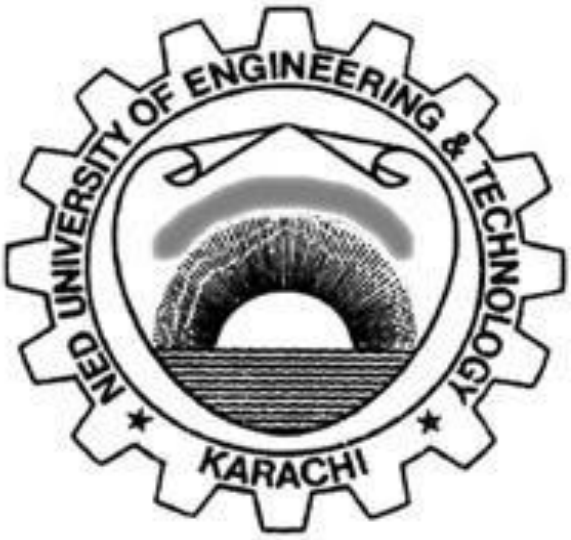
# Practical Workbook
# CS-323
# Artificial Intelligence




Name : _____

Year : _____

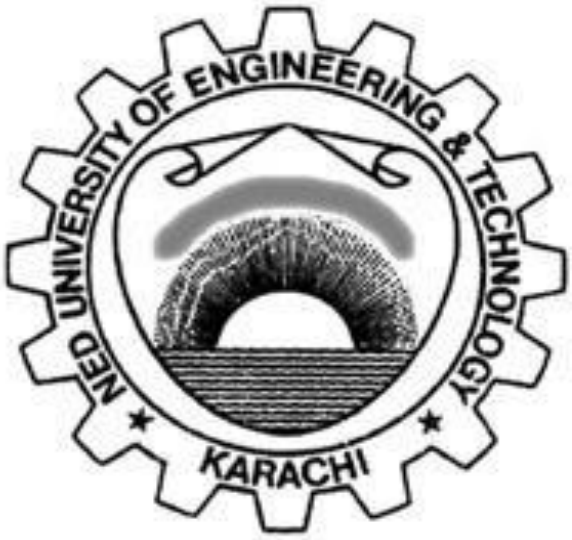
Batch : _____

Roll No : _____

Department: _____

**Department of Computer & Information Systems Engineering**
**NED University of Engineering & Technology,**

# Practical Workbook
# CS-323
# Artificial Intelligence




***Revised by:***

**Ms. Anita Ali**
**Dr. Saad Qasim Khan**
**Ms. Ibshar Ishrat**
**Ms.Tahreem Khan**

***Revised in:***

**August 2024**

**Department of Computer & Information Systems Engineering**
**NED University of Engineering & Technology**

# INTRODUCTION

The Laboratory Workbook supports the Practical Sessions of the course Artificial Intelligence (CS-323). The Workbook has been designed to cover the major areas of Artificial Intelligence including Expert Systems, Machine Learning, and Fuzzy Logic Systems.

The Course Profile of CS-323 Artificial Intelligence lays down the following Course Learning Outcome:

> **"Demonstrate** the use of modern tools and techniques for developing intelligent systems.(C3, PLO-3)"

All lab sessions of this workbook have been designed to assist the achievement of the above CLO. A rubric to evaluate student performance has been provided at the end of the workbook.

First five lab sessions of this workbook are related to Machine Learning algorithms using Artificial Neural Networks (ANN), which is a problem-solving paradigm, used to solve complex, non-linear problems where conventional algorithm solution is either not possible or not feasible. The section begins with laboratory session on implementation of basic logic function, and is followed by methods of creating and working on ANNs. Next lab session describes problems solving phases of ANNs; and finally, the effect of external have been observed on the performance of ANNs.

Lab sessions 6 and 7 explores the topics of Graph solving and searching options using Graph Solving tool.

Lab session 8 is based on consistency based CSP solver whereas Lab session 9 explores Stochastic local search based CSP solver.

Lab sessions 10 and 11 covers the basic and advanced concepts of developing Expert Systems. Lab session 12 covers the data-driven programming in Expert Systems.

Lab session 13 explains how to build Fuzzy Logic based applications using MATLAB Fuzzy Logic Toolbox. It also covers tools such as Fuzzy Tech for building these applications.

Lab 14 contains the complex engineering activity which needs to be carried out as per directions of lab instructor

# CONTENTS

# Lab Session 01

## *Implementing Simple Neural Network Using Perceptron*

## Artificial Neural Network

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well.
Artificial Neural Networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained Artificial Neural Network can be thought of as an "expert" in the category of information it has been given to analyze. This expert can then be used to provide projections given new situations of interest and answer "what if" questions.
Other advantages include:

1. *Adaptive learning:* An ability to learn how to do tasks based on the data given for training or initial experience.
2. *Self-Organization:* An ANN can create its own organization or representation of the information it receives during learning time.
3. *Real Time Operation:* ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. *Fault Tolerance via Redundant Information Coding:* Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

## A Simple Neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.

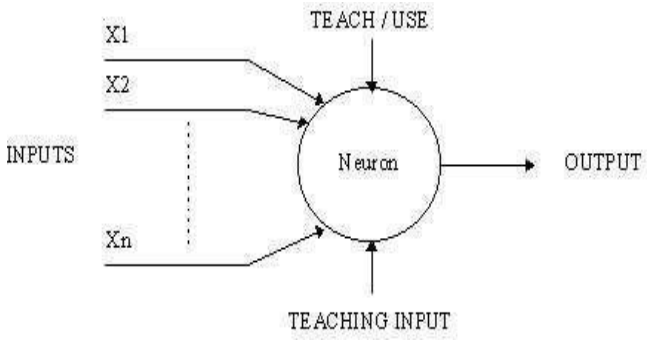



**Figure 1.1:** *A simple neuron*

## Network layers

The commonest type of Artificial Neural Network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of **"output**" units. (See figure 1.2)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behavior of the output units depends on the activity of the hidden units and the eights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

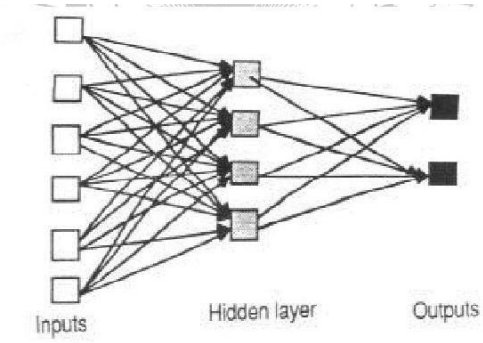



**Figure 1.2:** *A Simple Feed Forward Network*

We also distinguish single-layer and multi-layer architectures. The single-layer organization, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organizations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

## Perceptron

The most influential work on neural nets in the 60's went under the heading of 'perceptron' a term coined by Frank Rosenblatt. The perceptron (See figure 1.3) turns out to be an MCP model ( neuron with weighted inputs ) with some additional, fixed, pre-processing. Units labeled A1, A2, Aj , Ap are called association units and their task is to extract specific,

Localized featured from the input images. Perceptron mimics the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

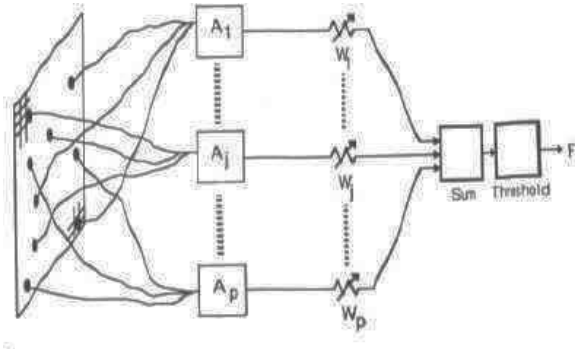



**Figure 1.3:** *A Perceptron*

# Transfer Functions

The behavior of an ANN depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- For ***linear*** (or ramp) the output activity is proportional to the total weighted output.
- For ***threshold units***, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.
- For ***sigmoid units***, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations.

To make an Artificial Neural Network that performs some specific task, we must choose how the units are connected to one another and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

# Implementation of Logic Functions

In this practical we will learn how basic logic functions can be implemented and trained, using MS Excel. The procedure is explained by implementing a 2-input AND gate.
First of all you have to include following columns in your Excel sheet:

$X_i$ : (column for inputs)
Z : (column for true output)
Y : (column for computed output)
D : (column for keeping track of the difference between true output & computed one)
$W_i$ : (column for initial weights, assigned arbitrarily in the first step)
Wf : (column for final weights, which is computed from initial weight and becomes the initial weight of the first step)

# Procedure

1. In input columns $X_1$ and $X_2$, include all possible values which can be provided to a 2-input AND gate, and in column *Z*, list all expected results.
2. In initial weights columns, $W_1$ & $W_2$, arbitrarily enter any values.
3. Apply following formula to Y (column for computed output);

$$Y = \Sigma W_i \ . \ X_i + bias$$

Where; *bias* is any constant (less than 1 for implementation of logic functions).

| S | $X_1$ | $X_2$ | Z | W1i | $W_{2i}$ | Y | D | $W_{1f}$ | $W_{2f}$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0.1 | 0.1 | 0 | 0 | 0.1 | 0.1 |
| | 0 | 1 | 1 | | | | | | |
| | 1 | 0 | 1 | | | | | | |
| | 1 | 1 | 1 | | | | | | |

**Figure 1.4:** *A portion of the Excel Sheet*

4. Calculate the difference between true and computed outputs, using the formula

$$D = Z - Y$$

5. Calculate final weights by applying the following formula.

$$W_f = W_i + \alpha D X_i$$

Where **α** is the learning rate, which is arbitrarily assigned and preferably kept lesser than 0.5

6. Final weights computed in for first set of inputs are passed on as initial weights for second set of inputs, for the same iteration.
7. It is observed that after completing the first iteration, values of true and computed do not match for each possible set of inputs, i.e. difference is non-zero, so the process is repeated up to the point where this difference becomes zero.

**Note:** *If 0.1 is selected as initial weights and learning rate is kept 0.2, and bias is set to zero, then result is obtained in 4th iteration.*

## Logic OR-Gate

Same method is followed for the implementation of logic OR-function.

## Logic NOR & NAND Implementation

For the implementation of NOR and NAND gates, a positive bias is added to the weighted sum of inputs.

## EXERCISES

1. Complete the following tables.

a.

| Operation | Weights | | Learning Rate | Iterations Required |
|---|---|---|---|---|
| | $W_1$ | $W_2$ | | |
| OR | 0.1 | 0.1 | 0.2 | 4 |
| | 0.9 | 0.8 | 0.2 | 1 |
| | 0.5 | 0.5 | 0.2 | 2 |
| | 0.2 | 0.4 | 0.2 | 3 |
| | -0.3 | -0.5 | 0.2 | 5 |

b.

| Operation | Weights | | Learning Rate | Iterations Required |
|---|---|---|---|---|
| | $W_1$ | $W_2$ | | |
| AND | 0.9 | 0.9 | 0.2 | 3 |
| | 0.1 | 0.1 | 0.2 | 2 |
| | 0.5 | 0.5 | 0.2 | 1 |
| | 0.9 | 0.7 | 0.2 | 3 |
| | -0.7 | -0.8 | 0.4 | 4 |

c.

| Operation | Weights | | Learning Rate | Bias | Iterations Required |
|---|---|---|---|---|---|
| | $W_1$ | $W_2$ | | | |
| NAND | 0.3 | 0.4 | 0.2 | 0.9 | 4 |
| | 0.9 | 0.9 | 0.2 | 0.7 | 6 |
| | 0.8 | 0.6 | 0.2 | 0.9 | 7 |
| | 0.9 | 0.7 | 0.2 | 0.2 | > 20 |
| | -0.7 | -0.8 | 0.4 | 0.8 | > 20 |

d.

| Operation | Weights | | Learning Rate | Bias | Iterations Required |
|---|---|---|---|---|---|
| | $W_1$ | $W_2$ | | | |
| NOR | 0.2 | 0.4 | 0.2 | 0.8 | 4 |
| | 0.8 | 0.9 | 0.2 | 0.8 | 5 |
| | 0.8 | 0.9 | 0.2 | 0.9 | 5 |
| | 0.8 | 0.9 | 0.2 | 0.6 | 4 |
| | 0.3 | 0.4 | 0.4 | 0.6 | 3 |

2. For NAND and NOR implementation, what is the effect of setting bias to value <0.5?

Iteration is going Infinitely

3. Add a positive bias <0.3 to AND and OR gate's output and check how many iterations are required to get correct output.

bias = 0.1 , AND = 3, OR = 4
bias = 0.2 , AND = 2, OR = 4

4. What are your observations, regarding the following?
   a. Lower value of learning rate is faster. (Yes/No)

Reason: _____

   b. For OR gate implementation, smaller values of weights require less iterations for obtaining correct result. (Yes/No)

Reason: _____

   c. For NAND gate implementation, larger values of weights require less iterations for obtaining correct result. (Yes/No)

Reason: _____

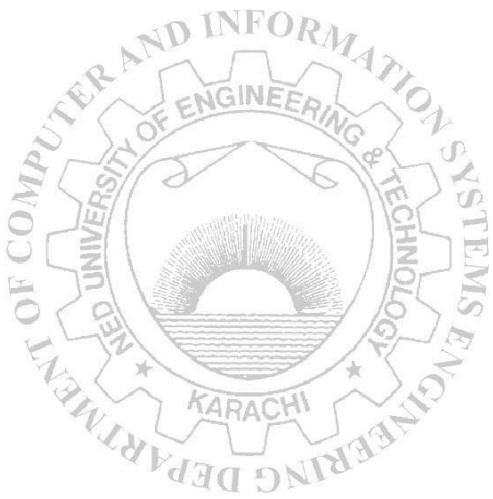
5. Implement XOR and XNOR functions and give all the formula you used in the implementation. Draw the MLPs used for the implementation of above functions. Also mention the following:

6.

| | ***XOR*** | ***XNOR*** |
|---|---|---|
| Learning rate | | |
| Initial weights | | |
| Bias | | |
| Iterations required | ` | |

7. Implement 3-input AND, OR, NAND and NOR gates.

Attach Excel Sheets here.



# Lab Session 02

## *Developing an Artificial Neural Network (ANN) Using Perceptron*

## EasyNN

EasyNN can be used to create, control, train, validate and query neural networks.

## Getting Started

In order to create a neural network, press the **`New`** toolbar button or use the **`File>New`** menu command to produce a new neural network,
An empty **`Grid`** with a vertical line, a horizontal line and an underline marker will appear. The marker shows the position where a grid column and row will be produced. Press the **`enter`** key and you will be asked *"Create new Example row?"* - answer **`Yes`**.
You will then be asked *"Create new Input/Output column?"* - answer **`Yes`**.
You have now created a training example with one input. The example has no name and no value. Press the **`enter`** key again and you will open the **`Edit`** dialog (see figure 2.1). This dialog is used to enter or edit all of the information in the Grid.

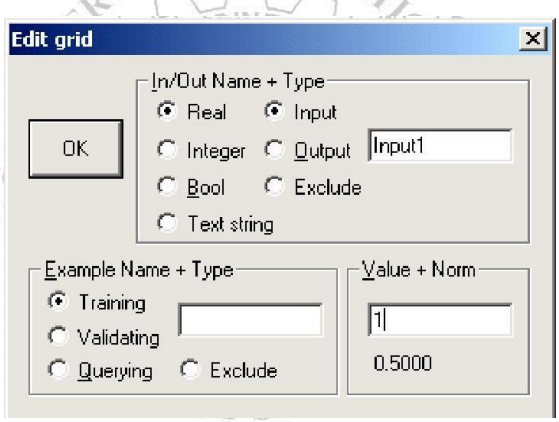



**Figure 2.1** *Edit Dialog*

Enter value, in the **Value + Norm** edit box and then tab.
Type input name in the **Example Name + Type** edit box and then tab. The **Type** is already set to Training so just **tab** again.
By following above procedure, you can enter training data.
To create the neural network press the  toolbar button or use the **Action>New Network** menu command. This will open the _New Network‘dialog. (See figure 2.2)

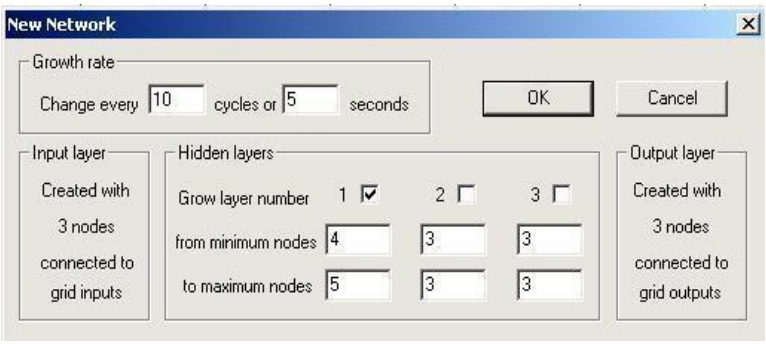



**Figure 2.2** *New Network Dialog*

The neural network will be produced from the data you entered into the grid.

Press the toolbar button or use the **`View>Network`** menu command to see the new neural network. This network will be somewhat like what is shown in figure 2.3

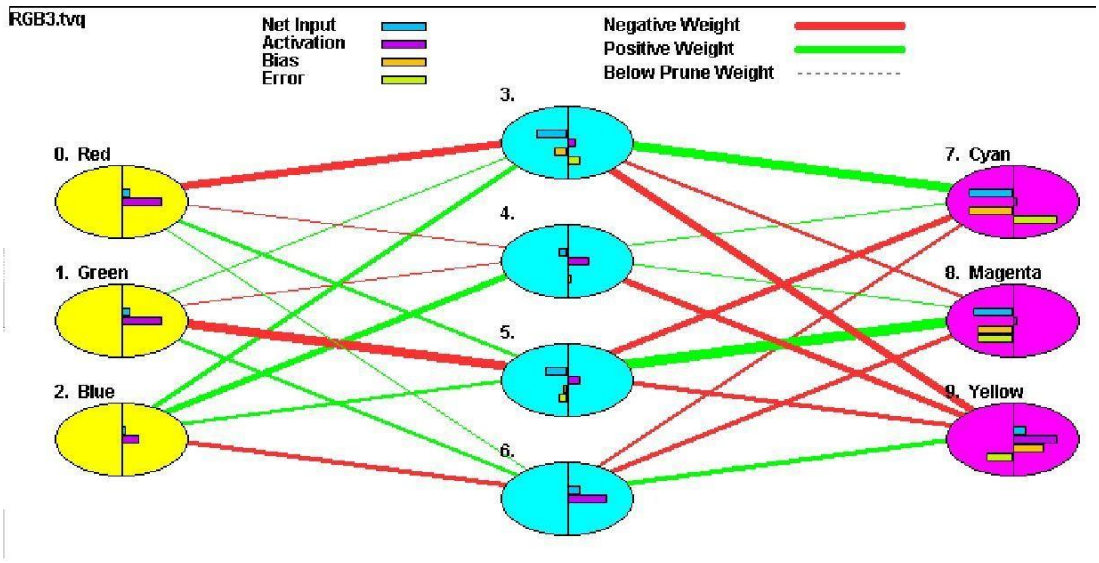



**Figure 2.3** *Artificial Neural Network*

The neural network controls now need to be set. Press toolbar button or sue **`Action>Change Controls`** menu command to open the **`Controls`** dialog. Check **`Optimize`** for both **`Learning Rate`** and **`Momentum`** and then press **`Ok`**. In this way, controls can be set and the neural network willbe ready to learn the data that you entered into the grid. (See figure 2.4)

Press toolbar button or use the **`Action>Start Learning`** menu command to open the **`Learning Progress`** dialog. The learning process will start and it will stop automatically, when the target error is reached. Press the **`Close`** button.

Press toolbar button or use the **`View>Graph`** menu command to see how the error reduced to the target.

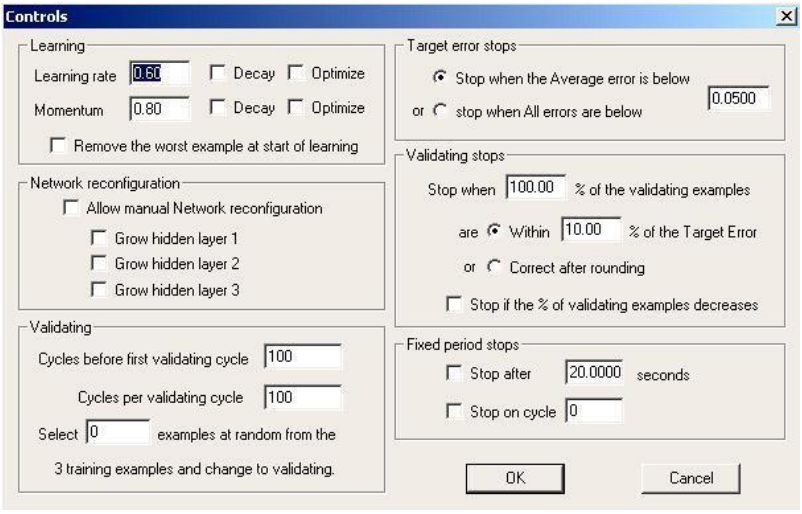



**Figure 2.4** *Controls Dialog*

Press  toolbar button or use the **Action>Query** menu command to open the **Query** dialog. Press the **Add Query** button and the example named ―Query‖ will be generated and selected. Values for Query can be inserted here and output can be observed (See figure 2.5)

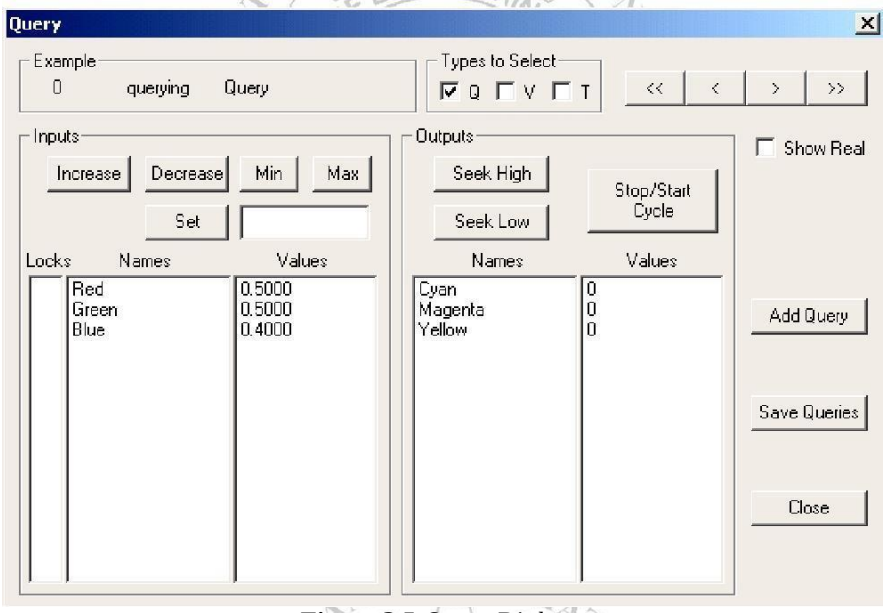



**Figure 2.5** *Query Dialog*

## Importing a File

Up till now, we have learned that how an artificial neural network can be grown and trained using EasyNN. Now we‘ll cover how real world data file, collected through different sources can be imported in EasyNN for neural network training.

## Procedure

EasyNN can import a text file to create a new Grid or to add new example rows to an existing Grid in the following manner.

1. **File>New** to create a new Grid or **File>Open** to add rows to an existing Grid.
2. **File > Import** File
3. Open the file that is to be imported.

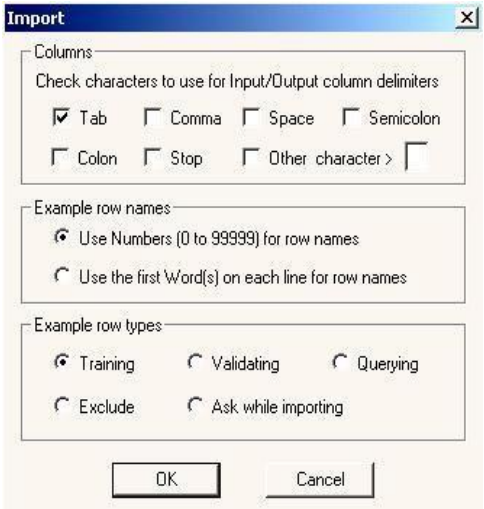



**Figure 2.6** *Import Window*

4. Check all the characters that are to be used for column delimiters.
5. Any words before the first delimiter on each line can be used for row names. If no row names are available then EasyNN can generate numbers for row names.
6. Press OK.

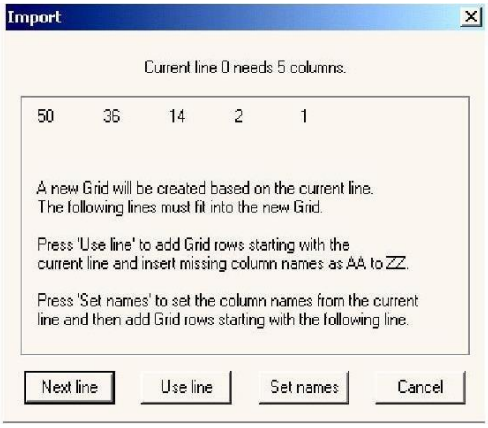



**Figure 2.7** *Selection pane*

7. Press **`Next line`** until the first line to be imported is shown.
8. Press **`Use line`** or **`Set names`** according to the instructions in the dialog.

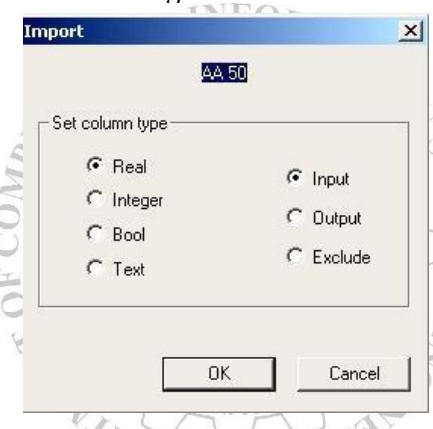



**Figure 2.8** *Options Window*

9. Set the column types when the first line is imported.
10. Press OK.
11. The rest of the file will be imported (See figure 2.9). Warnings will be produced for any lines in the file that are not suitable for the Grid.

