
Node01   = qNode(id='01',   Vehicles = 15, VehType = 'W',   BranchFreq = (1,2))
Node000  = qNode(id='000',  Vehicles = 10, VehType = 'SA',  BranchFreq = None)
Node001  = qNode(id='001',  Vehicles = 5,  VehType = 'SM',  BranchFreq = None)
Node010  = qNode(id='010',  Vehicles = 5,  VehType = 'WM',  BranchFreq = None)
Node011  = qNode(id='011',  Vehicles = 10, VehType = 'WA',  BranchFreq = (2,1))
Node0110 = qNode(id='0110', Vehicles = 7,  VehType = 'WAG', BranchFreq = None)
Node0111 = qNode(id='0111', Vehicles = 3,  VehType = 'WAE', BranchFreq = None)

# Sequence vehicles
iSeq = 1
while Node0.UnallocatedCount > 0 and iSeq <= 30:
    print iSeq, Node0.GetNextVehicle()
    iSeq += 1
```