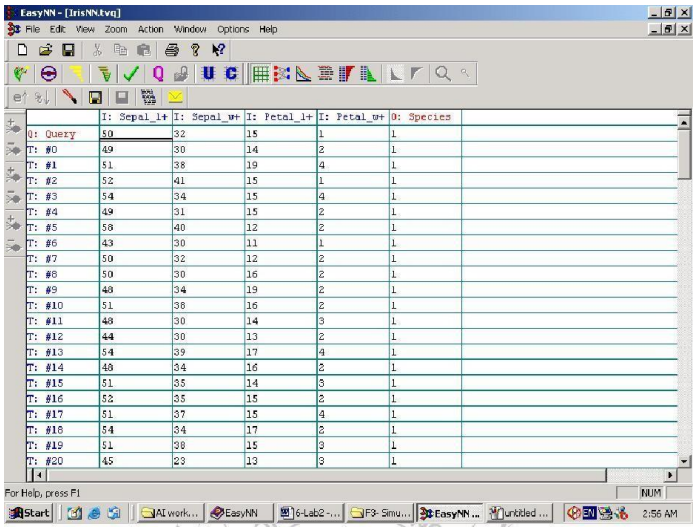



**Figure 2.9** *Grid window*

**Note:** *Network is grown and trained in the same manner as previously discussed.*

## Exercises

1. Create a neural network, by using the training data, presented in the table. And test it on the given queries.

| | Red | Green | Blue | Cyan | Magenta | Yellow | Output | Output | Type |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | 0 | 0 | 0 | 1 | Y | 0 | Training |
| | 1 | 0 | 1 | 0 | 1 | 0 | M | 0.5 | Training |
| | 0 | 1 | 1 | 1 | 0 | 0 | C | 1 | Training |
| | 0.9 | 0.9 | 0.3 | | | | | | Querying |
| | 0.6 | 0.6 | 0.4 | | | | | | Querying |
| | 0.5 | 0.5 | 0.5 | | | | | | Querying |
| | 0 | 0 | 0 | | | | | | Querying |
| | 1 | 1 | 1 | | | | | | Querying |
| | 0.8 | 0.2 | 0.7 | | | | | | Querying |

Learning rate selected: _____
Stop when average error is: _____

(a) Multiple output lines give better results. (Yes/ No)

Reason:

_____

_____

_____

2. Import a data file, provided by the instructor, grow an Artificial Neural Network, train it and test on different queries.

3. Give the following specification of your network.

(a) Learning rate selected : _____
(b) Stop when average error is: _____
(c) Inputs to the system and their type:

_____

_____

_____

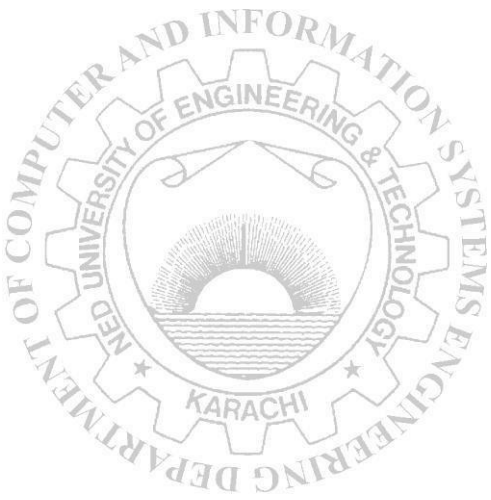
_____

_____

_____

_____

(d) Number of outputs and their types



# Lab Session 03

## *Applying Data Preprocessing for ANN*

There are a variety of parameters that play role in the success of any neural network solution. The External Parameters include the data. The quality, availability, reliability and relevance of the data used to develop and run the system are critical to its success. Even a primitive model can perform well if the input data has been processed in such a way that it clearly reveals the important information. On the other hand, even the best model cannot help us much if the necessary input information is presented in a complex and confusing way. Similarly, the internal parameters play a major role in the performance of ANN. These include the learning rate, momentum, number and size of layers etc.

## Data-flow in a typical ANN system

The Data Flow sequence in a typical ANN-based system is shown in Figure 3.1.

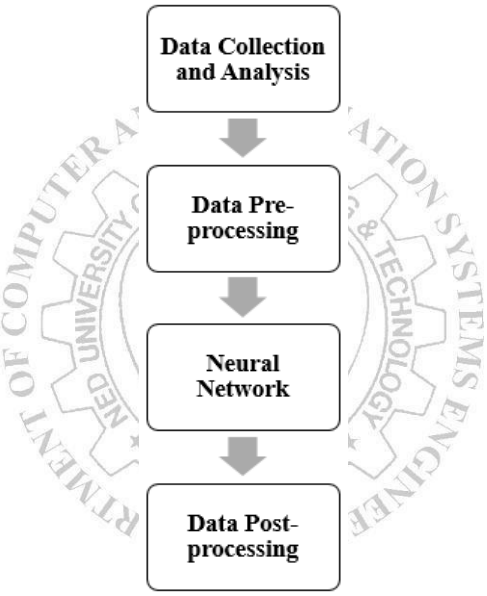



**Figure 3.1** *Data Flow in an ANN-based system*

The following steps are to be followed before data is actually presented to the Artificial Neural Network:

## I – Data Collection

The data collection plan typically consists of three tasks:

### 1 Identifying the data requirements

The first thing to do when planning data collection is to decide what data we will need to solve the problem. In general, it will be necessary to obtain the assistance of some experts in the field.

### 2 Identifying data sources

The next step is to decide from where the data will be obtained. This will allow us to make realistic estimates of the difficulty and expense of obtaining it. If the application demands real time data, these estimates should include an allowance for converting analogue data to digital form.

### 3 Determining the data quantity

It is important to make a reasonable estimation of how much data we will need to develop the neural

network properly. If too little data is collected, it may not reflect the full range of properties that the network should be learning, and this will limit its performance with unseen data. On the other hand, it is possible to introduce unnecessary expense by collecting too much data. In general, the quantity of data required is governed by the number of training cases that will be needed to ensure the network performs adequately.

# II – Data Preparation

When the raw data has been collected, it may need converting into a more suitable format. At this stage, we should do the following:

## 1 Data validity checks

Data validity checks will reveal any patently unacceptable data that, if retained, would produce poor results. A simple data range check is an example of validity checking. For example, if we have collected oven temperature data in degrees centigrade, we would expect values in the range 50 $^{o}$C to 400 $^{o}$C. A value of, say, -10 $^{o}$C, or 900 $^{o}$C, is clearly wrong.

## 2 Partitioning data

Partitioning is the process of dividing the data into validation sets, training sets, and test sets. By definition, **validation sets** are used to decide the architecture of the network; **training sets** are used to actually update the weights in a network; **test sets** are used to examine the final performance of the network. The primary concerns should be to ensure that: a) the training set contains enough data, and suitable data distribution to adequately demonstrate the properties we wish the network to learn; b) there is no unwarranted similarity between data in different data sets.

# III – Data Preprocessing

Data preprocessing consists of all the actions taken before the Neural Network comes into play. It is essentially a transformation T that transforms the raw real world data vectors X to a set of new data vectors Y:

$$Y = T(X)$$

such that:

i. Y preserves the ―valuable information‖ in X and
ii. Y is more useful than X.

This preprocessing of data is termed as "Normalization" and is most widely used technique for data preprocessing in machine learning. Normalization is necessary as it ensures uniformity of the numerical magnitudes of features. This uniformity avoids the dominance of features that have larger values compared to the other features or variables.

### 1. NORMALIZATION

Normalization is a specific form of feature scaling that transforms the range of features to a standard scale. Normalization or any other data scaling technique is required only when the dataset has features of varying ranges. Normalization encompasses diverse techniques tailored to different data distributions and model requirements. This lab session focuses on practicing normalization techniques on numerical data.

### For Numerical Data:

Fundamental techniques used in normalization are Min-Max scaling and Z-score normalization (standardization).

- **Min-Max Scaling**
  Typically termed as 'Normalization', this technique transforms the data features to a specified range, typically between 0 and 1. The formula for min-max scaling is:

$$X_{normalized} = X - X_{min} / X_{max} - X_{min}$$

Where,
X is a random feature value to be normalized,
$X_{min}$ is the minimum feature value in the dataset,
$X_{max}$ is the maximum feature value.

Note that this technique should be used when:
- There are few or no outliers in the data
- Upper and lower bounds are known approximately
- Data is uniformly distributed over a range
- Maintaining original shape of data is important

- **Z-Score Normalization (Standardization)**
  Standardization assumes a Gaussian distribution of data and transforms features to have a mean (μ) of 0 and a standard deviation (σ) of 1. The formula for standardization is:

$$X_{standardized} = X - \mu / \sigma$$

Note that this technique should be used when:
- The algorithm further applied expects normally distributed data
- Data is not restricted to a range
- Data features a number of standard deviations that lie away from the mean

## For Nominal Data:

Nominal data refers to unranked or unordered data, typically qualitative and used for categorization purposes. Examples include attributes like hair color (e.g., blonde, brunette, black). When normalizing nominal data, it's important to use techniques that don't rely on the numeric values of the data, such as one-hot encoding. One hot encoding basically creates a binary vector for each category. The difference between just assigning a label to a category or entity and one hot encoding can be observed from the Figure 3.1 attached below.

Label Encoding

| Food Name | Categorical # | Calories |
|---|---|---|
| Apple | 1 | 95 |
| Chicken | 2 | 231 |
| Broccoli | 3 | 50 |

→

One Hot Encoding

| Apple | Chicken | Broccoli | Calories |
|---|---|---|---|
| 1 | 0 | 0 | 95 |
| 0 | 1 | 0 | 231 |
| 0 | 0 | 1 | 50 |

**Figure 3.2** *Difference between Label and One Hot Encoding*

## For Ordinal Data:

A dataset that is ordered or ranked or used for qualitative purposes is termed as Ordinal data. When normalizing ordinal data, it's important to use techniques that preserve the order of the data. Techniques such as binning, ordinal encoding and scaling can be used to normalize ordinal data. For example, if the data has five ratings: poor, fair, good, very good, and excellent then the techniques will be applied as:

- **Ordinal Encoding:**
  Ordinal encoding assigns a numerical value to each ordinal level based on its order or rank. Then, normalized_data = {"poor": 1, "fair": 2, "good": 3, "very good": 4, "excellent": 5}

- **Binning:**
  Binning groups the ordinal data into a smaller number of bins or categories. Then, normalized_data = {"poor": "low", "fair": "low", "good": "medium", "very good": "high", "excellent": "high"}

- **Scaling:**
  Rescales the ordinal data to a specific range. Then, normalized_data = (data - 1) / (5 - 1)

# IV – Training the ANN

This involves the following steps:
1. Transfer Data from Spreadsheet (E.g. MS - Excel) to ANN Tool (E.g. EasyNN)
2. Grow a Neural Network
3. Set Controls (optional)
4. Start Training
5. View Results

# V – Querying the ANN

Querying the Network involves the following steps:
1. Transfer data from Spreadsheet to Tool
2. Query the Network
3. View Results

## Practical ANN-based Systems

Artificial Neural Networks are being used in various application domains, like prediction, classification, diagnosis and forecasting etc. in different situations like Stock Rate prediction, weather forecasting, Pattern Recognition and Medical Applications etc.

## An ANN based Heart Disease Diagnosis System

The case study has been taken up to diagnose the presence or absence of Heart Disease in clients. As mentioned above, there are certain steps that are to be considered before ANN comes into play. As mentioned above, these steps include:

**Data Collection –** A database of 75 attributes has been constructed for HDDS. After thorough analysis, irrelevant and redundant parameters have been removed. The final database comprises of 13 parameters, on the basis of which, the output is computed. These are:

1. Age (Real Number)
2. Gender (Binary)
3. Chest pain type (4 values) (Real Number)
4. Resting blood pressure (Real Number)
5. Serum cholesterol in mg/dl (Real Number)
6. Fasting blood sugar > 120 mg/dl (Binary)
7. Resting electrocardiographic results (values 0,1,2) (Real Number)
8. Maximum heart rate achieved (Real Number)
9. Exercise induced angina (Binary)
10. Old peak = ST depression induced by exercise relative to rest (Real Number)
11. The slope of the peak exercise ST segment (Real Number)
12. Number of major vessels (0-3) colored by fluoroscopy (Ordered)
13. Thal: 3 = normal; 6 = fixed defect; 7 = reversible defect (Real Number)

The output of the system is either absence or presence of Heart Disease (represented by 1 and 2 respectively). A total of 270 cases have been considered. Out of these cases, 55.56% showed an absence of disease and 44.44% showed presence of disease. It has been ensured that the dataset contains nearly equal percentage of both the classes of output.

A portion of the dataset is given here:

| Inputs | | | | | | | | | | | | | Output |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | Heart Disease |
| 44 | 0 | 3 | 108 | 141 | 0 | 0 | 175 | 0 | 0.6 | 2 | 0 | 3 | 1 |
| 71 | 0 | 4 | 112 | 149 | 0 | 0 | 125 | 0 | 1.6 | 2 | 0 | 3 | 1 |
| 45 | 0 | 2 | 112 | 160 | 0 | 0 | 138 | 0 | 0 | 2 | 0 | 3 | 1 |
| 57 | 1 | 3 | 150 | 168 | 0 | 0 | 174 | 0 | 1.6 | 1 | 0 | 3 | 1 |
| 65 | 1 | 4 | 120 | 177 | 0 | 0 | 140 | 0 | 0.4 | 1 | 0 | 7 | 1 |
| 46 | 0 | 3 | 142 | 177 | 0 | 2 | 160 | 1 | 1.4 | 3 | 0 | 3 | 1 |
| 77 | 1 | 4 | 125 | 304 | 0 | 2 | 162 | 1 | 0 | 1 | 3 | 3 | 2 |
| 56 | 0 | 4 | 200 | 288 | 1 | 2 | 133 | 1 | 4 | 3 | 2 | 7 | 2 |
| 67 | 1 | 4 | 120 | 237 | 0 | 0 | 71 | 0 | 1 | 2 | 0 | 3 | 2 |
| 54 | 1 | 2 | 192 | 283 | 0 | 2 | 195 | 0 | 0 | 1 | 1 | 7 | 2 |
| 62 | 0 | 4 | 160 | 164 | 0 | 2 | 145 | 0 | 6.2 | 3 | 3 | 7 | 2 |
| 67 | 1 | 4 | 160 | 286 | 0 | 2 | 108 | 1 | 1.5 | 2 | 3 | 3 | 2 |

There are some companies that collect and provide such statistical data, which may be used in Neural network based applications.

## Data Preparation

This step involves Validity checking and Partitioning of Data. The data has been partitioned into two parts – The training dataset and the test dataset. It has been ensured that the minimum and maximum value of each parameter lies in the training dataset because the ANN generates a working range for each parameter and values outside this range are clipped off.

## Data Preprocessing

The data has been preprocessed using the scaling technique, and then used to train the ANN. The samplescaled data is shown here:

| Input | | | | | | | | | | | | | Output |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | Heart Disease |
| 0.3125 | 0 | 0.666667 | 0.13207 | 0.03424 | 0 | 0 | 0.793893 | 0 | 0.09677 | 0.5 | 0 | 0 | 1 |
| 0.875 | 0 | 1 | 0.16981 | 0.05251 | 0 | 0 | 0.412213 | 0 | 0.25806 | 0.5 | 0 | 0 | 1 |
| 0.33333 | 0 | 0.333333 | 0.16981 | 0.07762 | 0 | 0 | 0.511450 | 0 | 0 | 0.5 | 0 | 0 | 1 |
| 0.58333 | 1 | 0.666667 | 0.52830 | 0.09589 | 0 | 0 | 0.786259 | 0 | 0.25806 | 0 | 0 | 0 | 1 |
| 0.75 | 1 | 1 | 0.24528 | 0.11643 | 0 | 0 | 0.526717 | 0 | 0.06451 | 0 | 0 | 1 | 1 |
| 0.35416 | 0 | 0.666667 | 0.45283 | 0.11643 | 0 | 1 | 0.679389 | 1 | 0.22580 | 1 | 0 | 0 | 1 |
| 0.5625 | 0 | 1 | 0.37735 | 0.64611 | 0 | 1 | 0.603053 | 1 | 0.30645 | 0.5 | 0.6667 | 1 | 2 |
| 1 | 1 | 1 | 0.29245 | 0.40639 | 0 | 1 | 0.694656 | 1 | 0 | 0 | 1 | 0 | 2 |
| 0.5625 | 0 | 1 | 1 | 0.36986 | 1 | 1 | 0.473282 | 1 | 0.64516 | 1 | 0.6667 | 1 | 2 |
| 0.79166 | 1 | 1 | 0.24528 | 0.25342 | 0 | 0 | 0 | 0 | 0.16129 | 0.5 | 0 | 0 | 2 |
| 0.52083 | 1 | 0.333333 | 0.92452 | 0.35844 | 0 | 1 | 0.946564 | 0 | 0 | 0 | 0.3333 | 1 | 2 |
| 0.6875 | 0 | 1 | 0.62264 | 0.08675 | 0 | 1 | 0.564885 | 0 | 1 | 1 | 1 | 1 | 2 |

## Training the ANN

1. Training involves importing the _train dataset‘ for HDDS in the tool.
2. The Neural Network is then generated and various parameters are defined, along with the terminating criterion.
3. The training begins once the controls are set and is terminated once the criterion is met.

## Testing the ANN

Once training is complete, the HDDS is tested for authenticity. This may be accomplished by either inserting the queries manually or by importing the _HDDS test dataset' into the tool.

On the basis of training, the ANN based HDDS predicts the presence or absence of Heart Disease in patients.

## Implementing Normalization with Python

Normalization can also be implemented using Python. Scikit-learn library can be used for this purpose. Scikit-learn is a versatile Python library that is engineered to simplify the complexities of machine learning. Before trying the code below install all the necessary packages. The code features sklearn which has been deprecated, so Scikit-learn library needs to be installed.

For this lab session we'll be using "Iris dataset" which was introduced in Lab 2. There are four features measured for each flower, these are the sepal length, sepal width, petal length, and petal width - all in centimeters. There are 150 instances (samples) in the dataset, with 50 samples from each of the three species. The Iris dataset is usually used for classification tasks, where the goal is to predict the correct species among the three classes. However, we will use this dataset to show the transformation of the data when applying normalization (min-max scaling).

The first step is to load the dataset as shown in Figure 3.2. After loading the dataset, it is divided 80-20 in training and testing sets as shown in Figure 3.3. To normalize the data, we will use the MinMaxScaler functionality as shown in Figure 3.4 from sklearn library and apply it to our dataset; we have already imported the required libraries earlier. After transforming the data, the statistical description can be viewed to observe the transformation by using the snippet in Figure 3.5.

```
import numpy as np
import pandas as pd
# To import the dataset
from sklearn.datasets import load_iris
# To be used for splitting the dataset into training and test sets
from sklearn.model_selection import train_test_split
# To be used for min-max normalization
from sklearn.preprocessing import MinMaxScaler
# To be used for Z-normalization (standardization)
from sklearn.preprocessing import StandardScaler

# Load the iris dataset from Scikit-learn package
iris = load_iris()

# This prints a summary of the characteristics, statistics of the dataset
print(iris.DESCR)

# Divide the data into features (X) and target (Y)
# Data is converted to a panda's dataframe
X = pd.DataFrame(iris.data)

# Separate the target attribute from rest of the data columns
Y = iris.target

# Take a look at the dataframe
X.head()

# This prints the shape of the dataframe (150 rows and 4 columns)
X.shape()
```

**Figure 3.3** *Initial steps before Normalization*

```
# To divide the dataset into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, Y ,test_size=0.2)
```

**Figure 3.4** *Splitting dataset*

```
# Good practice to keep original dataframes untouched for reusability
X_train_n = X_train.copy()
X_test_n = X_test.copy()

# Fit min-max scaler on training data
norm = MinMaxScaler().fit(X_train_n)

# Transform the training data
X_train_norm = norm.transform(X_train_n)

# Use the same scaler to transform the testing set
X_test_norm = norm.transform(X_test_n)
```

**Figure 3.5** *Normalizing the dataset*

```
X_train_norm_df = pd.DataFrame(X_train_norm)

        # Assigning original feature names for ease of read
X_train_norm_df.columns = iris.feature_names

X_train_norm_df.describe()
```

**Figure 3.6** *Statistical description after transformation*

## EXERCISES

1. Explore Kaggle or any other opensource to find some interesting structured data. Note that the dataset should have a mix of all the categories discussed above. A separate dataset containing categorical data can also be used for this exercise. Describe following for your dataset.

| | |
|---|---|
| Number of Input Parameters: | 5 |
| Number of Output Categories: | 3 |
| Inputs that need normalization: | |

2. Attach a portion of the chosen dataset here along with features names.

3. Identify appropriate data preprocessing techniques that can be applied to different input features. Mention name of few features that are preprocessed using following techniques.

Normalization: Age, Stay

Standardization:

One hot encoding:

Ordinal Encoding

Binning

Scaling:

4. Attach a portion of dataset after preprocessing.

5. Generate and train an ANN for above dataset and then query the network. Attach your code here.

6. In continuation with Exercise 5 fill in the following entries:

| | |
|---|---|
| Number of Output Categories: | |
| Number of Data Rows: | |
| Number of Training Rows: | |
| Number of Testing Rows: | |
| Learning Rate: | |
| Momentum: | |
| Number of Layers: | |
| Size of Layers: | |
| Number of Cycles for Training: | |
| Percentage of Correctness in Results: | |

7. Develop a GitHub repository for your ANN and share its URL here.

# Lab Session 04

## *Developing ANN Using ADALINE*

This lab session will introduce the concept of Algorithmic design of Artificial Neural Network (ANN). The algorithmic design of ANNs is necessary to understand the design philosophy of ANN implementations on software programming languages.

MATLAB, an abbreviation for 'matrix laboratory,' is a platform for solving mathematical and scientific problems. It is a proprietary programming language developed by MathWorks, allowing matrix manipulations, functions and data plotting, algorithm implementation, user interface creation and interfacing with programs written in programming languages like C, C++, Java and so on.

Widrow and Hoff [1960] development developed the learning rule that is very closely related to the perceptron learning rule. The rule, called Delta rule, adjusts the weights to reduce the difference between the net input to the output unit, and the desired output, which results in the least mean squared (LMS error). Adaline (Adaptive Linear Neuron) and Madaline (Multilayered Adaline) networks use this LMS learning rule and are applied to various neural network applications.

Adaline is found to use bipolar activations for its input signals and target output. The weights and the bias of the Adaline are adjustable. The learning rule used can be called as Delta rule, Least Mean Square rule or Widrow-Hoff rule. The derivation of this rule with single output unit, several output units. Since the activation function is an identity function, the activation of the unit is its net input.

When Adaline is to be used for pattern classification, then, after training, a threshold function is applied to the net input to obtain the activation. The Adaline unit can solve the problem with linear separability if it occurs.

## Architecture

The architecture of an adaline is shown in Fig. 4.1.

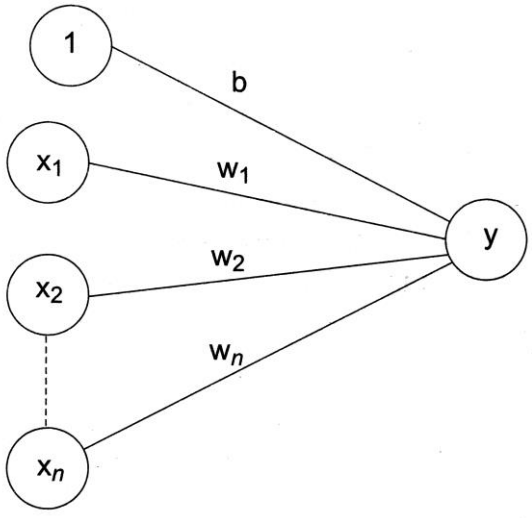



**Figure 4.1: Architecture of Adaline Network**

The Adaline has only one output unit. This output unit receives input from several units and also from bias; whose activation is always +1. The adaline also resembles a single layer network. It receives input

from several neurons. It should be noted that it also receives input from the unit which is always '+1' called as bias. The bias weights are also trained in the same manner as the other weights. In Fig. 1, an input layer with $x_1 \ldots x_i \ldots x_n$ and bias, an output layer with only one output neuron is present. The link between the input and output neurons possess weighted interconnections. These weights get changed as the training progresses.

# Algorithm

Basically, the initial weights of Adaline network have to be set to small random values and not to zero as discussed in Hebb or perceptron networks, because this may influence the error factor to be considered. After the initial weights are assumed, the activations for the input unit are set. The net input is calculated based on the training input patterns and the weights. By applying delta learning rule discussed in 3.3.3, the weight updation is being carried out. The training process is continued until the error, which is the difference between the target and the net input becomes minimum. The step based training algorithm for an adaline is as follows:

# Pseudo Code for Adaline network

## Training Algorithm

The following is the pseudo code for training an Adaline network.

**Step 1:** Initialize weights (not zero but small random values are used). Set learning rate α.

**Step 2:** While stopping condition is false, do Step 3-7.

**Step 3:** For each bipolar training pair s: t, perform Steps 4-6.

**Step 4:** Set activations of input units $x_i = s_i$ for $i$ 1 to n.

**Step 5:** Compute net input to output unit $y_{-in} = b + \Sigma x_i w_i$

**Step 6**: Update bias and weights, $i$ = 1 to n.

$$w_i(new) = w_i(old) + a(t - y_{-in})x_i$$

$$b(new) = b(old) + \alpha(t - y_{-in})$$

**Step 7:** Test for stopping condition.

The stopping condition may be when the weight change reaches small level or number of iterations etc.

## Testing Algorithm

The application procedure, which is used for testing the trained network is as follows. It is mainly based on the bipolar activation.

**Step 1:** Initialize weights obtained from the training algorithm.

**Step** 2: For each bipolar input vector x, perform Steps 3-5.

**Step 3:** Set activations of input unit.

**Step 4:** Calculate the net input to the output unit. $y_{-in} = b + \Sigma x_i w_i$

**Step 5:** Finally apply the activations to obtain the output y.

$$y = f(y - in) = \begin{cases} 1, & if\ y - in \geq 0 \\ -1, & if\ y - in < 0 \end{cases}$$

**Exercise**

Develop a MATLAB program for OR function with bipolar inputs and targets using Adaline network. The truth table for the OR function with bipolar inputs and targets is given as,

| X1 | X2 | Y |
|---|---|---|
| -1 | -1 | -1 |
| -1 | 1 | 1 |
| 1 | -1 | 1 |
| 1 | 1 | 1 |

# Lab Session 05

## *Developing ANN Using Backward Propagation*

## Designing Artificial Neural Network in Python

Python is a high-level programming language which is applicable in every computing domain. It was developed in 1991 by Guido van Rossum. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python offers significant code readability with easy to program language constructs.

Python offers extensive library support for designing Artificial Neural Network (ANN). In Python, ANN algorithm can be written in simple syntax or frameworks may be used for ANN implementation. Python offers a large number of frameworks for implementation of ANNs from a simple perceptron level implementation to deep neural networks such as Convolutional Neural Networks (CNNs) and Long-Short Term Memory Networks (LSTM).

In Python programming environment, numpy library is used for scientific and linear algebra operations. Supposing that Python and pip are already installed on your system, numpy can be installed by running the following command on the python command prompt.

```
pip install numpy
```

In this lab, a very simple network with 2 layers is selected for implementation. In order to do that a very simple dataset is required, so the XOR dataset in used for implementation. The Truth table for XOR gate along with network diagram is shown in Fig. 5.1 and 5.2, respectively.

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Figure 5.1** *Truth Table for XOR gate*

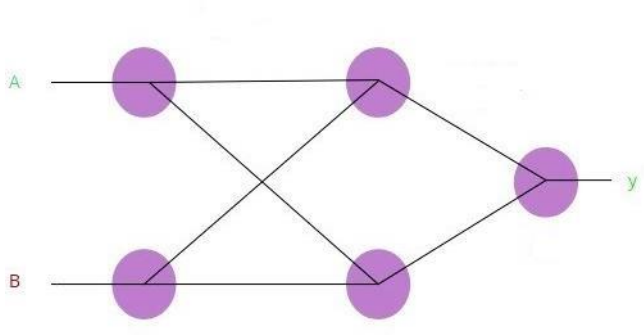



**Figure 5.2** *Network diagram for XOR gate*

## Implementation in Python

Here, the parameters to be learned are the weights W1, W2 and the biases b1, b2. From the basic theory of NNs we know that the activations A1 and A2 are calculated as following:

```
A1 = h(W1*X + b1)
A2 = g(W2*A1 + b2)
```

Where g and h are the two activation functions we chose(for us sigmoid and tanh) and W1, W1, b1, b2 are generally Matrices.

First we will implement our sigmoid activation function defined as follows: g(z) = 1/(1+e^(-z)) where z will be a matrix in general. Luckily numpy supports calculations with matrices so the code is relatively simple:

```
import numpy as np
     def sigmoid(z):
       return 1/(1 + np.exp(-z))
```

Next we have to initialize our parameters. Weight matrices W1 and W2 will be randomly initialized from a normal distribution while biases b1 and b2 will be initialized to zero. The function initialize_parameters(n_x, n_h, n_y) takes as input the number of units in each of the 3 layers and initializes the parameters properly:

```
def initialize_parameters(n_x, n_h, n_y):
  W1 = np.random.randn(n_h, n_x)
  b1 = np.zeros((n_h, 1))
  W2 = np.random.randn(n_y, n_h)
  b2 = np.zeros((n_y, 1))

parameters
        = {
          "W1": W1,
          "b1" : b1,
          "W2": W2,
          "b2" : b2
          }
          return parameters
```

The next step is to implement the Forward Propagation. The function forward_prop(X, parameters) takes as input the neural network input matrix X and the parameters dictionary and returns the output of the NN A2 with a cache dictionary that will be used later in back propagation.

```
def
forward_prop(X,
parameters):
            W1 = parameters["W1"]
            b1 = parameters["b1"]
```

```
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache = {
        "A1": A1,
        "A2": A2
    }
    return A2, cache
```

We now have to compute the loss function. We will use the Cross Entropy Loss function. Calculate_cost(A2, Y) takes as input the result of the NN A2 and the ground truth matrix Y and returns the cross entropy cost:

```
def calculate_cost(A2, Y):
  cost = -np.sum(np.multiply(Y, np.log(A2)) + np.multiply(1-Y,
np.log(1-A2)))/m
  cost = np.squeeze(cost)
  return cost
```

Now the most difficult part of the Neural Network algorithm, Back Propagation. The code here may seem a bit weird and difficult to understand but we will not dive into details of why it works here. This function will return the gradients of the Loss function with respect to the 4 parameters of our network (W1, W2, b1, b2):

```
def backward_prop(X, Y, cache, parameters):
  A1 = cache["A1"]
  A2 = cache["A2"]
  W2 = parameters["W2"]
  dZ2 = A2 - Y
  dW2 = np.dot(dZ2, A1.T)/m
  db2 = np.sum(dZ2, axis=1, keepdims=True)/m
  dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
  dW1 = np.dot(dZ1, X.T)/m
  db1 = np.sum(dZ1, axis=1, keepdims=True)/m
  grads = {
    "dW1": dW1,
    "db1": db1,
    "dW2": dW2,
    "db2": db2
  }
  return grads
```

We will use **Gradient Descent** algorithm to update our parameters and make our model learn with the learning rate passed as a parameter:

```
def update_parameters(parameters, grads, learning_rate):
  W1 = parameters["W1"]
  b1 = parameters["b1"]
  W2 = parameters["W2"]
  b2 = parameters["b2"]
  dW1 = grads["dW1"]
  db1 = grads["db1"]
  dW2 = grads["dW2"]
  db2 = grads["db2"]
  W1 = W1 - learning_rate*dW1
  b1 = b1 - learning_rate*db1
  W2 = W2 - learning_rate*dW2
  b2 = b2 - learning_rate*db2

  new_parameters = {
    "W1": W1,
    "W2": W2,
    "b1" : b1,
    "b2" : b2
  }
  return new_parameters
```

By now we have implemented all the functions needed for one circle of training. Now all we have to do is just put them all together inside a function called model() and call model() from the main program. Model() function takes as input the features matrix X, the labels matrix Y, the number of units n_x, n_h, n_y, the number of iterations we want our Gradient Descent algorithm to run and the learning rate of Gradient Descent and combines all the functions above to return the trained parameters of our model:

```
def model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate):
parameters = initialize_parameters(n_x, n_h, n_y)
  for i in range(0, num_of_iters+1):
    a2, cache = forward_prop(X, parameters)
    cost = calculate_cost(a2, Y)
    grads = backward_prop(X, Y, cache, parameters)
    parameters = update_parameters(parameters, grads,
learning_rate)
    if(i%100 == 0):
      print('Cost after iteration# {:d}: {:f}'.format(i, cost))
  return parameters
```

The **training part** is now over. The function above will return the trained parameters of our NN. Now we just have to make our **prediction**. The function predict(X, parameters) takes as input the matrix X with elements the 2 numbers for which we want to compute the XOR function and the trained parameters of the model and returns the desired result y_predict by using a threshold of 0.5:

```
def
predict(X,
parameters):
              a2, cache = forward_prop(X, parameters)
```

```
    yhat = a2
    yhat = np.squeeze(yhat)
    if(yhat >= 0.5):
      y_predict = 1
    else:
      y_predict = 0
    return y_predict
```

We are done with all the functions needed finally. Now let's go to the main program and declare our matrices X, Y and the hyperparameters n_x, n_h, n_y, num_of_iters, learning_rate:

```
np.random.seed(2)
    # The 4 training examples by columns
    X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
    # The outputs of the XOR for every example in X
    Y = np.array([[0, 1, 1, 0]])
    # No. of training examples
    m = X.shape[1]
    # Set the hyperparameters
    n_x = 2     #No. of neurons in first layer
    n_h = 2     #No. of neurons in hidden layer
    n_y = 1     #No. of neurons in output layer
    num_of_iters = 1000
    learning_rate = 0.3
```

Having all the above set up, training the model on them is as easy as calling this line of code:

```
    trained_parameters = model(X, Y, n_x, n_h, n_y, num_of_iters,
                               learning_rate)
```

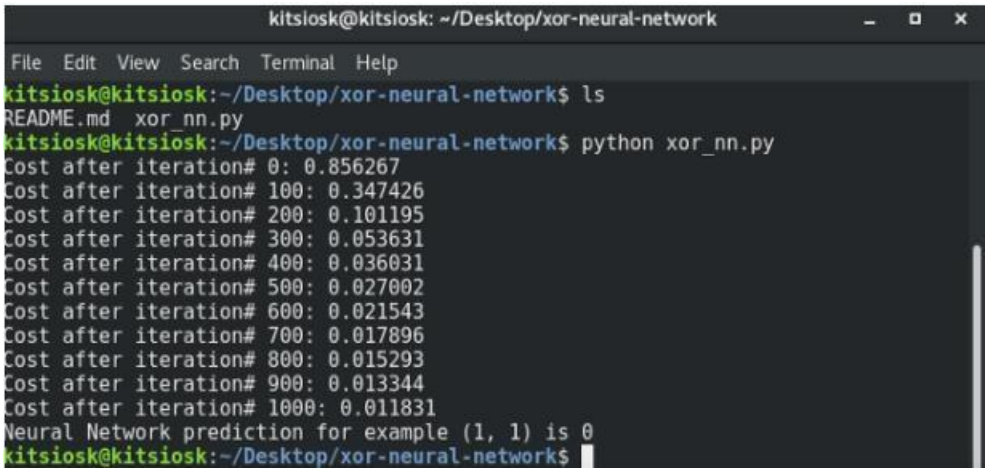
Finally let's make our prediction for a random pair of numbers, let's say (1,1):

```
# Test 2X1 vector to calculate the XOR of its elements.
# You can try any of those: (0, 0), (0, 1), (1, 0), (1, 1)
X_test = np.array([[1], [1]])
y_predict = predict(X_test, trained_parameters)
# Print the result
print('Neural Network prediction for example ({:d}, {:d}) is
{:d}'.format(
    X_test[0][0], X_test[1][0], y_predict))
```

That was the actual code! Let's see our results. If we run our file, let's say xor_nn.py, with this command

```
  python xor_nn.py
```

we get the following result, that is indeed correct because 1XOR1=0.




# Implementation of Logic Functions

## Complete Code

This code is written by Konstantinos Kitsios. Available at [https://gitlab.com/kitsiosk/xor-neural-net/blob/master/xor_nn.py](https://gitlab.com/kitsiosk/xor-neural-net/blob/master/xor_nn.py).

```
"""
    Simple Neural Network with 1 hidden layer with the number
    of hidden units as a hyperparameter to calculate the XOR
function
"""

import numpy as np

def sigmoid(z):
    return 1/(1 + np.exp(-z))

def initialize_parameters(n_x, n_h, n_y):
    W1 = np.random.randn(n_h, n_x)
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)
    b2 = np.zeros((n_y, 1))

    parameters = {
        "W1": W1,
        "b1" : b1,
        "W2": W2,
        "b2" : b2
    }
    return parameters

def forward_prop(X, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
```

```
    b2 = parameters["b2"]

    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    cache = {
        "A1": A1,
        "A2": A2
    }
    return A2, cache

def calculate_cost(A2, Y):
    cost = -np.sum(np.multiply(Y, np.log(A2)) + np.multiply(1-Y,
np.log(1-A2)))/m
    cost = np.squeeze(cost)

    return cost

def backward_prop(X, Y, cache, parameters):
    A1 = cache["A1"]
    A2 = cache["A2"]

    W2 = parameters["W2"]

    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T)/m
    db2 = np.sum(dZ2, axis=1, keepdims=True)/m
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T)/m
    db1 = np.sum(dZ1, axis=1, keepdims=True)/m

    grads = {
        "dW1": dW1,
        "db1": db1,
        "dW2": dW2,
        "db2": db2
    }

    return grads

def update_parameters(parameters, grads, learning_rate):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]

    W1 = W1 - learning_rate*dW1
    b1 = b1 - learning_rate*db1
    W2 = W2 - learning_rate*dW2
```

```
    b2 = b2 - learning_rate*db2

    new_parameters = {
        "W1": W1,
        "W2": W2,
        "b1" : b1,
        "b2" : b2
    }

    return new_parameters


def model(X, Y, n_x, n_h, n_y, num_of_iters, learning_rate):
    parameters = initialize_parameters(n_x, n_h, n_y)

    for i in range(0, num_of_iters+1):
        a2, cache = forward_prop(X, parameters)

        cost = calculate_cost(a2, Y)

        grads = backward_prop(X, Y, cache, parameters)

        parameters = update_parameters(parameters, grads,
learning_rate)

        if(i%100 == 0):
            print('Cost after iteration# {:d}: {:f}'.format(i,
cost))

    return parameters

def predict(X, parameters):
    a2, cache = forward_prop(X, parameters)
    yhat = a2
    yhat = np.squeeze(yhat)
    if(yhat >= 0.5):
        y_predict = 1
    else:
        y_predict = 0

    return y_predict

np.random.seed(2)

# The 4 training examples by columns
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])

# The outputs of the XOR for every example in X
Y = np.array([[0, 1, 1, 0]])

# No. of training examples
m = X.shape[1]

# Set the hyperparameters
n_x = 2     #No. of neurons in first layer
n_h = 2     #No. of neurons in hidden layer
```

```
n_y = 1     #No. of neurons in output layer
num_of_iters = 1000
learning_rate = 0.3

trained_parameters = model(X, Y, n_x, n_h, n_y, num_of_iters,
learning_rate)

# Test 2X1 vector to calculate the XOR of its elements.
# Try (0, 0), (0, 1), (1, 0), (1, 1)
X_test = np.array([[1], [1]])

y_predict = predict(X_test, trained_parameters)

print('Neural Network prediction for example ({:d}, {:d}) is
{:d}'.format(
    X_test[0][0], X_test[1][0], y_predict))
```

## Exercises

Use the above lab example for XNOR gate implementation and retrain the network. Write down the modifications (only) to the above mentioned Python code for XNOR implementation.

2. Develop a GitHub repository for your ANN and share its URL here.

_____

3. Attach your code here.

# Lab Session 6

## *Applying Uninformed Searching Techniques for Problem Solving*

## Creating A New Graph

To create a new graph in the applet, simply select 'Create New Graph' from the 'File' menu. By default, the applet opens with an empty canvas in ‘Create’ mode.

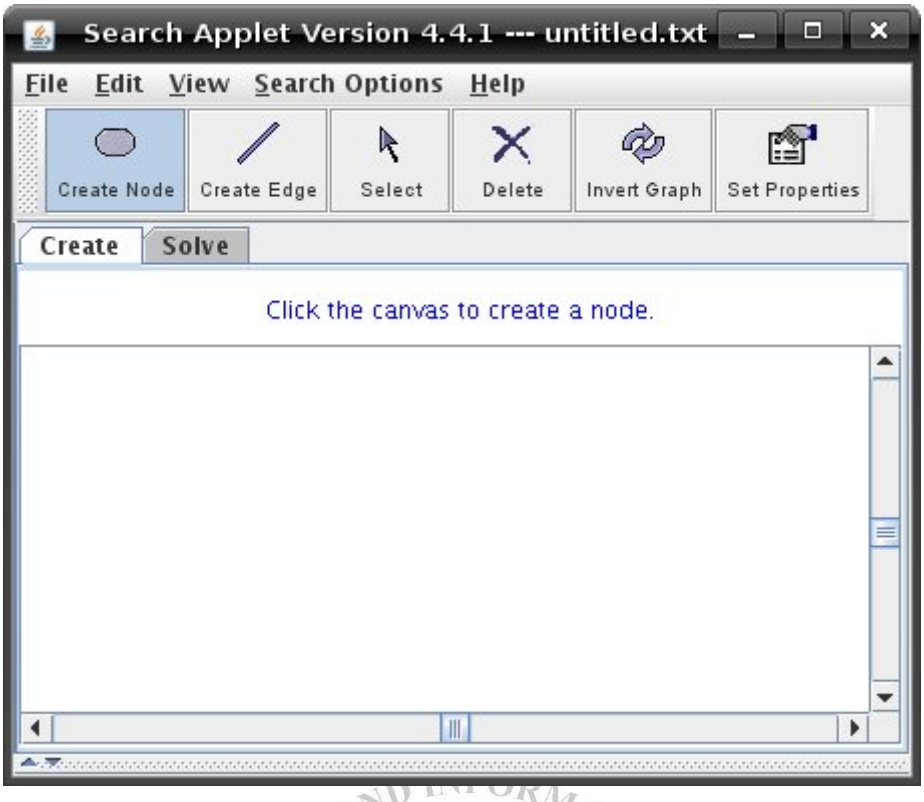



**Figure 6.1** *Search Applet blank version*

The 'Create Node' button is automatically active when applet starts, you can find this button on the tool bar at the top of the window. While this button is active, left clicking on the blank white canvas will create a new node, and similarly for each button that is depressed. Each time a button is clicked, the message panel above the canvas displays a message giving information of what can be done next.
Coming back to creating nodes, whenever a new node is created, ‘Node Properties’ dialogue box opens.

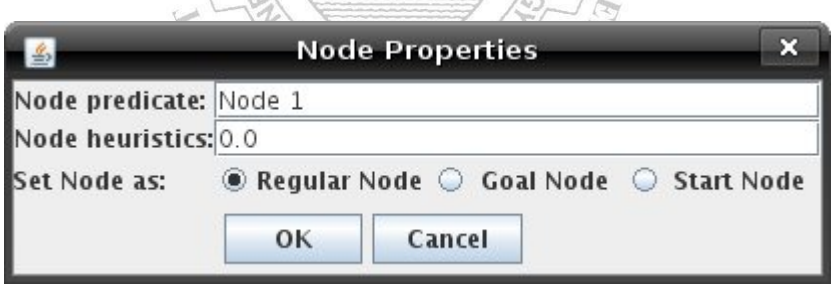



**Figure 6.2** *Node properties dialogue box*

A new name can be set for a node by typing it into the box beside the words 'Node Predicate'. Node heuristics as well as type of node can be set (regular, goal, or start node). Click 'OK' to create the node or 'Cancel' to

cancel the node creation.

After creating some more nodes for the graph, next stage is to link them together by using edges. Click on the 'Create Edge' button. Now click on a node from which this edge will start. After this click on the node on which this edge will end. An 'Edge Properties' dialog box will open where the edge's cost can be set:


**Figure 6.3** *Edge properties dialogue box*

Click 'OK' to create the node or 'Cancel' to cancel the edge creation. After creating an edge, an arrow can be observed between the two nodes, pointing from the first node to the second node.
Edge costs and node heuristics can also be set automatically by going to the 'Search Options' menu and clicking on 'Set Node Heuristics Automatically' for node heuristics and 'Set Edge Costs Automatically' for edge costs.
To enable viewing the node heuristics or edge costs on the graph, click on "Show Node Heuristics" or "Show Edge Costs" check boxes under the view menu.

Figure below shows a sample graph generated after creating different nodes and joining them by using edges:

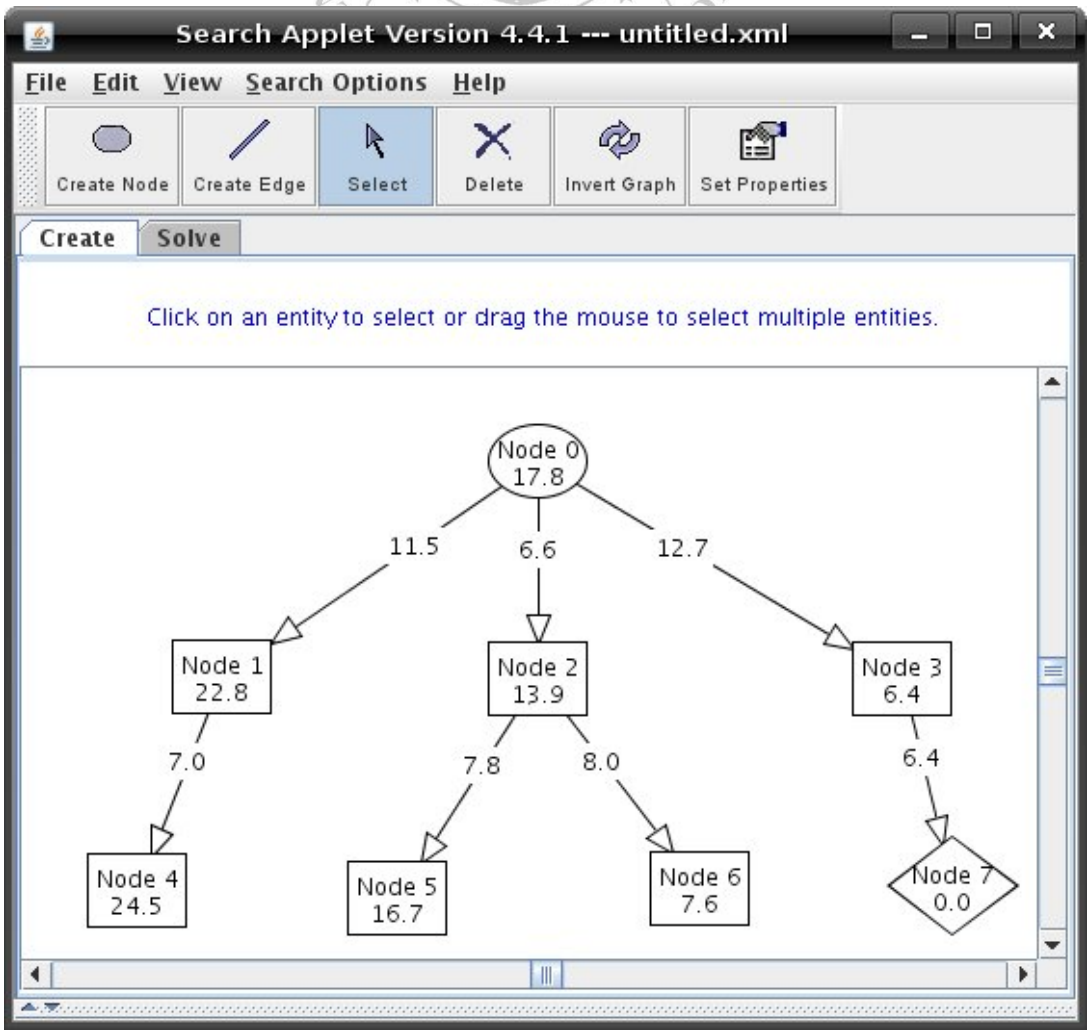



**Figure 6.4** *Graph creation*

## Loading A Pre-existing Graph

AISpace applet contains pre-existing graphs as well which can be used to understand solving graphs. To open a pre-existing graph, go to 'File', then 'Load Sample Problem' which opens a dialogue box containing a list of graphs.

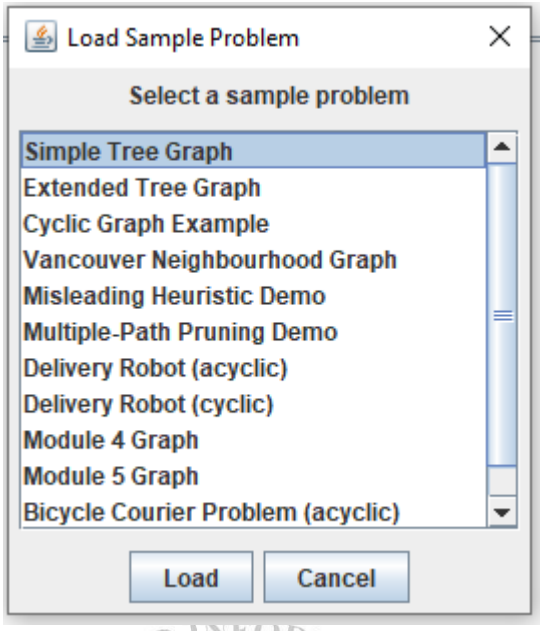



**Figure 6.5** *Load sample problem dialogue box with list of examples*

## Solving a Graph

Next step after learning to create a new graph is learning how to solve one. For doing this 'Simple Tree Graph' example can be loaded from the list of sample problems as discussed above. Once the example is loaded, change the mode from 'Create' to 'Solve' as shown in Fig. 6.6

When in 'Solve' mode, a new panel at the bottom of the screen shows algorithm selected for solving the graph as well as area containing the current path of the graph. This can also be observed in Fig. 6.6. Modifications can be introduced in this graph by going back to 'Create' mode.

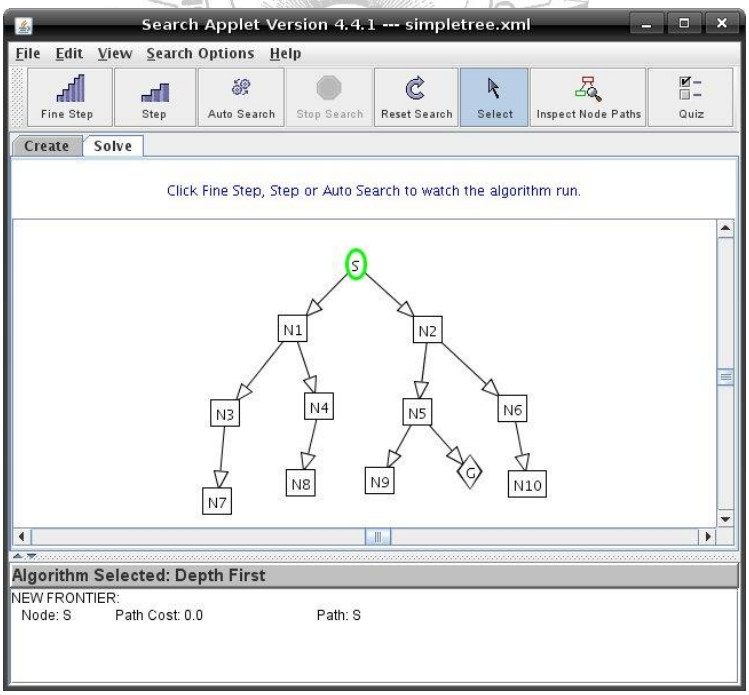



**Figure 6.6** *Graph in 'Solve' mode*

By solving or searching a graph, we aim to find the path along the edges from a start node to a goal node.

There are four ways to run a search algorithm.

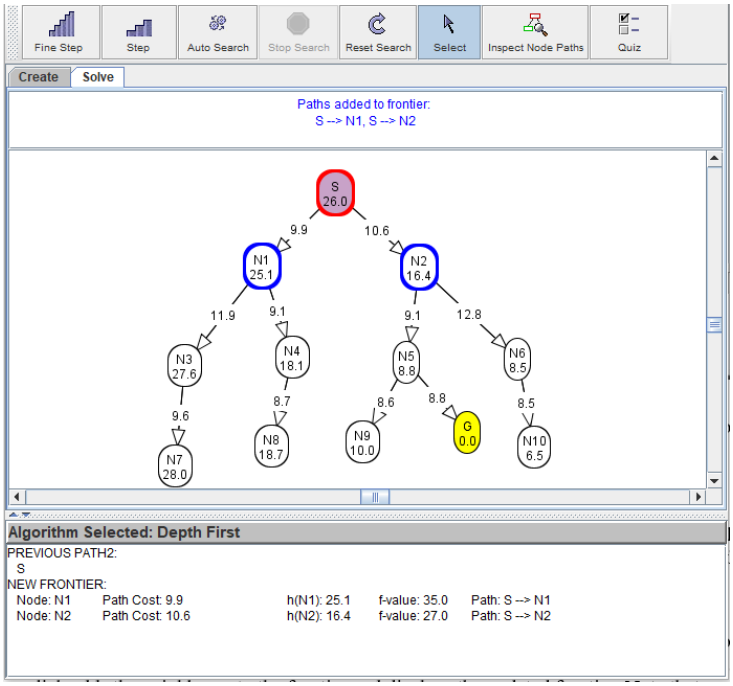



**Figure 6.7** *Graph after 'Step' button pressed once*

1. **Step:** When 'Step' button is clicked once, the search advances to visit one node, i.e. current node (selected from the first node in the frontier). The nodes on the frontier at the time and the neighboring nodes of the current node are displayed on the graph.
2. **Fine Step:** This button shows in three steps what happens whenever 'Step' button is pressed once. One click on the 'Fine Step' button selects the first node on the current frontier to make it the current node. A second click displays all the neighbors of the current node. A third click adds the neighbors to the frontier and displays the updated frontier.
3. **Auto Search:** As the name suggest, Auto search runs using the search algorithm selected and displays on the graph the order of each node being visited
4. **Quiz:** This is a fun button to use. By pressing this, the applet quizzes according to the search algorithm selected. To proceed with the searching, the correct node must be selected. After correct selection the node is shown as the current node and its neighbors are added to the frontier. The process continues till goal node is reached.

Before starting the search on the graph, observe the 'Legend' dialogue box which can be opened by clicking on 'Help' then 'Legend for Nodes/Edges'.

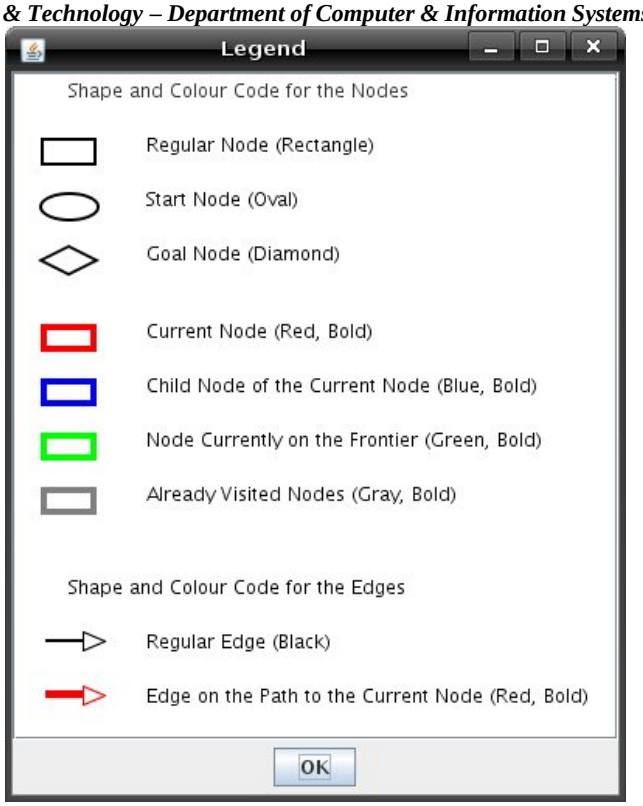



**Figure 6.8** *Legend for Nodes/Edges*

Finally, when the path is found from the start to the goal node, a dialogue box appears:

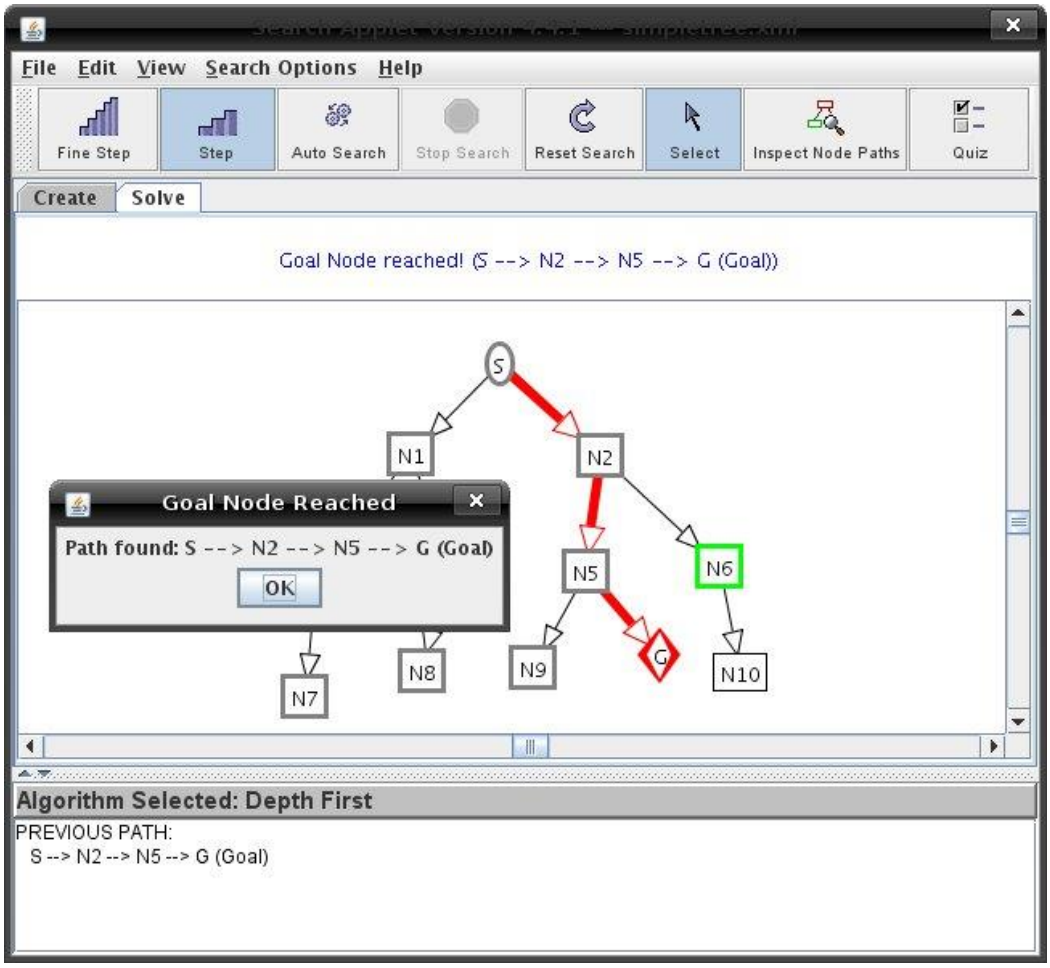



**Figure 6.9** *Graph after goal node is reached*

## Python code for Breadth First Search (BFS):

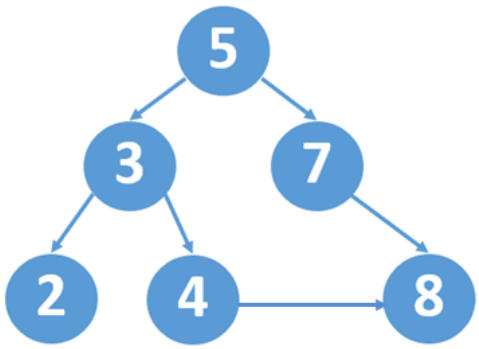



**Figure 6.10** *Example Graph for Code testing*

The code below first creates the graph for which we will use the breadth-first search. After creation, two lists are initialized, one to store the visited node of the graph and another one for storing the nodes in the queue.

Following these preliminaries, a breadth-first search function is declared with the parameters as visited nodes, the graph itself and the node respectively. The visited and queue lists get appended inside the function. Try the code below:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:

        if neighbour not in visited:
         visited.append(neighbour)
         queue.append(neighbour)
```

```
# Driver Code
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')    # function calling
```

### Python code for Depth First Search (DFS):

Using the same Figure 6.10, python code can be developed for DFS. As in the previous code for BFS, we first create a graph for traversal. Following this, a set for storing the value of the visited nodes is created to keep track of the visited nodes of the graph.

The DFS function is programmed with the parameters as visited nodes, the graph itself and the node respectively. The function checks whether any node of the graph is visited or not using the "if" condition. If not, the node is printed and added to the visited set of nodes. Next, the *for loop* we will go to the neighboring node of the graph and again call the DFS function to use the neighbor parameter. Try the code below:

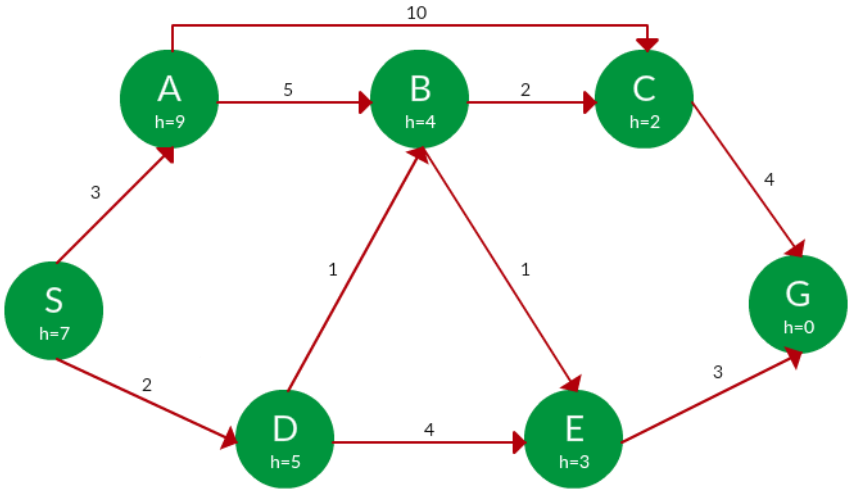
```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

## Exercises




1. Create given graph and solve it using both BFS and DFS. Determine sequences of paths that are explored by BFS and DFS. Also mention path cost.

| **BFS** | **DFS** |
|---|---|
| Path: _____ | Path: _____ |
| Path Cost: _____ | Path Cost: _____ |

| Attach screenshots here |
|---|
| |

2: Develop python code for Iterative Deepening Search (IDS). Write the program and attach output.

| Attach your code here |
| --- |

| Attach screenshot of output here |
| --- |

3. Develop a GitHub repository for your code share its URL here.

# Lab Session 7

## *Applying Informed Searching Techniques for Problem Solving*

### Searching Options:

### Algorithm Options:

There are six different algorithms that can be used to solve a graph. You can change what algorithm to select by clicking on the algorithm of your choice in the 'Search Options' menu. They differ in how they select the next node from the frontier. Below is a brief description of the algorithms.

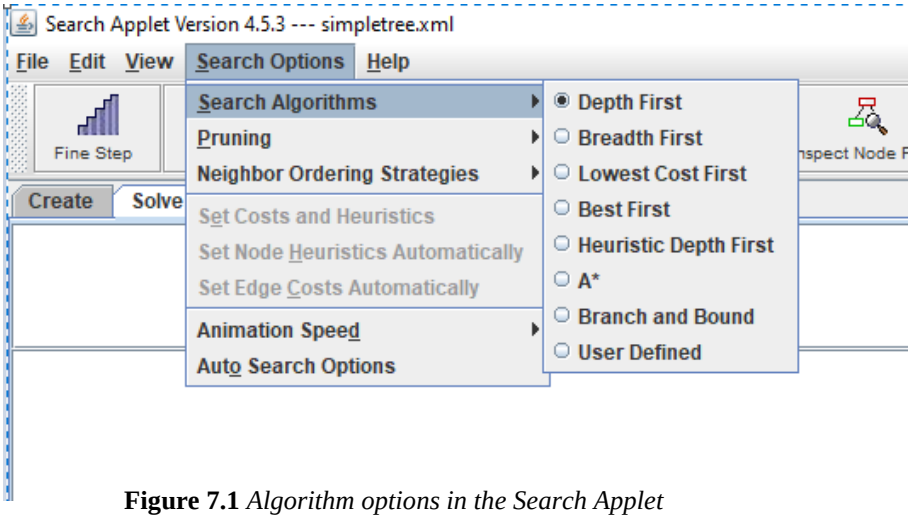



**Figure 7.1** *Algorithm options in the Search Applet*

**Uninformed - do not use any node heuristic value or edge cost information:**

- **Depth-first search:** tries to search paths to their completion before trying new ones. When a path fails, the algorithm backtracks to the last point at which it could have chosen a different path, and then tries that one. In other words, when a node is expanded, the next node to be chosen is the first child of the previous node.

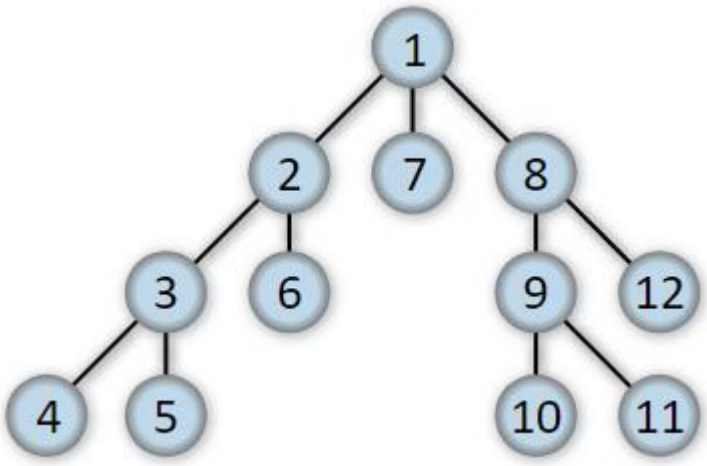



**Figure 7.2** *Order of node visit in Depth-first search*

- **Breadth-first search:** treats the frontier like a queue, first-in-first-out. It chooses the leftmost element in the frontier. If that element is not the goal node, its children are placed at the end of the queue and then the next leftmost element is checked.

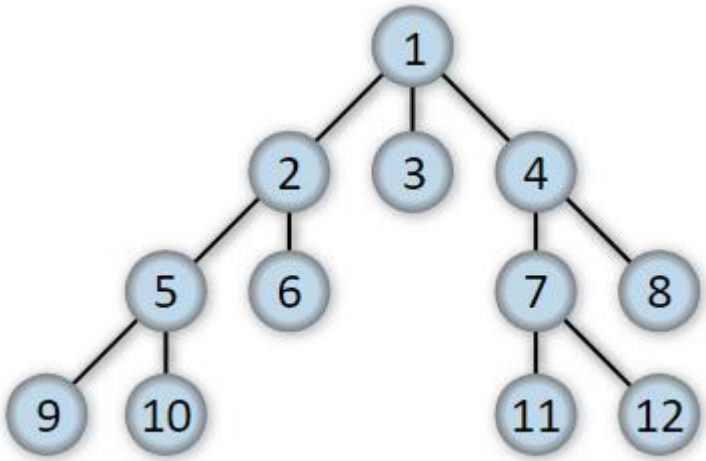



**Figure 7.3** *Order of node visit in Breadth-first search*

- **User defined:** user chooses whichever node of their choice on the frontier.

**Use edge costs:**

- **Lowest cost search:** chooses the element of the frontier with the lowest cost. It treats the frontier like a priority queue, ordered by g(n), which is a function that gives the cost of the path from the start node to the goal node.

**Use node heuristic values:**

- **Best first search:** chooses the element of the frontier with the lowest value of h(n), since it tries to choose the element that appears to be closest to the goal. It treats the frontier like a priority queue, ordered by the heuristic function.
- **Heuristic depth search:** a version of depth first search that uses heuristic knowledge to guide the search. It makes the local best choice, searching the most promising child of the previous node, instead of the leftmost child. Which child is most promising is determined by the value of h(n).

**Use both node heuristic values and edge costs:**

- **A*:** for each path on the frontier, this search uses an estimate, f(n), of the total path length from the start node to the goal node by adding the length of the path from the start node to a frontier node, g(n), plus the heuristic value of the node, h(n). Then the node with the lowest f(n) is chosen from the frontier.

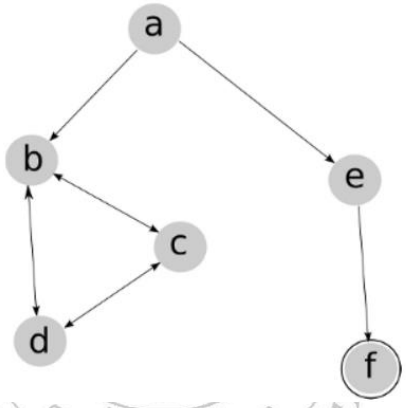
**Pruning Options:**

There are three different methods that you can choose when there are cycles in a graph. You can change the pruning options by clicking on the option of your choice in the 'Graph Options' menu. Below is a description of the different options:

- **None:** no checking.
- **Multiple-Path pruning:** if the node is already on the path from the start node to that node, check to see if the path to the goal node is shorter and choose the shortest path.
- **Loop detection:** prune a node that already appears on the path from the start node to that node.

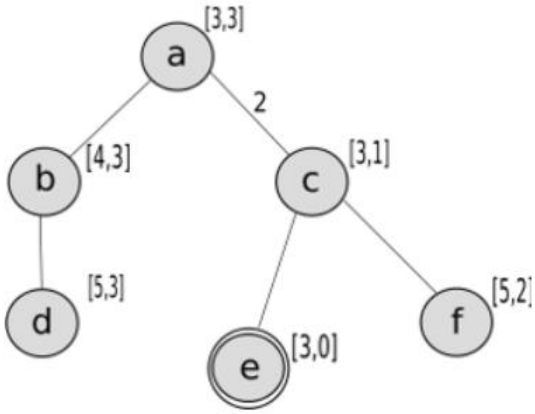
## EXERCISES

1. Consider the search problem represented in the given figure, where **a** is the start node and **f** is the goal node. Identify preferable searching algorithm (BFS or DFS). Give reasons to support your choice.




2. Which sequences of paths are explored by BFS and DFS in this problem?

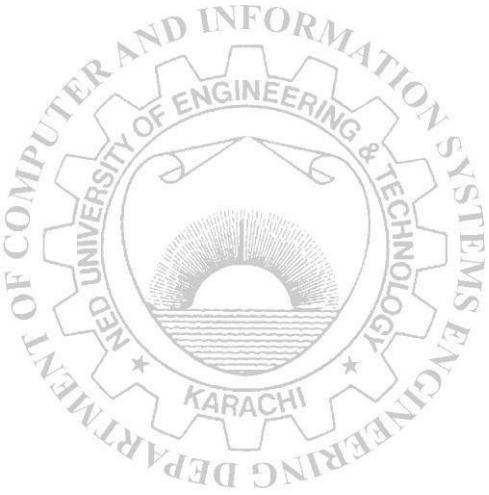
3. Consider the search problem represented in the following figure, where *a* is the start node and *e* is the goal node. The pair [*f*, *h*] at each node indicates the value of the *f* and *h* functions for the path ending at that node. Given this information, what is the cost of each arc? The cost <*a*,*c*> = 2 is given as a hint.




a) Is the heuristic function h admissible? Explain why or why not.

b) Trace A* on this problem. Show what paths are in the frontier at each step.

Attach screenshots here



# Lab Session 8

## *Solving CSPs by Enforcing Arc Consistency*

### Constraint Satisfaction Problems

Constraint satisfaction problems (CSPs) are pervasive in AI problems. A constraint satisfaction problem is the problem of assigning values to variables that satisfy some constraints. This constraint satisfaction problem solver (arc consistency) tool is designed to help you learn about solving CSPs with a systematic search technique called arc consistency.

The applet starts up in 'Create' mode, and you can see this because the 'Create' tab is the currently selected tab. If at any point, you would like to restart the CSP creation process from scratch after you've started to create a CSP, simply select 'Create New CSP' from the 'File' menu. If you have just started the applet, or created a new CSP, the screen should look like the one shown below:

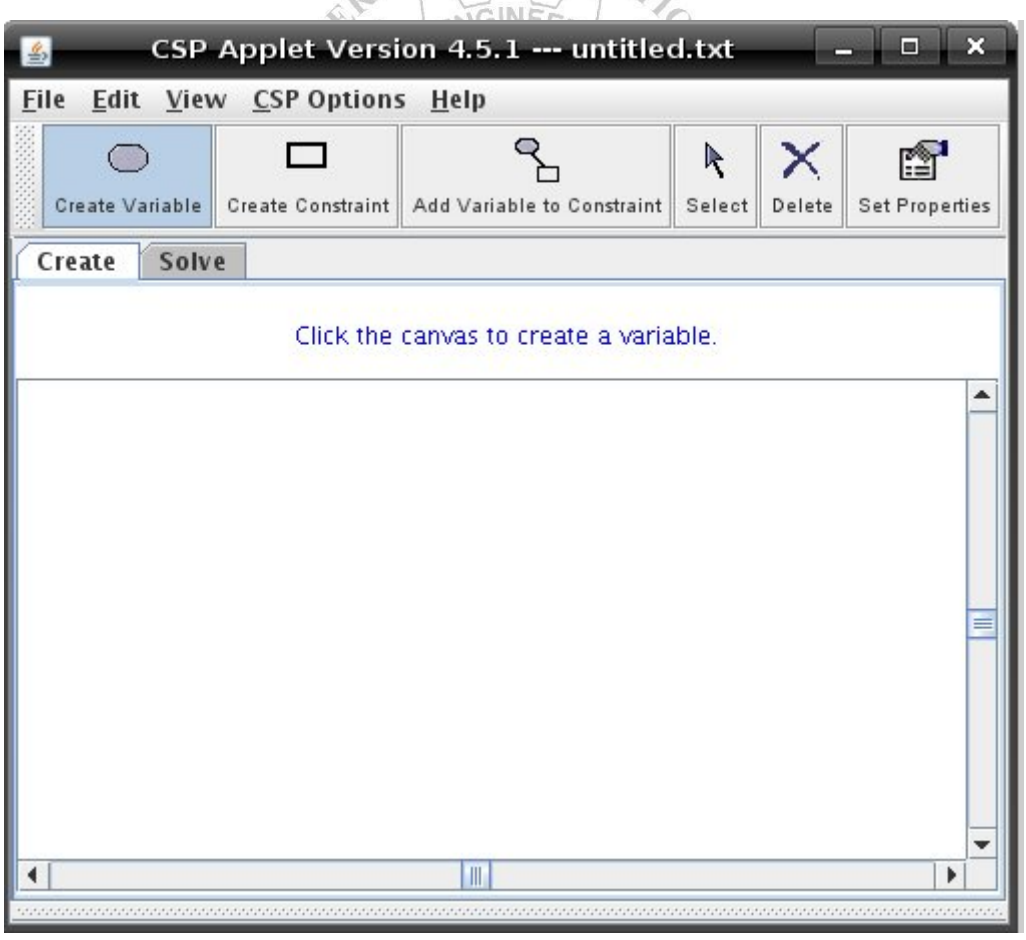



**Figure 8.1** *Applet in create mode*

The applet automatically begins in the 'Create Variable' submode, which you can see because it is the button that is depressed on the toolbar at the top of the window. If this button is depressed, then anytime you left click on the blank white canvas, a variable will be created, and similarly for each button that is depressed. Each time a button is clicked, the message canvas above the main canvas displays a message giving you information about what you can do next. If you click on the canvas now, the 'Variable Properties' dialog box will open:

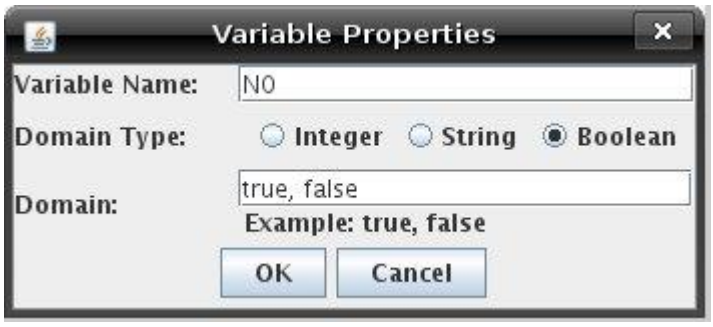



**Figure 8.2** *Dialog Box for Variable Properties*

Each time you create a new variable, the Variable Properties dialog box will appear with a new variable name 'Ni', where i gets incremented with each new variable. You can also enter a new name if you like, by deleting 'Ni' and typing a new name into this box.

A variable can be of type Integer, String, or Boolean. Select the appropriate radio button of the type that you want the current variable to be. For each type selected, an example of how to type in domain values in the 'Domain' field appears below the domain text area. The default domain values for type Boolean is 'true false'. There is no default values for the other types. After you have finished editing all the properties for this variable click 'OK'. The variable appears on the canvas, where your mouse was clicked, with the variable name and domain values visible. Also, you can always edit a variable again after it has been created by clicking on the 'Set Properties' button in 'Create' mode, and then clicking on a variable to bring up the 'Variable Properties' dialog box again. Create a few more variables if you like.

There are many ways to create constraints between variables. With the 'Create Constraint' button depressed, you have three options (these options also appear on the message panel when 'Create Constraint' is clicked):

- You can create an empty constraint, to add variables to later, by simply clicking once on a blank area of the white canvas.
- To create a unary constraint, click on a variable and then click on a blank area of the canvas.
- Finally, to create a binary constraint, you must first have 2 variables created. Click on one variable to start creating an edge. Click on another variable to create the constraint.

With the 'Add Variable to Constraint' button depressed, you can connect an existing variable to an existing constraint to add the variable to that constraint. Click on the variable or the constraint which you want to connect and the click on the respective constraint or variable.
For each of these options, a 'Constraint Properties' dialog box, similar to the one below will appear on the screen:

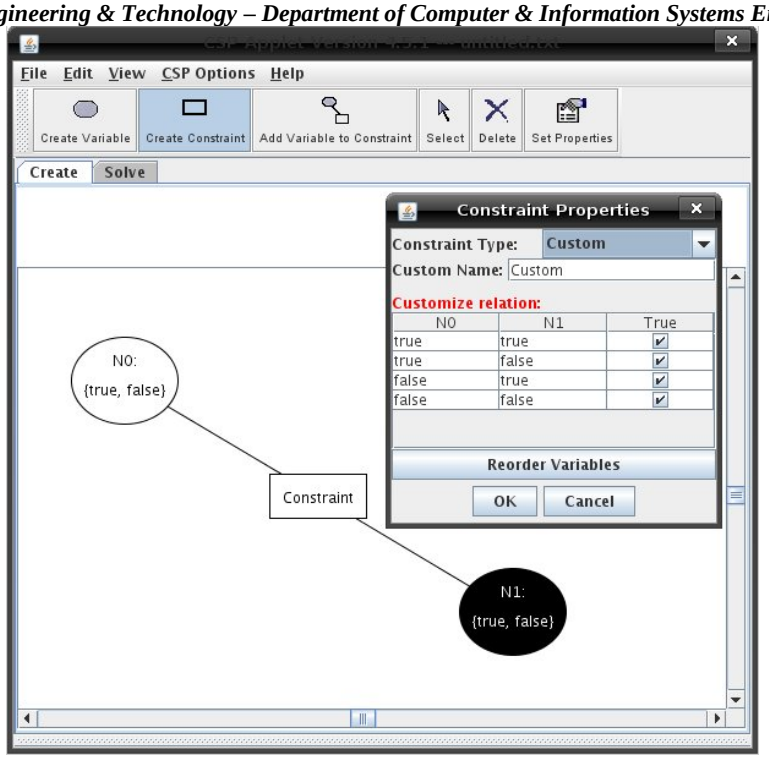



**Figure 8.3** *Dialog Box for Constraint Properties*

The default type of a constraint is "Custom". With this type, a truth table for all the variables participating in the current constraint appears below 'Customize relation'. You can manually select the truth values for each combination of domain values for the variables in the truth table by clicking on the 'True' checkboxes directly on the table. You can label the custom constraint by entering text into the text box next to "Custom Name." You can also select a built in constraint from the available constraint types in the 'Constraint Type' combo box. Different constraint types appear depending on the domain types of the variables participating in the constraint.

See the General Help page 'CSP Specification' to see the available built in constraint types. If you select a built in constraint type, you can take the complement of the type by checking the 'Take Complement' check box. You can reorder the variables participating in the constraint by clicking on the reorder variables button and following the directions. Note that only in some cases does reordering the variables in a constraint make a difference. Once you have finished editing the properties of the constraint, click 'OK' and the constraint will appear on the canvas as a square box, between its variables, with the constraint type visible. Also, every time you add a variable to the constraint, the 'Constraint Properties' dialog box will reappear so you can make the appropriate modifications to the existing constraint if necessary. And, just as for a variable, you can edit an existing constraint at any time by clicking on the 'Set Properties' button and then clicking on the constraint box to have the 'Constraint Properties' dialog box appear again.

After creating a few more variables and constraints, your CSP might look something like this:

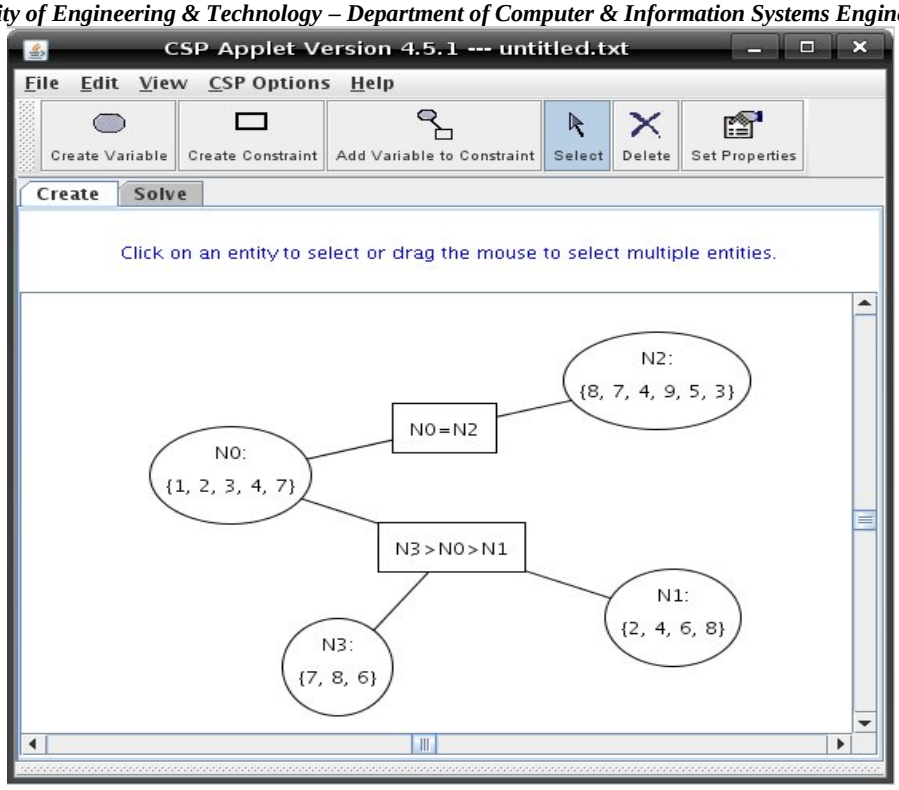



**Figure 8.4** *Constraint Satisfaction Problem*

## Solving a Constraint Satisfaction Problem

To start solving the CSP, switch over to 'Solve' mode by clicking on the 'Solve' tab. Some menu items which were previously unavailable in 'Create' mode are now enabled. Also, a 'Domain-splitting History' panel will appear at the bottom of the window. The toolbar buttons will change to give you solving options. The solve toolbar will initially look like the toolbar below:


**Figure 8.5** *Solve Toolbar*

Generally, with CSPs, you would first make a CSP arc-consistent and then check for solutions or, if necessary, recursively split a variable's domain and make the CSP arc-consistent again. An 'arc' between variables refers to the constraint relation between the variables. An arc is *arc consistent* if for each domain value in one variable, there exists a value in the domain of each other variable such that the constraint between the variables is satisfied. A CSP is 'arc consistent' if all of its arcs are arc consistent. A 'solution' to a CSP is an assignment of a unique value to each variable such that all of the constraints are satisfied.
This applet provides many ways to make a CSP arc-consistent:

- Clicking on the 'Fine Step' button will randomly pick arcs in the CSP and make them arc consistent. Blue edges mean that an arc has not been made arc consistent. The first fine step will highlight one of these blue edges and then proceed to make it arc consistent. If the domain of the variable connected to the edge is inconsistent with the constraint relation that it is connected to, then the second fine step will make the arc red. The third fine step will remove values from the domain of the variable that made the arc inconsistent if necessary. When an arc is consistent it will appear green. Note that a green arc may turn blue again if it becomes inconsistent as you are solving the graph and removing domain values from other variables.
- Clicking on the 'Step' button will also randomly pick arcs in the CSP and make them arc consistent. One 'Step' is equivalent to three 'Fine Steps'.

- Clicking directly on a blue arc will carry out a 'Step' on that arc to make it consistent.
- Clicking on the 'Auto Arc-Consistency' button will fine step through the entire CSP for you, until the CSP is arc consistent or has no solution.
- Clicking on the 'AutoSolve' button will recursively make the CSP arc-consistent and split domains until a solution is found. Clicking on this button again will find another solution, and so on until there are no more solutions. You can specify how you want AutoSolve to split domains by opening the 'AutoSolve Options' dialog under the CSP Options menu. Here you can specify how AutoSolve selects variables to split and how to split them. By default, AutoSolve selects the variable with the smallest number of domain values left and splits the domain in half.

You can stop Auto Arc-Consistency and AutoSolve at any time by clicking the 'Stop' button. To start solving again from the state in which you stopped, simply carry out any of the solving methods mentioned above. You can also change the speed of arc-consistency and select whether you want each fine step to be shown as the CSP is being solved or not. Both of these options are available in the 'CSP Options' menu.
You can reset the CSP to its initial state at any time by clicking on the 'Reset' button.
Once a CSP has been made arc consistent, there are three possibilities:

- There are no domain values left in any variable, which means the CSP has no solution, or no value assignment to each variable such that each constraint is satisfied. In this case the message panel will say the CSP has no solution.
- Each variable in the CSP has exactly one value left in its domain. This is a solution to the CSP, and the final variable value assignment will appear in the message panel.
- If each variable in the CSP has one value in its domain, but at least one variable has greater than one value, then there exists a solution, but domain splitting is required to find it.

After making our example CSP arc consistent it looks like this:

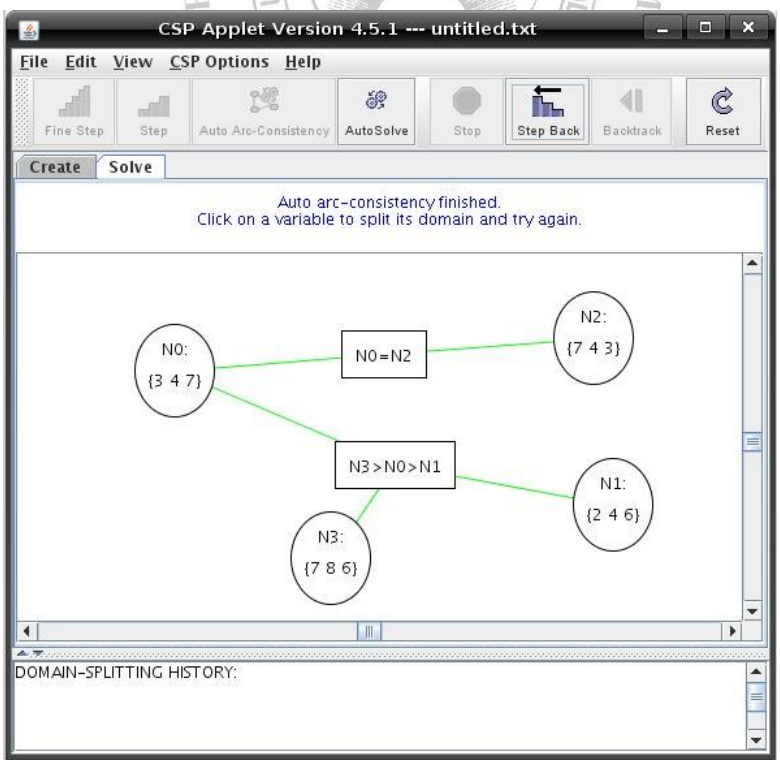



**Figure 8.6** *Arc Consistent CSP*

This CSP needs domain splitting to find a solution. Click on any variable that has greater than one value in its domain to split it. The "Split the Domain..." dialog will be shown as below:

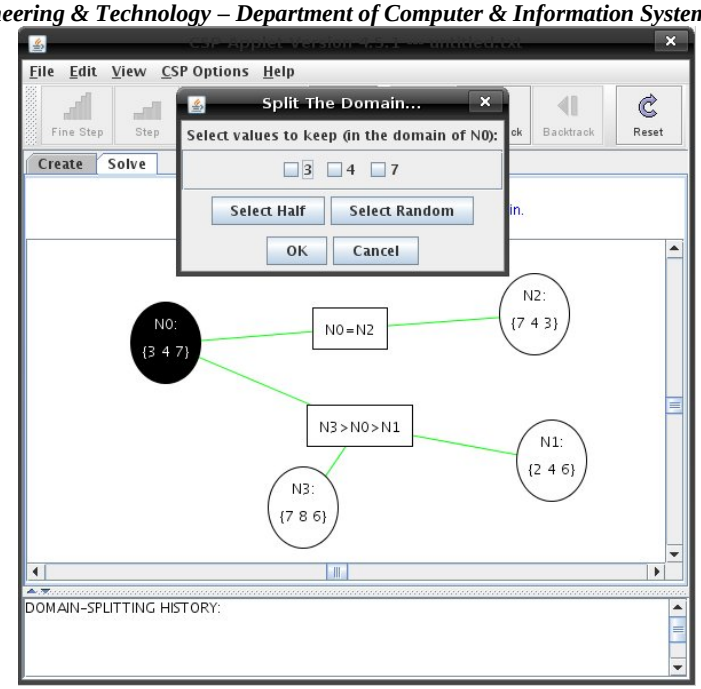



**Figure 8.7** *Split the Domain Dialog Box*

The domain values for this variable are displayed. You can manually select which domain values you would like to keep, to solve the CSP with, by clicking on the corresponding value checkboxes. You can also allow the applet to select the first half of the values to keep, or randomly select values to keep. Once you have a reduced domain, some of the arcs in the CSP that were green will have turned blue, meaning that the CSP has to be made arc consistent again. After each split the reduced domain values of a variable will appear in the 'Domain-splitting History' panel at the bottom of the applet window.
A solution to the CSP may not exist given a certain split. In this case a failure for that variable assignment will appear in the 'Domain-splitting History' panel. You can then recover the variable domain values that you discarded when you split a domain by clicking on the 'Backtrack' button and then try solving again.

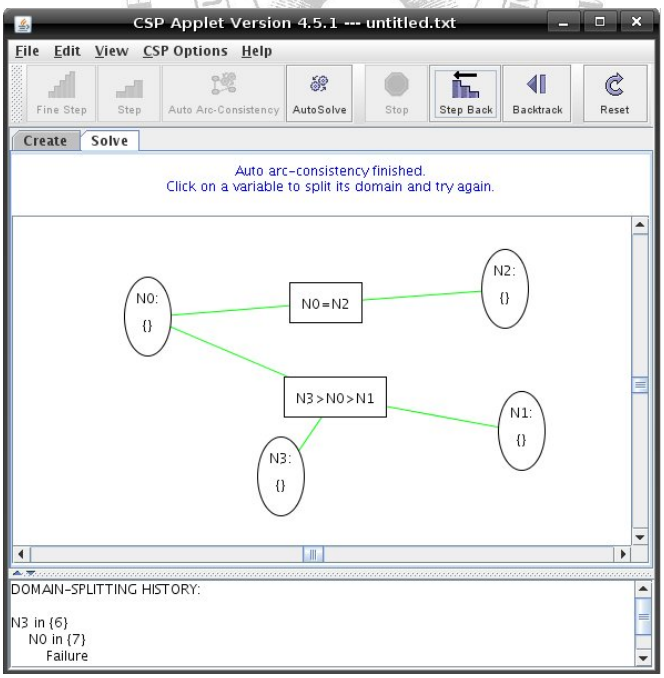



**Figure 8.8** *Backtrack button option*

You may have to split, solve, and backtrack through a CSP recursively until a solution is found.

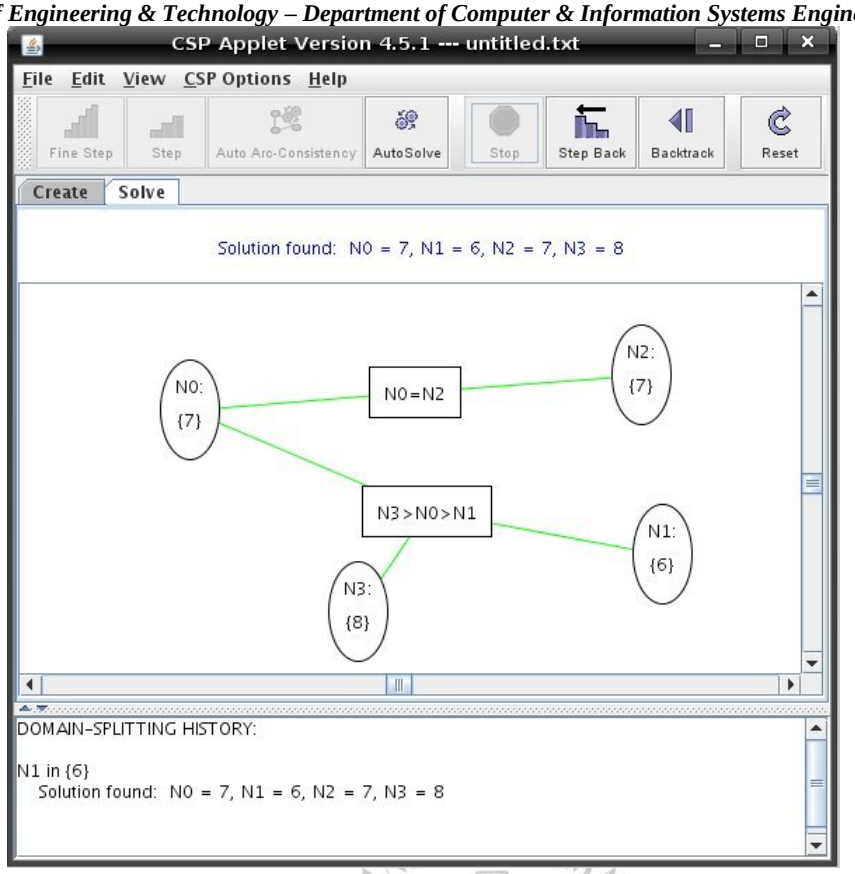



**Figure 8.9** *Solution of CSP*

The history of this process is kept for you in the 'Domain-splitting History' panel.

```
DOMAIN-SPLITTING HISTORY:

N0 in {1 2}
   Failure
N0 in {3 4 7}
   N0 in {3}
      N3 in {7 8}
         N3 in {7}
            Solution found:  N0 = 3, N1 = 2, N2 = 3, N3 = 7
         N3 in {8}
            Solution found:  N0 = 3, N1 = 2, N2 = 3, N3 = 8
      N3 in {6}
         Solution found:  N0 = 3, N1 = 2, N2 = 3, N3 = 6
   N0 in {4 7}
      N1 in {2}
         N3 in {6}
            Solution found:  N0 = 4, N1 = 2, N2 = 4, N3 = 6
```

**Figure 8.10** *Domain Splitting History*

The CSP applet comes with several pre-defined examples to allow you to start working with CSPs without having to create one yourself. To load a sample CSP, go to the 'File' menu and select 'Load Sample CSP'. A dialog will open with a drop-down menu allowing you to select a particular CSP.

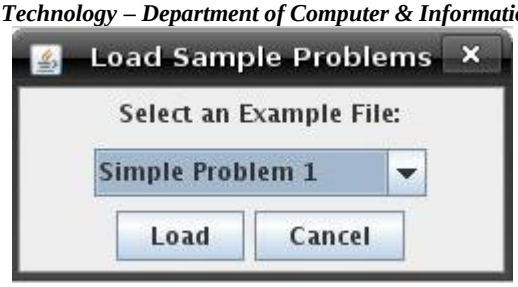



**Figure 8.11** *Dialog Box to load sample problems*

- **Simple Problem 1:** This simple CSP uses three integer variables and two constraints.
- **Simple Problem 2:** This simple CSP uses three integer variables and three constraints.
- **Scheduling Problem 1:** This CSP uses integer variables and some available integer constraints to model an example scheduling problem of trying to schedule meeting times.
- **Scheduling Problem 2:** This CSP is similar to 'Scheduling Problem 1' but more difficult to solve and uses some custom constraints.
- **Crossword Problem 1:** This is a 3x3 crossword problem (where A1, A2, A3 are the three rows across; and D1, D2, D3 are the three columns). The values of the domains are 3-letter words.
- **Crossword Problem 2:** This is a simple crossword problem with five variables and five constraints.
- **Five Queens Problem:** This models the problem of placing 5 queens on a 5x5 chessboard such that no two queens can attack each other.
- **Eight Queens Problem:** Similar to "Five Queens Problem," except this time with eight queens on an 8x8 chessboard.

## Exercises

1. Imagine the following scenario: a family of four needs to figure out how each family member will commute to work or school given several constraints. The family consists of a mother, father, son and daughter. Each family member can bicycle or ride in the car. Additionally, the son has a pogo stick he can use for commuting to school. The assignment of transportation modes to family members is subject to the following constraints:
   - There are only two bicycles.
   - The car can only hold three people.
   - The son and daughter must take the same mode of transportation.
   - The son and daughter can only go by car if at least one of the parents is going by car, i.e. the parent(s) driving them to school.

What are the variables in this problem? What values are in the domain of each variable? Write down your answers before continuing to the next task.

_____

_____

_____

_____

2. Open the consistency for CSP tool and load the file http://www.aispace.org/exercises/FamCommuteCSP.xml by clicking File → Load from URL. Inspect the CSP problem closely. Are these the variables you listed in the previous section? Look at the domain of each variable. Right-click on one of the constraints and select "Set Properties of Constraint." Examine the current constraint properties. Do the settings seem sensible given what you know about the constraints from section 2? Why or why not?

_____

_____

3. Given what you know about the values in the variable domains and the given constraints, do you expect that arc consistency will remove any values from any variable's domain?
   - Click the "Solve" tab.
   - We will now run arc consistency. You can adjust the arc consistency speed to be slower or faster depending on how closely you want to monitor the process, by selecting CSP Options → Arc-Consistency Speed from the menu.
   - Click the "Auto Arc-Consistency" button to begin.

   Was the result what you expected? Why or why not?

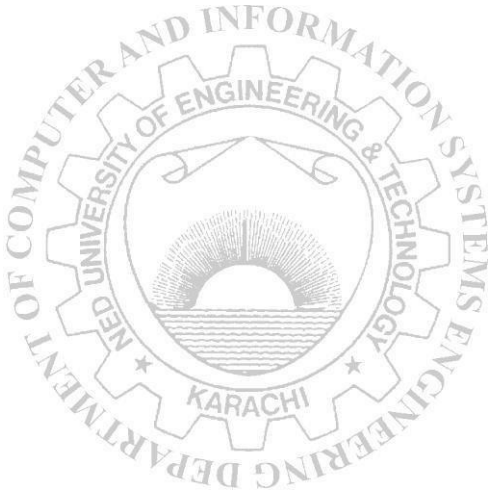
4. Before we try to find a solution to the CSP, make a prediction about what solution(s) exist(s). List as many as you think we will find. You can now click "AutoSolve" to attempt to solve the CSP. If one solution is found, you can click "AutoSolve" again to search for a second, third, fourth, etc. How many solutions were found and what were they?

5. We'll now add an additional constraint. We want a unary constraint that says the daughter cannot go by car.

- Go back to the "Create" tab.
- Click the "Create Constraint" button at the top and then click on the canvas near the lower-right corner. Give the constraint a sensible name and click "Ok."
- Click the "Add Variable to Constraint" button and link the constraint and relevant variable by left-clicking each.
- Look at the constraint properties table and make sure you select/unselect the correct boxes for this constraint.
- Move back to the "Solve tab." Make a prediction about how many solutions there will now be. Then go through the steps in sections 4 and 5 again.

  How many solutions were found? Is there a simpler way of specifying that the daughter cannot go by car, instead of adding a unary constraint as we did?

Attach screenshots here



# Lab Session 9

## *Solving CSPs Using Stochastic Local Search Techniques*

Consider the following CSP:

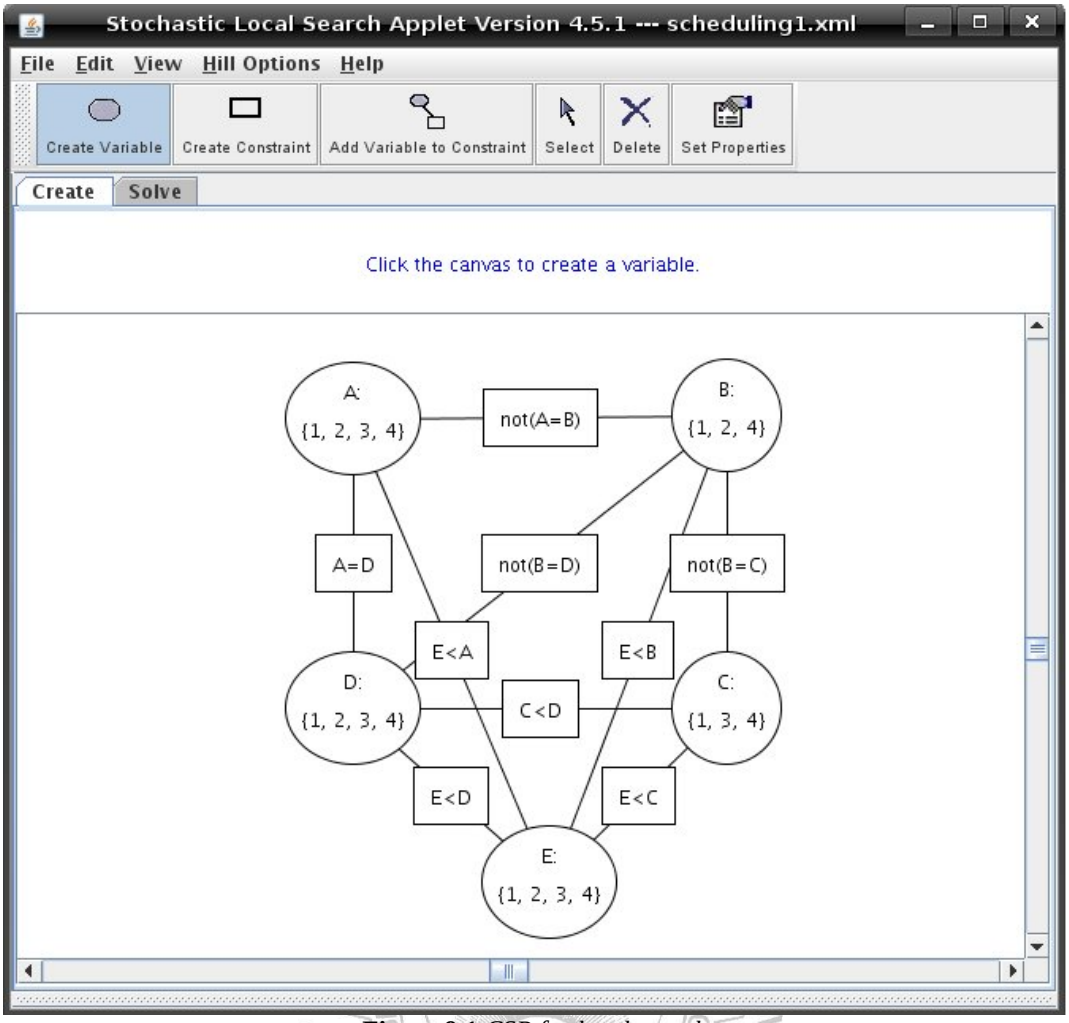



**Figure 9.1** *CSP for local search*

## Solving a Stochastic Local Search Based CSP

Once you're done building a Constraint Satisfaction Problem (CSP) in the applet, you can begin searching for a solution by switching to 'Solve' mode. Click on the 'Solve' tab above the main white canvas to do this. The toolbar buttons will change to give you solving options. Some menu items which were previously unavailable in 'Create' mode are now enabled. Also, the selected algorithm name is visible below the canvas. The solve toolbar will initially look like the toolbar below:


**Figure 9.2** *Solve Toolbar*

To begin searching for a solution, you must first initialize the CSP by clicking the 'Initialize' button. This will assign a value to each variable in the CSP, from its domain, randomly or by lowest value. The initialization method can be set in the 'Algorithm Options' dialog under the 'Hill Options' menu.

Each variable in the CSP now has one value assigned to it. Also, some arcs appear green and some arcs appear red. Red arcs mean a constraint between variables is not met with the given variable value assignment. Green arcs mean the constraint is consistent. As you search for a solution, and variable assignments change, so will the colors of arcs.

When you initialize the CSP, a "Trace" window will appear showing the initial variable-value assignment and the number of conflicts for that assignment. This window will always correspond to the CSP in the main applet window, even as you step or solve.

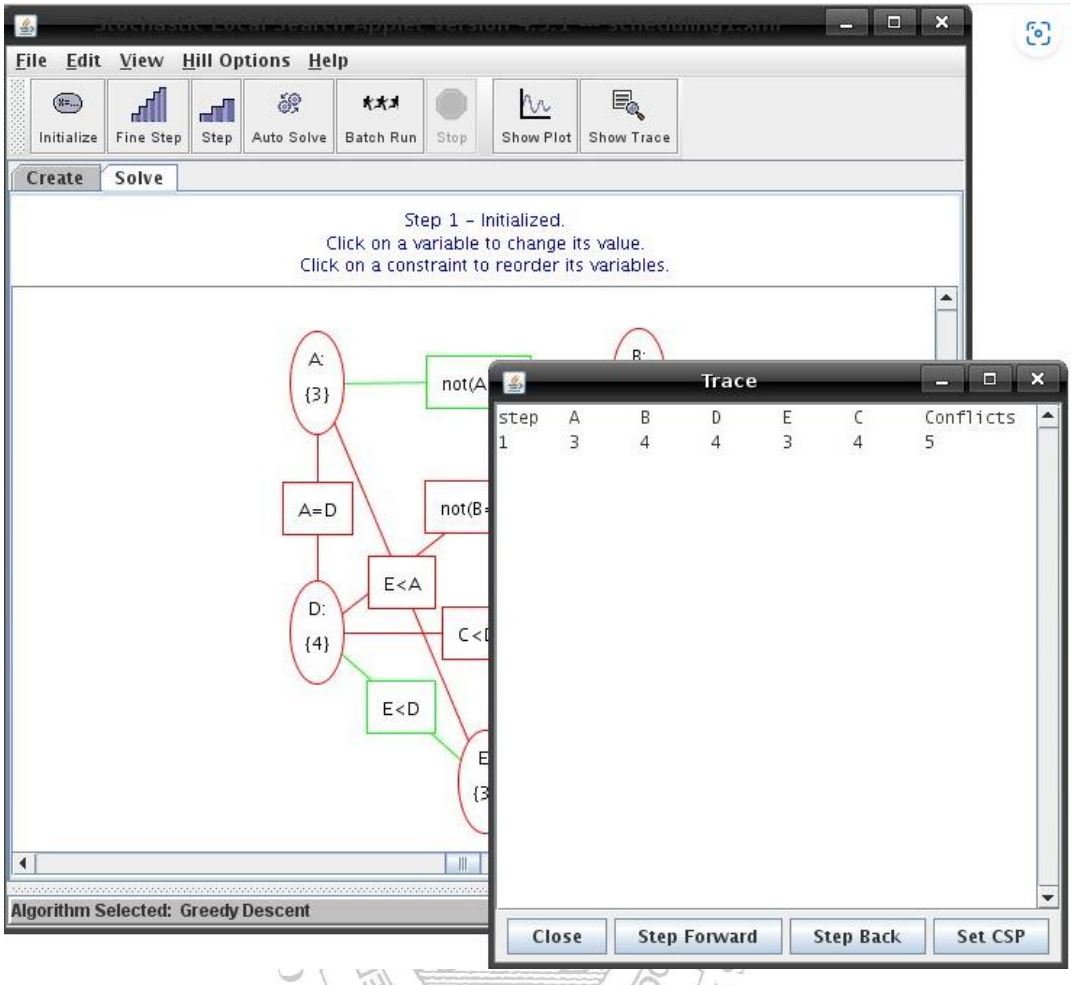



**Figure 9.3** *Initialization of CSP*

Now that the CSP has been initialized, you can begin searching for a solution. You can 'Step' through the search manually, by clicking the 'Step' button. This will let the applet pick a variable or value to change in the CSP according to the search algorithm you selected. With some algorithms, you can also 'Fine Step' through the search. In these cases, one 'step' is equal to two 'fine steps'.

If you get tired of clicking the 'Step' or 'Fine Step' button, you can use the 'Auto Solve' button to let the

applet step continually, according to the specified algorithm, until a solution is found or until the predefined number of steps have been completed. If a solution is found, the message canvas will display the value assignments that solve the CSP. Recall that a solution means that there is a unique value for each variable such that all the constraints are satisfied. If there is a solution there should be zero conflicts left in the 'Trace'

window.

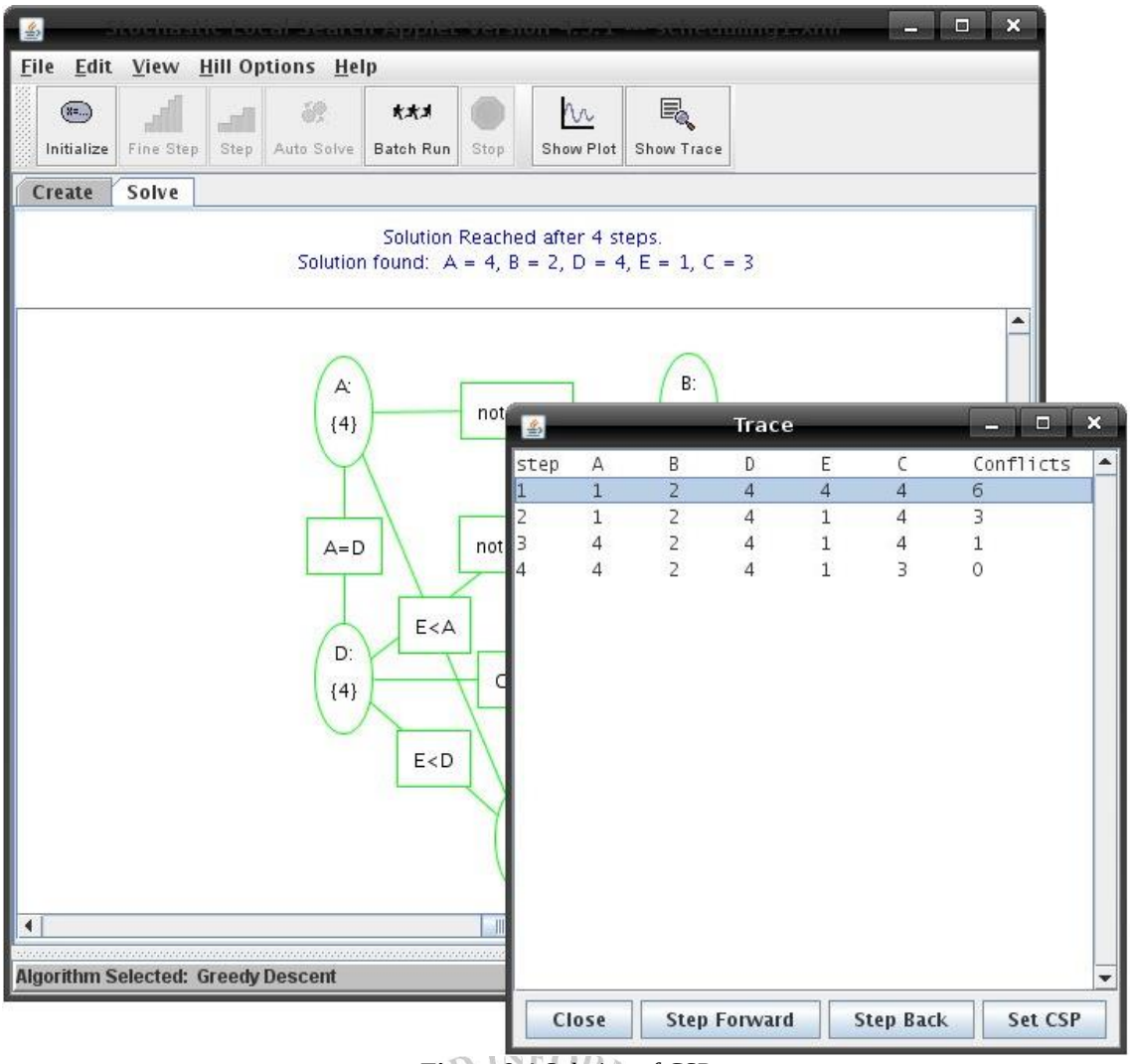



**Figure 9.4** *Solution of CSP*

The number of steps to stop solving at can be specified in the 'Algorithm Options' dialog under the 'Hill Options' menu. The default termination number is 100 steps. You can also manually stop the applet as it searches for a solution by clicking on the 'Stop' button. You may then resume solving by any of the methods mentioned above.

As you search for a solution, the 'Trace' window will trace the steps and variable assignments. You also have the option of stepping back through the trace by clicking on the 'Step Back' button in the 'Trace' window. And if you click on a specific assignment (a row in the 'Trace' window), you can set the CSP back to that assignment by then clicking on the 'Set CSP' button. The graphical display of the CSP will update to correspond to the current assignment in the 'Trace' window.

This may be useful if you want to look at a certain assignment in more detail, or if you want to make your own change to the CSP after a certain assignment. You can manually change a variable's value by clicking on the variable in the CSP and then clicking on a value from the possible domain values for that variable that appear in the 'Node Details' dialog box that comes up. As you hover over a domain value in this dialog, the arrows pointing to the node will update to show how many constraints would be satisfied with that value assignment. Green arrows mean a constraint is satisfied, red arrows mean it is inconsistent for that value assignment.

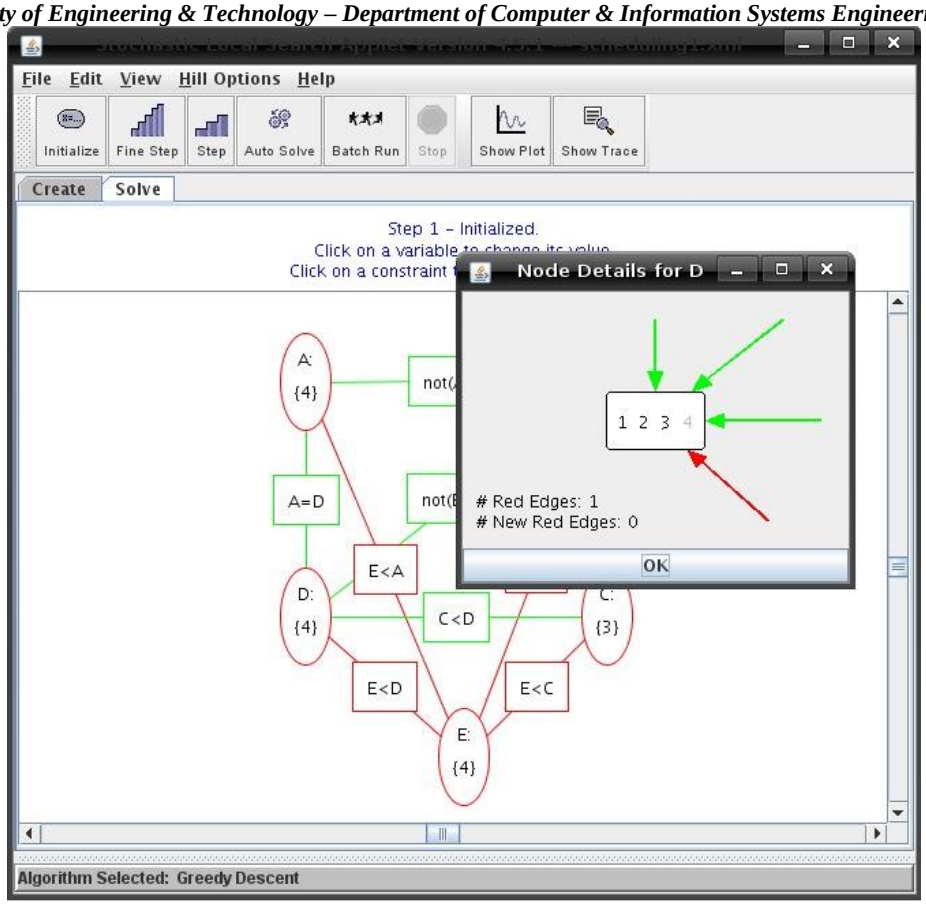



**Figure 9.5** *Green and red arrows in CSP*

You can view a plot for a search by clicking on the 'Show Plot' button. The 'Hill-Climbing Plot' window that appears plots the number of conflicts over the step count, just as the 'Trace' window shows the number of conflicts with each step.

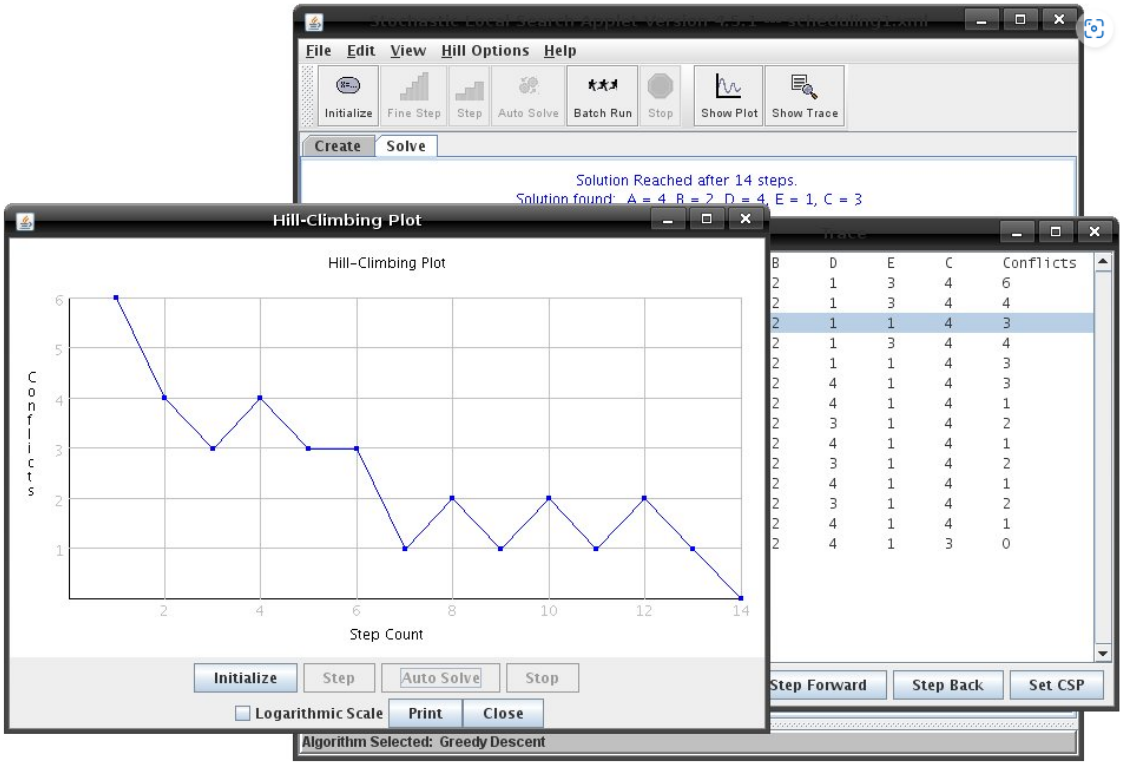



**Figure 9.6** *Hill Climbing Plot*

The applet also allows you to automatically run several searches, using the selected algorithm, in a batch run by clicking on the 'Batch Run' button. A runtime distribution of the batch run will appear in the 'Batch Plot' window. You can set batch run properties such as the number of trials to attempt per run, or the termination number in the 'Batch Run Options' dialog under the 'Hill Options' menu. You can also plot more than one batch run and compare their plots. To view the detailed statistics and settings of any plot, either click on the "Statistics & Settings" button, or click on the plot number of the plot you want to analyze in the legend to the right of the runtime distribution.

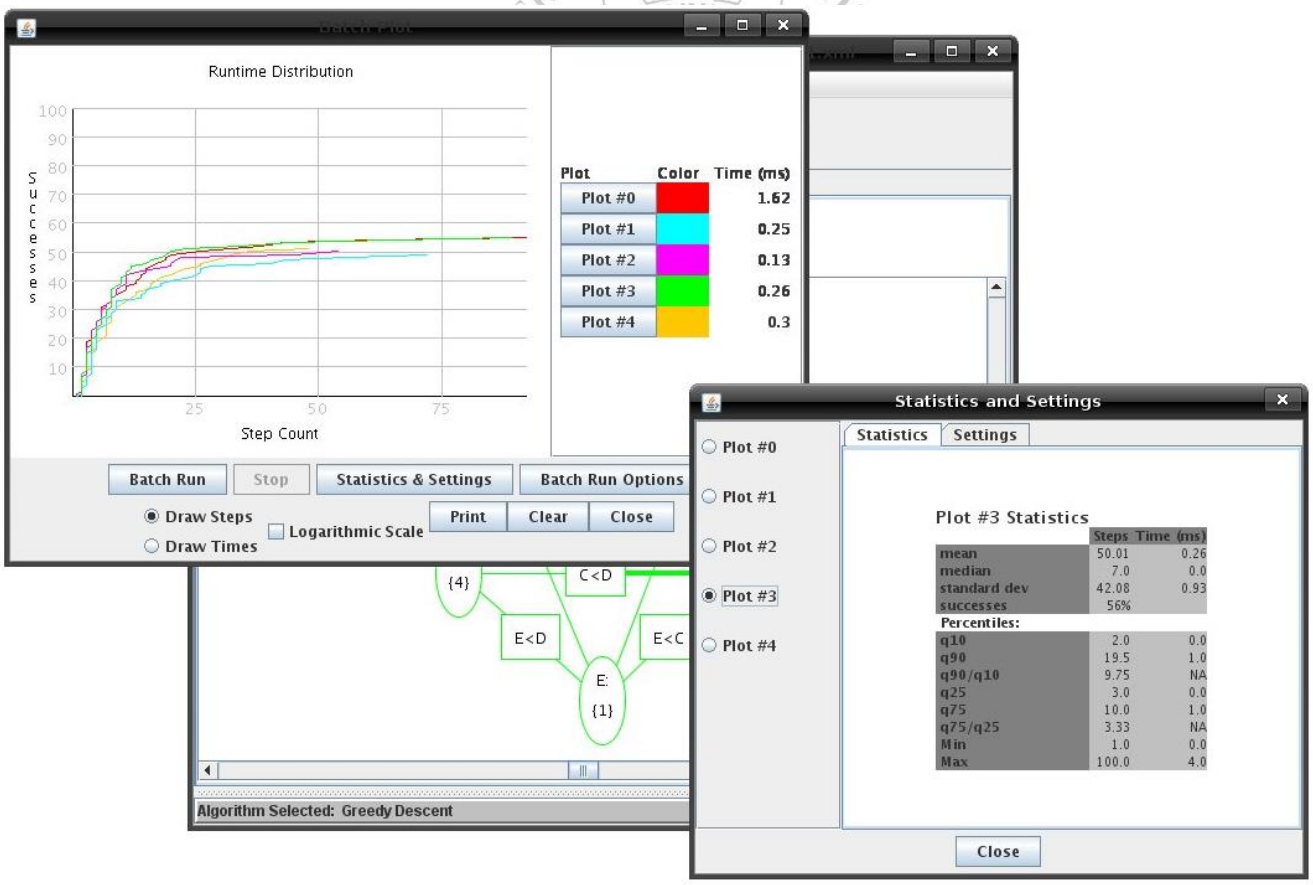
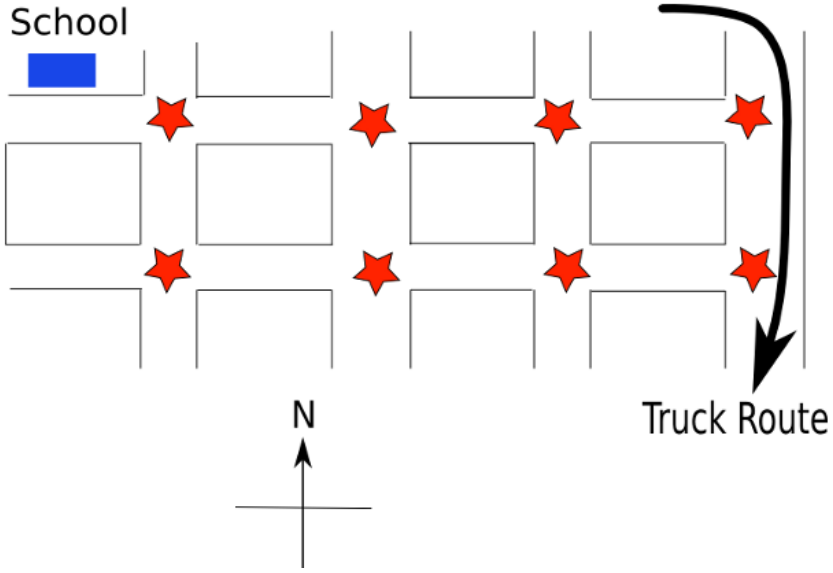



**Figure 9.7** *Runtime Distribution*

## Exercises

Consider the following scenario. You are on a city planning committee and must decide how to control the flow of traffic in a particular residential neighborhood. At each intersection, you have to decide whether to install a 4-way stop, a roundabout, or an uncontrolled intersection (in an uncontrolled intersection, the streets intersect without signs, roundabouts or other traffic controls). There are several restrictions in how the intersections can be controlled. The following figure gives an overview of the neighborhood.

The red stars indicate the 8 intersections under consideration. The intersection nearest the school in the NW corner of the neighborhood must contain a 4-way stop for safety reasons. Along the east side of the neighborhood runs a truck route, and no roundabouts can be placed on this street because they pose a problem for large trucks. Also, it is not allowed to have 4-way stops at consecutive intersections or to have two consecutive uncontrolled intersections. Finally, due to the cost of installing roundabouts compared with the other options, each block can have at most one of its four corners with a roundabout.




1. How would you represent the above problem as a CSP? Identify the variables, their domains, and the constraints involved. Once you are done a sketch on paper, open the stochastic local search tool and load the file http://www.aispace.org/exercises/roundabouts.xml by clicking File → Load from URL. This shows one possible representation, but there might be more than one correct representation.

2. Using the local search tool, experiment with several local search algorithms for solving this problem.
   - Greedy Descent
     From the menu, choose Hill Options -> Algorithm Options and then select Greedy Descent from the dropdown menu. Click Ok. Click Initialize. This will assign a value to each variable. Note: you can choose Hill Options -> Auto Solve Speed -> Very Fast to speed up the solver. Click Auto Solve. What happens? Does it find a solution within 100 steps? Hypothesize why or why not. Now click Batch Run, which will calculate the runtime distribution and plot the percentage of successes against the number of steps. What does the runtime distribution tell you about this solver?
   - Greedy Descent
     Go back to Algorithm Options and now select Greedy Descent with Random Restarts. Click Batch Run again and compare the runtime distributions. Do the random restarts improve Greedy Descent? Why or why not?
   - Random Walk
     Go back to Algorithm Options and now select Random Walk. Click Batch Run again. How does this compare with plain Greedy Descent? How would the two algorithms compare if you gave them 10000 steps? (a logarithmic scale might help the visualization)

# Lab Session 10

## *Developing Knowledge-Based Systems*

### Expert Systems

Expert systems (or knowledge-based systems) allow the scarce and expensive knowledge of experts to be *explicitly* stored into computer programs and made available to others who may be less experienced. They range in scale from simple rule-based systems with flat data to very large scale, integrated developments taking many person-years to develop. They typically have a set of **`if-then`** rules which forms the *knowledge base*, and a dedicated *inference engine*, which provides the execution mechanism. This contrasts with conventional programs where domain knowledge and execution control are closely intertwined such that the knowledge is *implicitly* stored in the program. This explicit separation of the knowledge from the control mechanism makes it easier to examine knowledge, incorporate new knowledge and modify existing knowledge.

### VisiRule

VisiRule is a graphical tool developed by Logic Programming Associates that can be used to draw and execute decision charts. Nodes are the main constructs in VisiRule that represent questions and/or computable functions and expressions that cover the various paths through the network. The problem domain into multiple files each containing one or more charts. Charts can have continuation nodes to support modularity and scalability.

VisiRule generates code in the form of Flex rules (LPA's expert system product) that can be executed, inspected, and exported for embedding in external applications. Charts can be exported as Windows Metafiles (WMF) for usage within other common desk- top applications such as Word.

### Starting VisiRule

Clicking on VisiRule icon opens the following window as in Fig 10.1

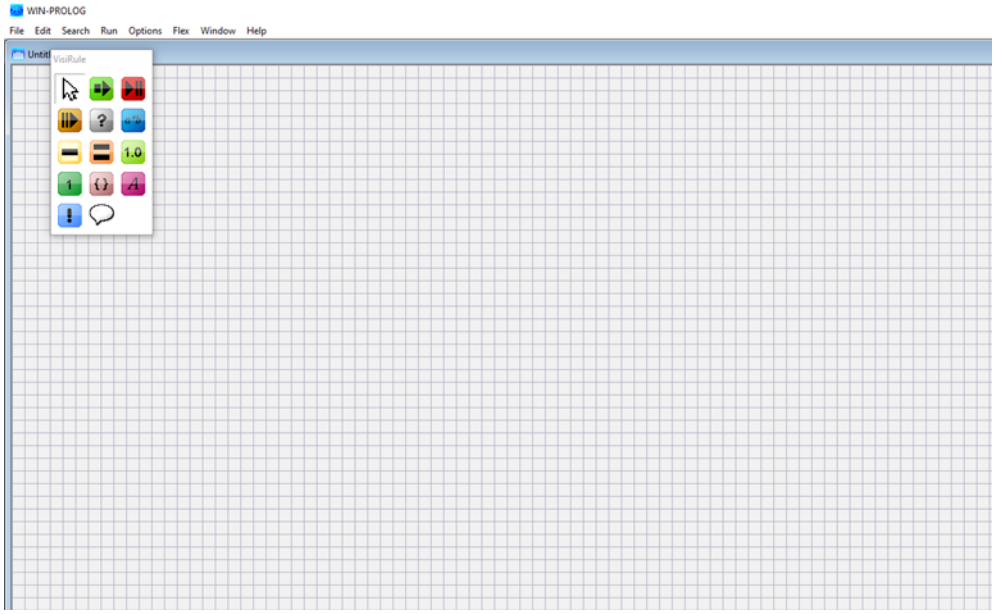



**Figure 10.1** *VisiRule start-up window*

## Important Pragmatics when using VisiRule

A VisiRule chart is composed of an arbitrary number of connected boxes (nodes) of different types. There are various steps in drawing a chart:

- creating boxes
- linking boxes
- editing the text of boxes
- aligning and resizing boxes

Once a 'complete' chart is created, i.e. one with at least one start node and one end node and all other nodes connected correctly, then the code generation and execution can be used to run the chart.

## Creating Boxes and Links

The order in which the boxes and links are created is not significant. If the structure of the chart is known, all the boxes may be drawn first, followed by making the connections.

## Box Types

Boxes can be created in the same mode, and then set to the correct box type using the Right-Click properties option. But normally, authors switch to the correct box type in advance of creating a new box.

**Table 10.1** *Different box types in VisiRule*

| **Different Types of Boxes** | |
|---|---|
| **Color** | **Explanation** |
| Yellow | Single choice question |
| Salmon | Multiple choice question |
| Pale Green | Number Input question |
| Gray Green | Integer Input question |
| Pink | Set Input |
| Magenta | String Input |
| Green | Start |
| Red | End |
| Orange | Continue |
| White | Expression |
| Blue | Code |
| Light blue | Statement |

## Editing Boxes

Boxes do come with a certain amount of default pre-filled text; for instance, questions always come with a numerically generated unique question name, a prompt text called 'prompt', and an explanation text called 'explanation'. They can be edited at any stage.

## Relative Positioning of Boxes

Whilst the order in which boxes are created is not significant, their physical location is; as the VisiRule code generator uses a left to right depth-first search mechanism when collecting menu items to populate menus and also when working out which expression to test in which order.

## Box Fields

Box fields can be suppressed. So, for instance, if we do not want to display question names (they are only of internal usage), we can make the first field of any, some or all questions hidden and therefore no longer visible.

## Box Alignment

You may choose to leave final alignment of boxes until your chart is established. Prior to that, it does make sense to try and get boxes to line up as this keep the links vertical and/or horizontal. You are STRONGLY advised to use the right and left arrow keys to nudge selected boxes into line.

## Avoid using Spaces and Hyphens

Try NOT to use spaces or hyphens (or even a CapitalInitialLetter), as this means quotes will be needed in say an expression. You CAN always use underscores (this_is_ok).

## VisiRule Floating Toolbar

The floating VisiRule toolbar palette contains all the tools that are used to build a VisiRule chart. It can be shown or hidden using the "Options/Show Toolbar" menu item and can be moved to any position you desire. The VisiRule toolbar is a global resource, shared by all the VisiRule Worksheet windows. So, there can be many files open at the same time.


**Figure 10.2** *VisiRule toolbar*

Now let's build and run a simple chart

If you lose the floating VisiRule palette, you should be able to get it back from the Option menu.

## Creating a VisiRule Chart

## Start Node

Every chart starts with a Start node – so select the Green Box (containing the right arrow head). Now click in the VisiRule blue canvas and drag a rectangle. When the mouse is released, the start node is created:

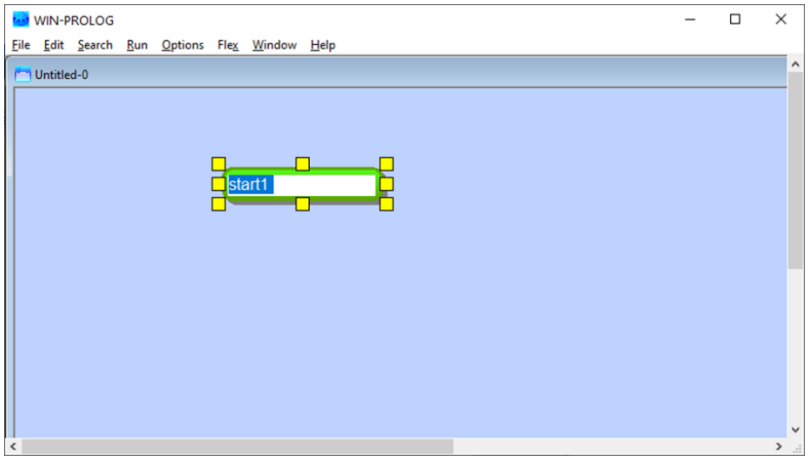



**Figure 10.3** *Creating the Start Node*

## Single Choice Question

Now let's add a single choice question. Go back to the floating tool palette and select the light brown box on the 3rd row with a single bar in it:

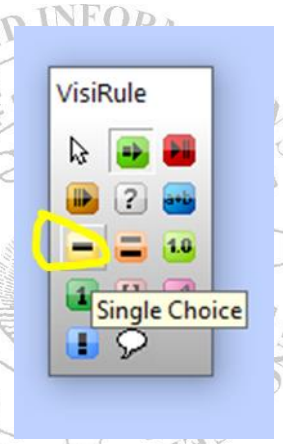



**Figure 10.4** *Single choice question*

Now click and drag in the canvas to generate the single choice question.

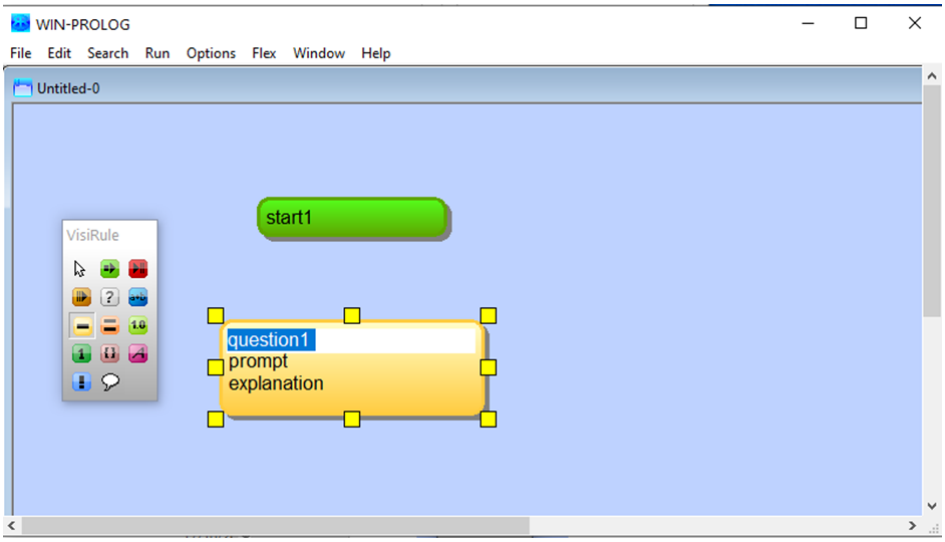



**Figure 10.5** *Generating Single choice question*

Now these two boxes can be joined up simply by clicking back in the start node and drag a line to the question node (and releasing it). Note that select should not be active while joining boxes.

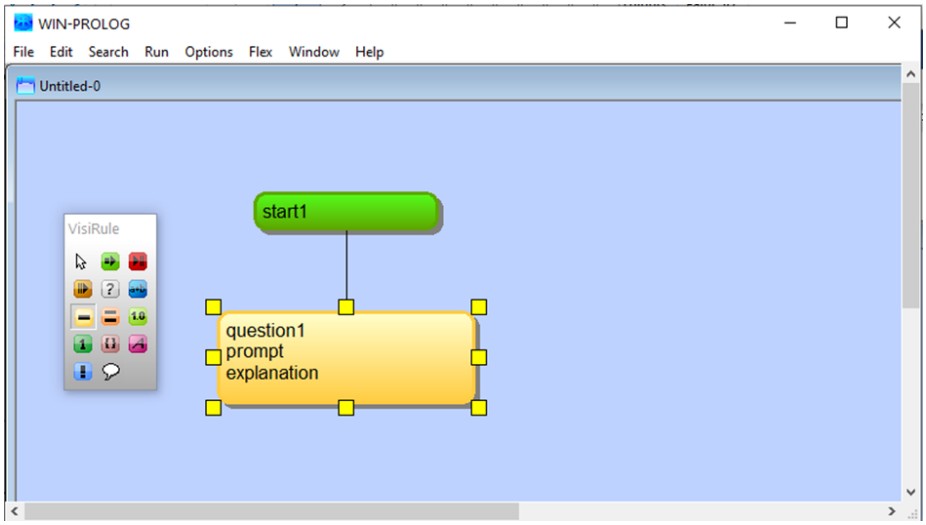



**Figure 10.6** *Boxes connected successfully*

If the link is successful, the Target box will get selected – this is indicated by the 8 square yellow boxes.

However, if this does NOT happen, then probably the Target was missed and now it needs to be deleted and tried again.

Next, we can edit the contents of the question.

Each question has 3 fields: **Name, Prompt and Explanation**

We can tab between them (or use the dedicated extended VREDITOR which is supplied separately)

Best practice is to name the question with something relevant and give useful prompts. Explanations can be completed later.

Click on the question and rename it – in this case to Gender.

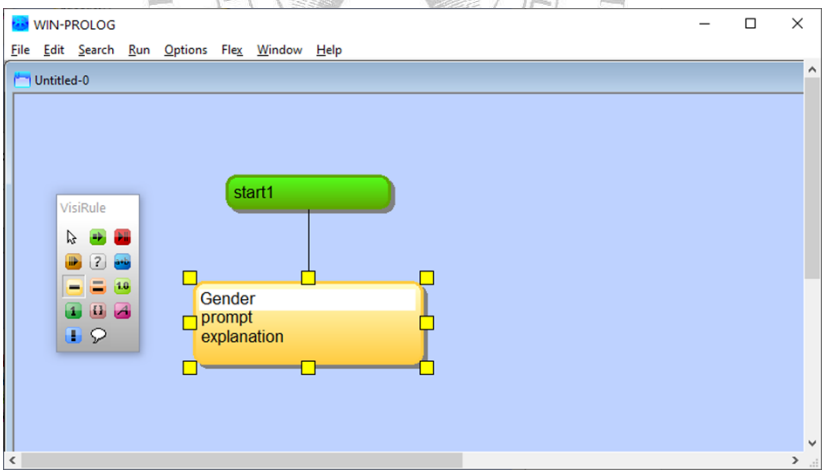



**Figure 10.7** *Changing from 'question 1' to 'Gender'*

So – we need to double click on the word ‘question1’ and type Gender. Now tab or click to the next field – it should auto select the content so some text can be typed in.

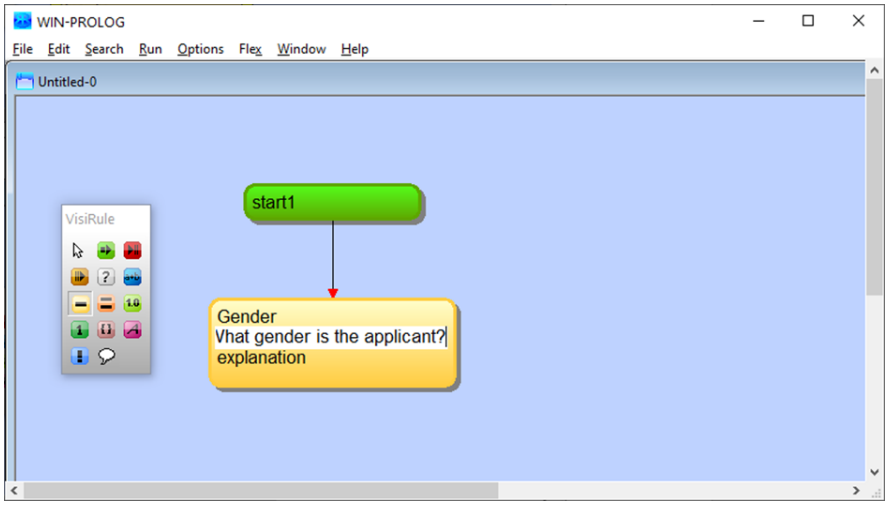



**Figure 10.8** *Writing question in the prompt field*

Now we are going to define 3 expressions: Male, Female and Neutral.

## Expressions

Expressions are defined using the white box with the ‘?’ mark.

VisiRule will inspect all of these and form ask able questions based on its understanding of the expected answers. This avoids double updating problems where you have to define (a) what’s in a question and (b) what happens with what answers. The order in which we do things is not significant.

Let's select the Expression box on the tool palette

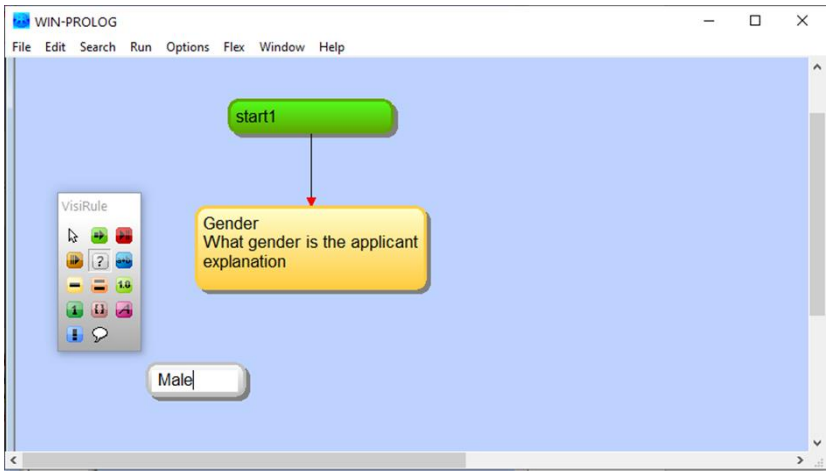



**Figure 10.9** *Updating expression box*

After creating and updating the expression box, connect the two boxes like shown below

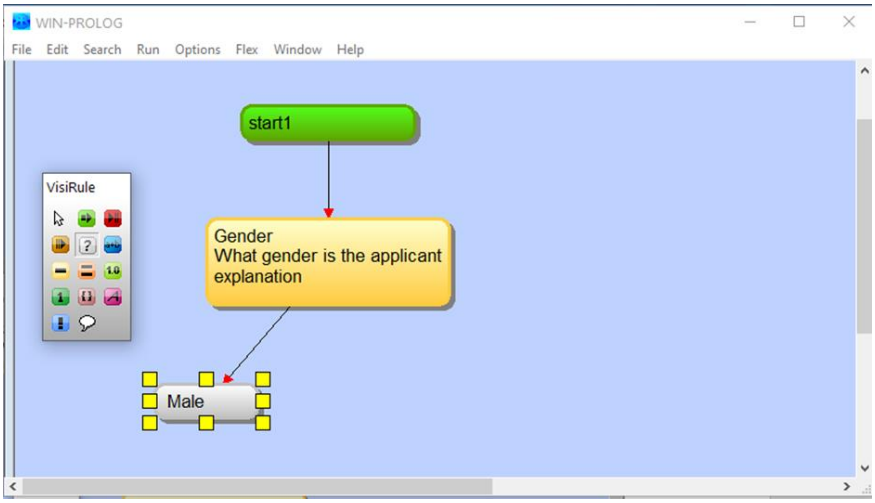



**Figure 10.10** *Connecting expression and question boxes together*

After creating other two expression boxes, the chart should look something like this

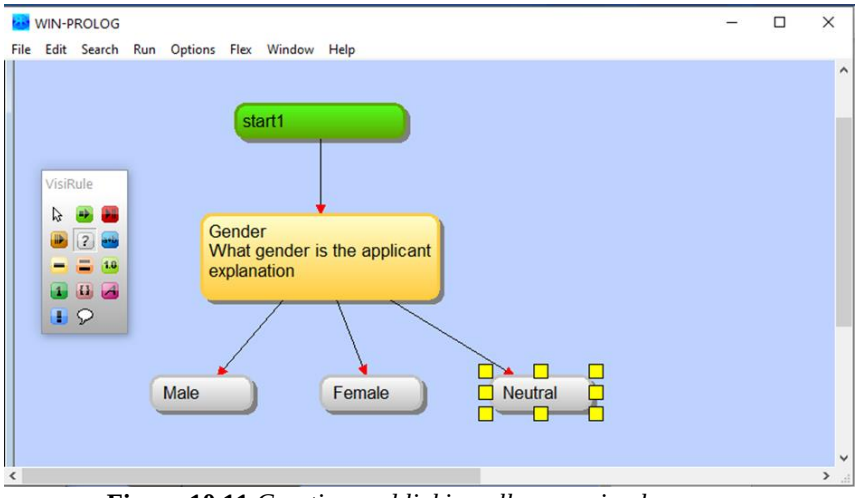



**Figure 10.11** *Creating and linking all expression boxes*

Now we will add some extra questions for each Branch.

To keep it simple we will ask Male to choose a book genre and Females to choose a Film genre and Neutral to choose a food type. Some selected screens are shown below.

Note that each box can be created and edited, then linked.

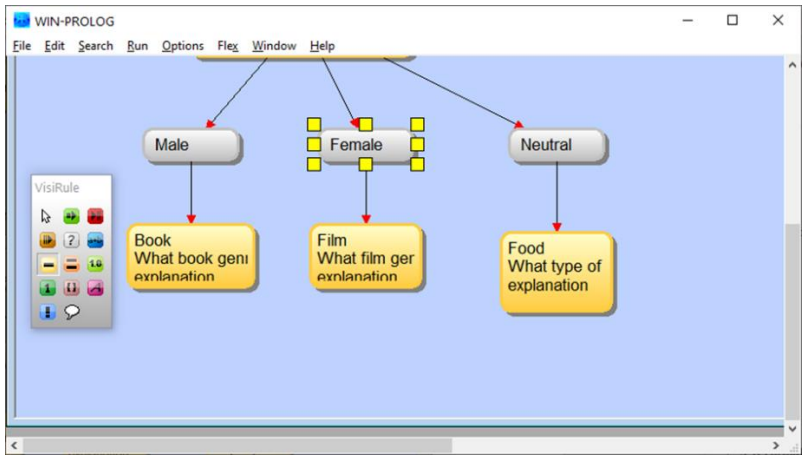



**Figure 10.12** *After adding another single choice question to each Branch*

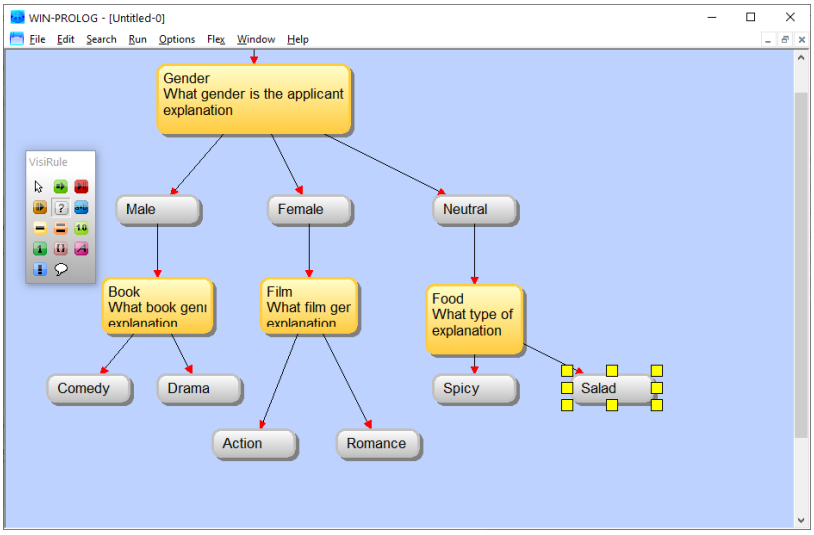



**Figure 10.13** *Connecting expression boxes to questions*

## Conclusion (End) Nodes

Now let's add some conclusions and that will complete the chart. For now, we will have 6 conclusions

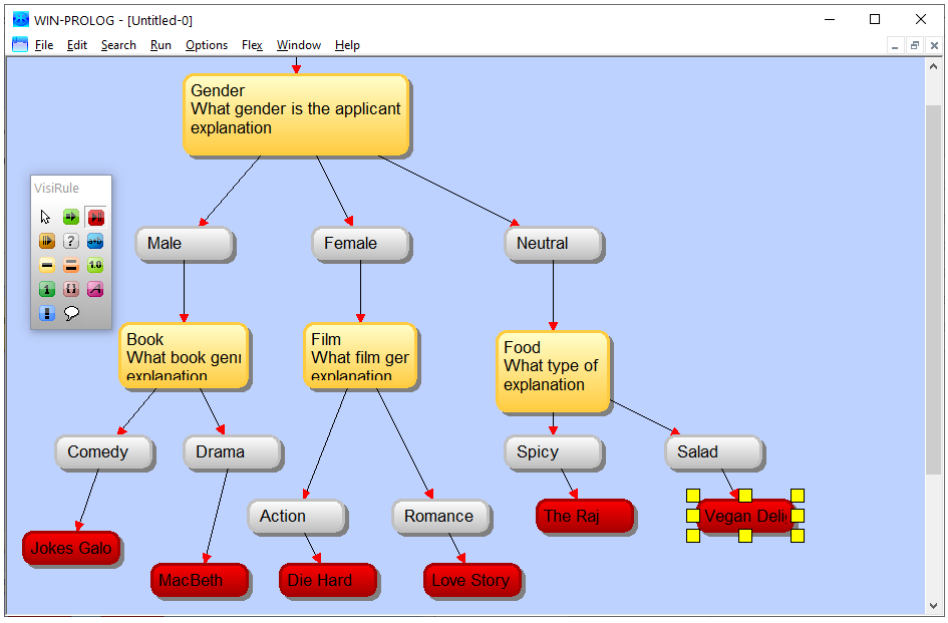



**Figure 10.14** *Updating and connecting End Nodes to end the graph*

After the chart is completed, save it so that it can be run. Before running this chart, we need to generate code for this chart. Right click on the canvas and select 'Show Code'

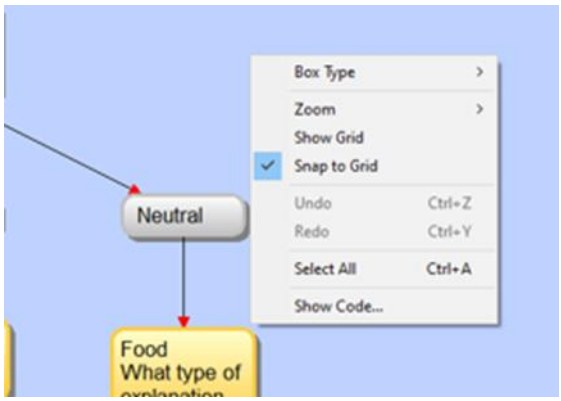



**Figure 10.15** *Generating code for the graph*

The Flex KSL rules have been generated and are in the flex code pane of the Generated code dialog. Click the Run button and these will be compiled into an executable Prolog format and run.

This will ask the first question:

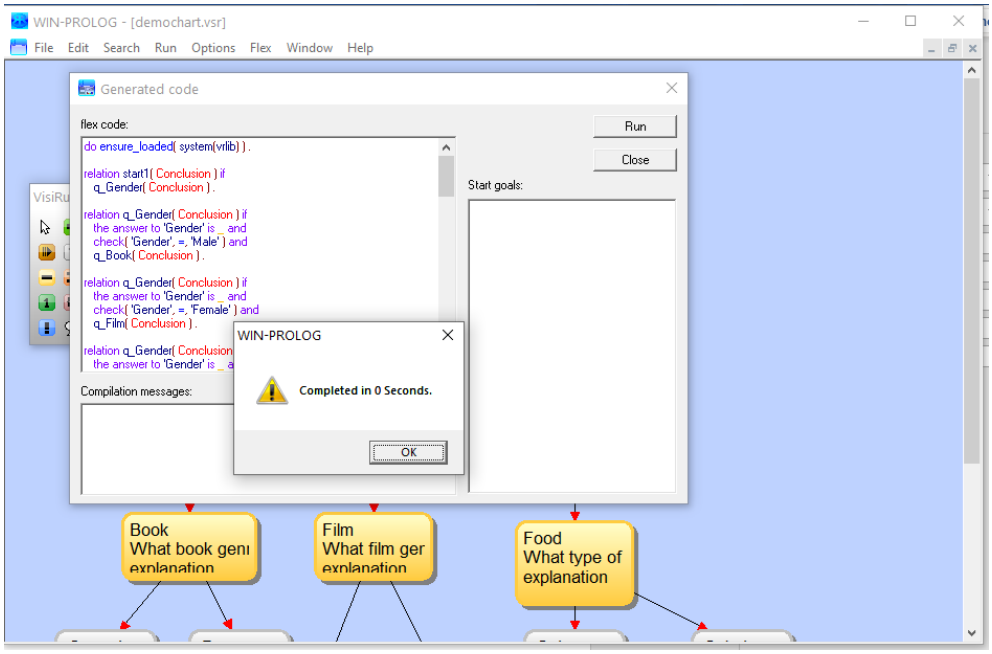



**Figure 10.16** *Generated Code*

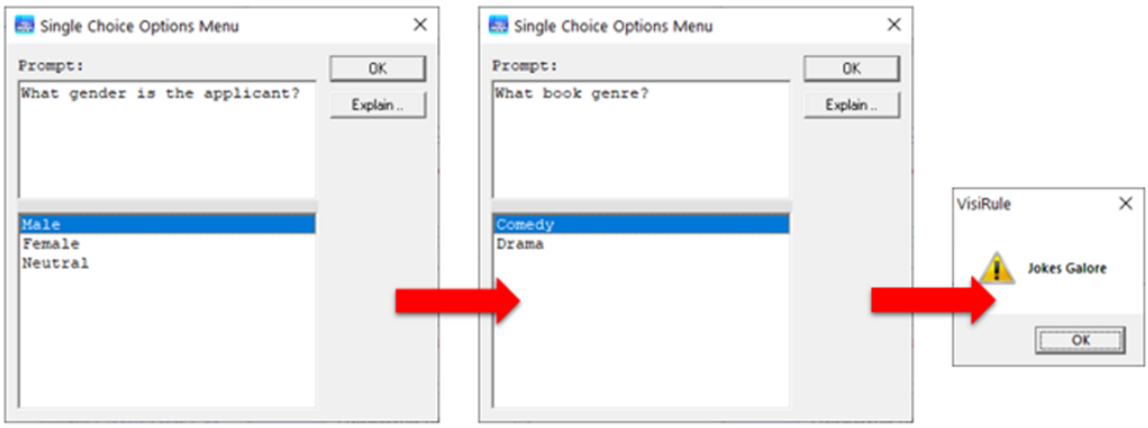



**Figure 10.17** *Sequence of questions if 'Male' branch is selected*

VisiRule can see if there are any other solutions available for that combination of answers. In this case,

none.


**Figure 10.18** *Checking for other solutions*

## One Expression or Two or …? Single-choice questions

Consider a simple question with 4 answers all leading to the same next node. We can put each answer into an expression on its own. OR because we have no branching - we can put all the options into a single expression box and have:

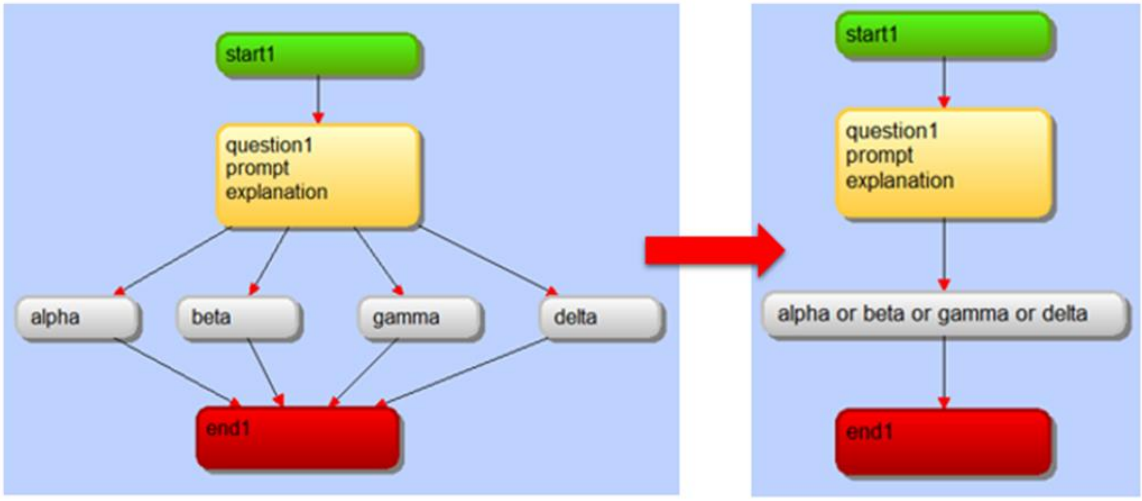



**Figure 10.19** *Merging Four Expression Boxes into One Leading to Same End Node*

Now we'll change the property of single answer question box to multiple answer question box.

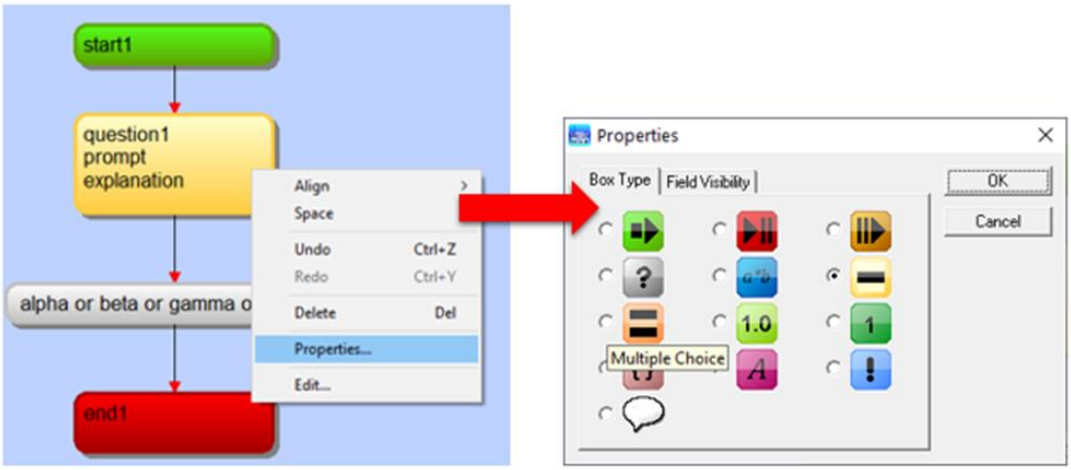



**Figure 10.20** *Changing box properties*

After changing the box properties, we can change the expression to a set and the 'or' to ','.

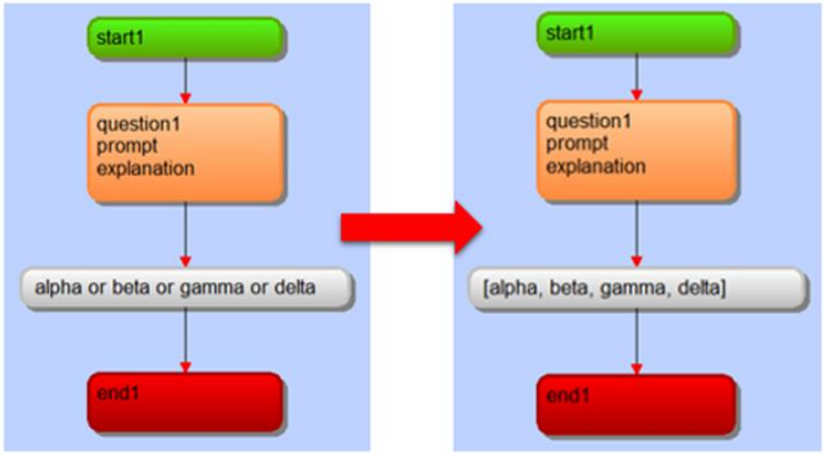



**Figure 10.21** *Changing box type and expression*

This will only succeed if all 4 are selected (as = is implied). We edit the expression so as to succeed on any selection.

The < operator has a special meaning on multi-choice questions.

? < [a,b,c] will succeed if the answer is 'less than' the set [a,b,c] i.e. a or b or c or ab or ac or bc PLUS
? = < [a,b,c] also includes abc

Note that we also add an explicit test that the answer contains something and then introduce a second expression to deal with the empty set separately.

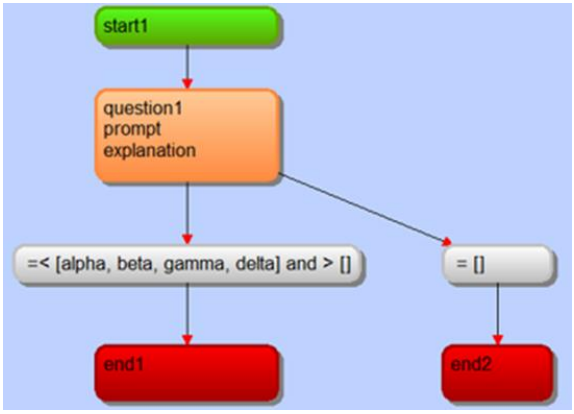



**Figure 10.22** *Separate branch added for none of the options selected*

The easiest way to output text and or partial results is to use flash/1 in a code box. Note the text to be output in the flash message should be quoted. This is because it contains a space. Here 'qusetion1' links to the multiple-choice question box. It should be quoted in single quotes to show the option chosen in the question.

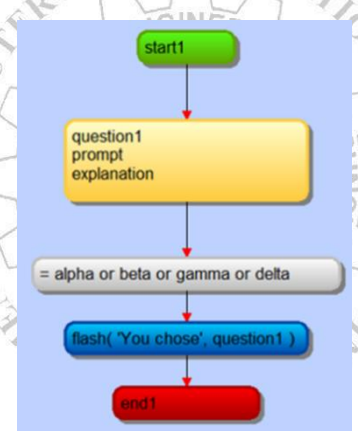



**Figure 10.23** *Inserting Code block to show partial results before the actual outcome*

## EXERCISES

1. Design a system which takes height and weight as number inputs and calculates the density according to the input taken. [Hint: Use number question, statement and code box types].

Attach screenshots here

2. Given below is a decision table for a chart "Should I go for a walk? A dilemma". Design an expert system according to the information given in the decision table.

| DECISION TABLE | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Weather | Hot + Dry | Hot + Windy | | | | | Cold + Wet | | | | |
| Physical Limitations? | - | Yes | No | | | | Yes | No | | | |
| Safe Time | - | - | Yes | | | No | - | Yes | | | No |
| Energy Levels | - | - | High | Moderate | Low | - | - | High | Moderate | Low | - |
| **Output/Decision** | Don't Go! | Don't Go! | go for walk | 15-20 min walk | Gentle stroll | Don't Go! | Don't Go! | go for walk | 15-20 min walk | Gentle stroll | Don't go! |

Attach screenshots here

# Lab Session 11

## *Constructing Complex Rule-Based Systems*

Having familiarized ourselves with basics of developing basic rule-based systems using VisiRule, let us move forward towards more complex systems. We will start with a simple problem with simple rules to draw conclusions, and add more complexity as we move forward.

### Company Loan Example

Let us consider the process of whether or not to grant an employee a company loan. Let us start with a simple set of rules as shown in Table 1. A simple decision table can be constructed from Table 1, as shown in Table 2 where each question is a row, and each column holds the various answers.

*Table 1: Simple set of Rules*

| **Simple Set of Rules** | | |
|---|---|---|
| If Full time = yes | and Over4yrs = yes | then answer = Grant Loan |
| If Full time = yes | and Over4yrs = no | then answer = Unclear |
| If Full time = no | and Over4yrs = yes | then answer = Unclear |
| If Full time = no | and Over4yrs = no | then answer = No Loan |

*Table 2: Initial Decision Table*

| **Initial Decision Table** | | | | |
|---|---|---|---|---|
| Full Time | yes | yes | no | no |
| Over4 years | yes | no | yes | no |
| | Grant Loan | Unclear | Unclear | No Loan |

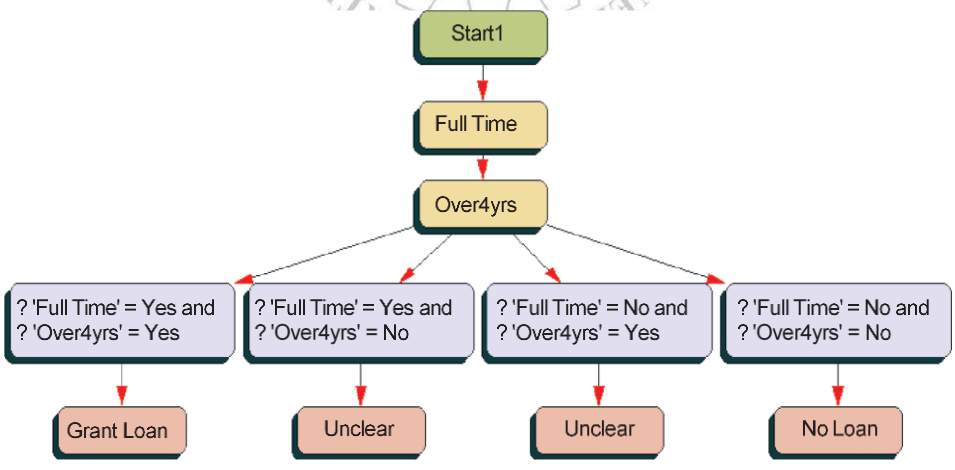



**Figure 11.1** *Initial chart with compound logic expressions*

We have two questions, each of which has two possible answers, there- fore four possible combinations. The questions, as they stand, are independent and can be asked in either order.
Using VisiRule, we can draw a simple chart, Fig. 11.1, where each expression box evaluates a compound logic expression referencing the two previously asked questions. We can make the logic simpler by adding in more boxes in other words the boxes containing more than one expression to be evaluated is now broken down as shown in Fig. 11.2, either method is applicable

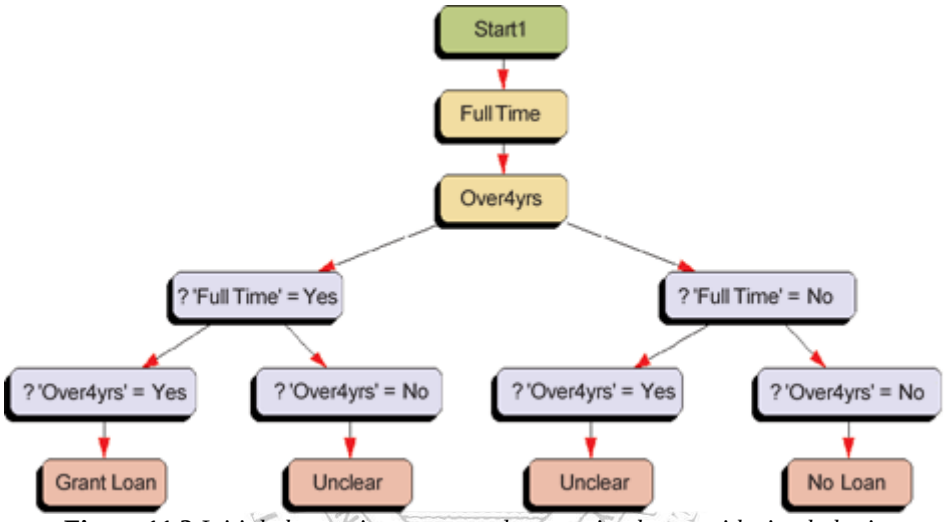



**Figure 11.2** *Initial chart using structured expression boxes with simple logic*

After setting up the initial system let's add more complexity. We can add one more question to be asked depending on the answer given in the first question. This imposes an ordering of the questions in Table 2 as shown in Table 3. We will also add another row to the decision table, Table 4.

*Table 3: Modified Rules with Split Questions*

| **Modified rules with split questions** | | |
|---|---|---|
| If Full time = yes | and Over3yrs = yes | then answer = Grant Loan |
| If Full time = yes | and Over3yrs = no | then answer = Unclear |
| If Full time = no | and Over5yrs = yes | then answer = Unclear |
| If Full time = no | and Over5yrs = no | then answer = No Loan |

*Table 4: Extended decision table with split questions*

| **Extended decision table with split questions** | | | | |
|---|---|---|---|---|
| Full Time | yes | yes | no | no |
| Over3 years | yes | no | - | - |
| Over5years | - | - | yes | no |
| | Grant Loan | Unclear | Unclear | No Loan |

Since Fig. 11.1 or Fig. 11.2 is no longer applicable to this scenario, we will transform it into Fig. 11.3. We need to do this because we would have to ask all three questions up front and then test the logic.

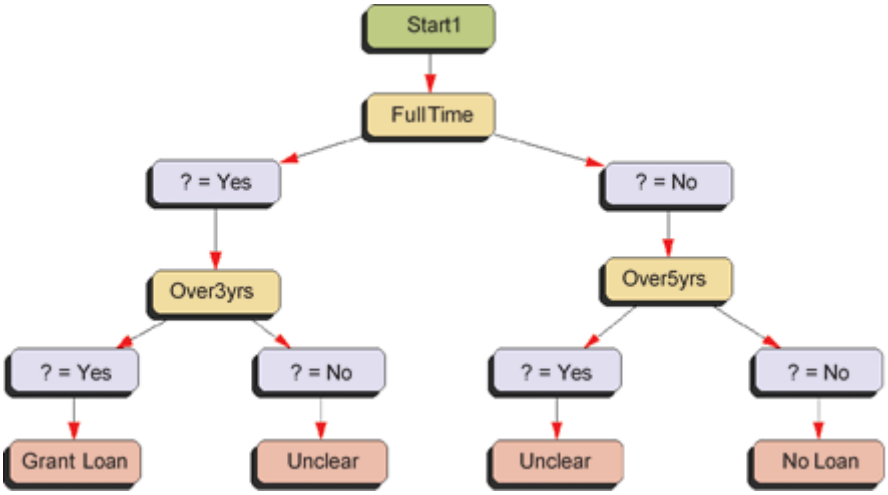



**Figure 11.3** *Extended chart with split questions*

Now, let's merge the two 'Unclear' solutions into a single track, call it 'Check_previous', and then continue it. This frequently occurs; two positive responses produce an approval, two negatives, a rejection, but mixed answers need further exploration. For this purpose, let's extend our rules and modify Table 3 to Table 5 and then define some continuation rules for the intermediate conclusion, 'Check_previous' as shown in Table 6. After setting up new rules the decision table gets extended as shown in Table 7.

*Table 5: Merged rules referencing previous loan*

| **Merged rules referencing previous loan** | | |
|---|---|---|
| If Full time = yes | and Over3yrs = yes | then answer = Grant Loan |
| If Full time = yes | and Over3yrs = no | then answer = Check_previous |
| If Full time = no | and Over5yrs = yes | then answer = Check_previous |
| If Full time = no | and Over5yrs = no | then answer = No Loan |

*Table 6: New rules relating to previous loan*

| **New rules relating to previous loan** | |
|---|---|
| If Check_previous and Previous_loan = yes | then answer = Unclear |
| If Check_previous and Previous_loan = no | then answer = No Loan |

*Table 7: Question about previous loan added to decision table*

| **Question about previous loan added to decision table** | | | | | | |
|---|---|---|---|---|---|---|
| Full Time | yes | yes | yes | no | no | no |
| Over3 years | yes | no | no | - | - | - |
| Over5years | - | - | - | yes | yes | no |
| Previous_loan | - | yes | no | yes | no | - |
| | Grant Loan | Unclear | No Loan | Unclear | No Loan | No Loan |

In our chart, we can simply join the branches represented by the two 'Unclear' nodes in the third chart in Fig. 11.3 and link that merged node into our new question, 'Previous_loan', as in Fig. 11.4.

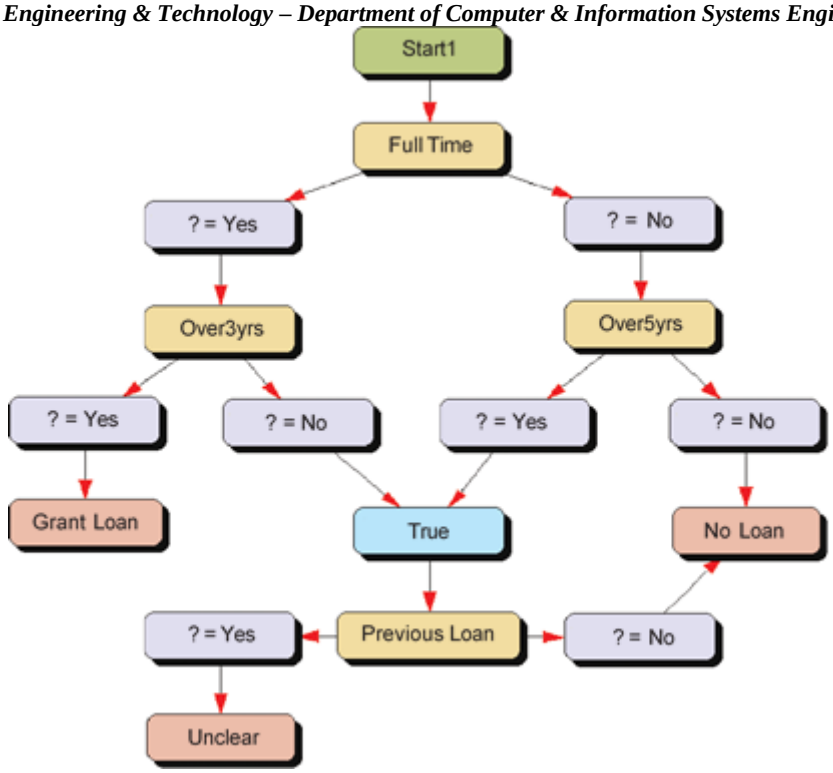



**Figure 11.4** *Merged paths lead to previous loan*

Finally, let us resolve the remaining 'Unclear solution'. We continue the "yes" branch to 'Previous_loan' and leave the "no" branch to still give, No Loan. Replacing 'Unclear' with 'Previous_loan' results in a slight change to the rules previously shown in Table 6 to give a new set of rules as shown in Table 8. The continuation rules for the intermediate conclusion, 'Check_repaid', are simple as shown in Table 9.

*Table 8: Modified rules relating to previous loan*

| **Modified rules relating to previous loan** | |
|---|---|
| If Check_previous and Previous_loan = yes | then answer = Check_repaid |
| If Check_previous and Previous_loan = no | then answer = No Loan |

*Table 9: New rules relating to loan repayment*

| **New rules relating to loan repayment** | |
|---|---|
| If Check_repaid and Repaid_on_time = yes | then answer = Grant Loan |
| If Check_repaid and Repaid_on_time = no | then answer = No Loan |

Now, we have only two outcomes and five binary questions; users will be asked two, three, or four depending on their responses. The corresponding decision table with the new question, Repaid_on_time, is shown in Table 10.

*Table 10: Question about loan repayment added to decision table*

| **Question about loan repayment added to decision table** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Full Time | yes | yes | yes | yes | no | no | no | no |
| Over3 years | yes | no | no | no | - | - | - | - |
| Over5years | - | - | - | - | yes | yes | yes | no |
| Previous_loan | - | yes | yes | no | yes | yes | no | - |
| Repaid_on_time | - | yes | no | - | yes | no | - | - |
| | Grant Loan | Grant Loan | No Loan | No Loan | Grant Loan | No Loan | No Loan | No Loan |

We develop the final rule-based system as shown in Fig. 11.5 by linking the "no" answer to Previous_loan back to the box marked "No Loan" and continuing on with the "yes" branch through to Repaid_on_time.


**Figure 11.5** *Merged paths for repaid loans*

## EXERCISES:

1. Develop a rule-based system used by a personnel department to establish holiday entitlement which states:

- The number of holidays depends on age and years of service. Every employee receives at least 22 days
- Additional days are provided according to the following criteria: Only employees who are younger than 18 or at least 60 years old, or employees with at least 30 years of service will receive five extra days.
- If the employee has at least 15 but less than 30 years of service, two extra days are given. These two days are also provided for employees who are 45 or older. The two extra days cannot be combined with the five extra days.
- Employees with at least 30 years of service and also employees aged 60 or more, receive three extra days, on top of the possible additional days already supplied

Attach screenshots here

2. Develop a knowledge-based system based on the following rules:

- The medical diagnosis depends upon user's selection of symptoms as is presented to him at the start
- If the user has both dry cough and general malaise but no fever then he may have Influenza.
- If the user has both dry cough and general malaise and he also chooses fever, then he may have Laryngitis
- If the user suffers from running nose and sneezing, ask if the user has any allergies.
    - If the user has allergies, he may have hay fever
    - if he doesn't have allergies, he may have common cold
- If the user is both breathless and wheezing, he may have asthma
- For any other combination the system should answer that it can't diagnose
- If the user selects nothing out of the options given, then system should print 'Guess you are ok'

Attach screenshots here

# Lab Session 12

## *Practicing Data-Driven programming in Expert Systems*

## Data Driven Programming

The frame system of *flex* can be used for the representation of data and knowledge. In this chapter we describe data-driven programming, where rather than program the control-flow or logic of a system, procedures are attached to individual frames or groups of frames. These procedures lie dormant and are activated whenever there is a request to update, access or create an instance of the frame or slot to which they are attached. This concept of attached procedures, sometimes referred to as *active value*s, is also found in object-oriented programming.

## Data-Driven Procedures

There are four types of data-driven procedures:

- Launches
- Watchdogs
- Constraints
- Demons

The following diagram shows when and where the various procedures are invoked.

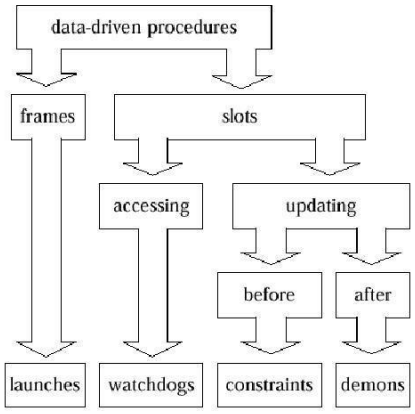



**Figure 12.1** *Data driven procedures*

## Launches

A **`launch`** procedure is activated whenever there is a request to create an instance of the frame to which it is attached.

A **`launch`** procedure has three main parts:

- context A test to see whether certain conditions hold.
- test A test to see whether certain conditions hold.

- action A series of commands to be performed.

The action will only be invoked if the test succeeds. The launch is invoked *after* the instance has been created.

## *Example:*

Suppose we have a frame system representing a company's personnel structure, and that a new employee Dave is to be added. The launch will automatically be invoked whenever a new instance of employee is requested, and provided the conditions hold, the actions will be performed.

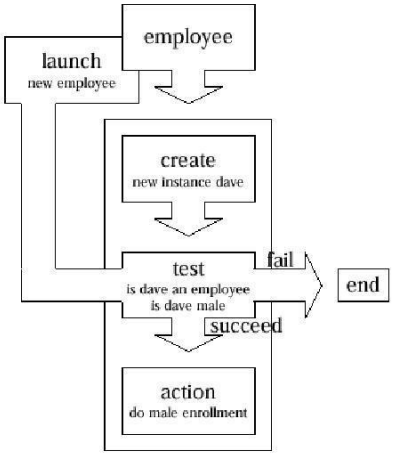



**Figure 12.2** *Employee Example*

The launch procedure in this example, attached to the employee frame, is set up to collect the personal details about new male employees.
The KSL code for this example would be written as follows:

```
frame employee
   default sex is male .

launch new_employee
   when Person is a new employee
   and sex of Person is male
   then male_enrolment_questions(Person) .

instance dave is an employee .
```

The `male_enrolment_questions` will be defined elsewhere.

## Constraining the Values of Slots

A **`constraint`** is attached to an individual slot, and is designed to constrain the contents of a slot to valid values. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *before* the update.

A **`constraint`** has three main parts:

- context A test to see whether certain conditions hold.
- check A test to see whether the update is valid.

- error A series of commands to be performed for an invalid update.

The check will only be made if the context holds. If the check is successful then the update is allowed, otherwise the error commands are invoked and the update is not allowed.

## *Example:*

If we had a frame system representing instances of water containers, we could put a constraint on the contents slot of any jug, such that when the value for the particular slot is being updated, a test is performed making sure that the new value is less than the value of the jug`s capacity, thus ensuring the jug does not overflow!

In the example the constraint is activated if the contents attribute of any jug changes. The prospective new value for the slot is then tested to see if it is less than that jug's capacity. If the test succeeds the update is allowed. If the test fails the update is not allowed and a message is displayed.

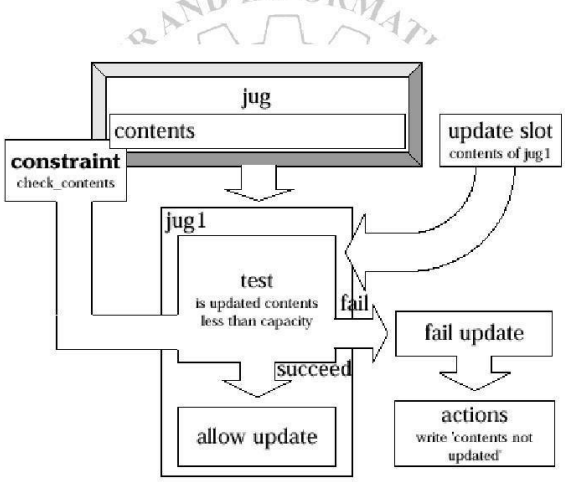



**Figure 12.3** *Jug Example*

The code for this example could be written as follows:

```
frame jug
   default contents are 0 and
   default capacity is 7 .

instance jug1 is a jug .

constraint check_contents
   when the contents of Jug changes to X
   and Jug is some jug
   then check that number( X )
   and X =< Jug`s capacity
   otherwise write( 'contents not updated' )
   and nl .
```

Note the use of `Jug`, a local variable, which will unify with any frame or instance which has, in this case, a `contents` attribute.

## Attaching Demons to Slot Updates

A `demon` is attached to an individual slot. It is activated whenever the *current* value of that slot is updated, and the activation occurs immediately *after* the update.

A **demon** has two main parts:

- context A test to see whether certain conditions hold.
- action A series of commands to be performed.

The slot is updated and then, given that the context holds, the actions will be invoked.

A **demon** can be tailored such that it fires only for given values and/or only under certain circumstances.

## *Example:*

If we are modelling the action of kettles, we could attach a demon to the `temp` slot of any instance of the frame `kettle`.

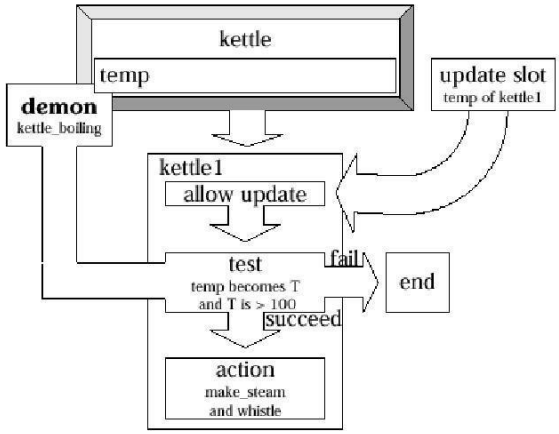



**Figure 12.4** *Kettle Example*

Whenever the temp slot is updated, a check will be made on the new value, such that, if it is greater than 100, then the actions make_steam and whistle are performed.

The code for this example could be written as follows:

```
frame kettle
   default temp is 0 .

instance kettle1 is a kettle .

demon kettle_boiling

   when the temp changes to T
   and T is greater than 100
   then make_steam
   and nl
   and whistle .
```

## Restricting the Access to Slots

A **watchdog** is attached to an individual slot. It is activated whenever the *current* value of the slot is accessed.

A watchdog has three main parts:

- context A test to see whether certain conditions hold.
- check A test to see whether the access is valid.
- error A series of commands to be performed if access is invalid.

The check will only be made if the context holds. If the check is successful then the access is allowed. Otherwise, the error commands are invoked and the access is denied.

A watchdog can be used to check the access rights to an attribute of a frame. It is invoked whenever there is a request for the *current* value *(not* the default value) of that slot (attribute-frame pair).

## *Example:*

In the example shown below, the watchdog is activated when the contents of a file are requested. A check on the user's classification is then made, and if the check succeeds the access is allowed. If the check fails the access is denied and a warning message is displayed.

The KSL for a similar example is shown below. In this case, only users with sufficiently high access priority may find out the balance in a bank account.

The current_user frame stores the access code of the current user, which is checked in the account_security watchdog.

```
frame bank_account
   default balance is 0.

frame current_user default
   name is '' and default
   access is 0.

watchdog account_security
   when the balance of Account is requested
   and Account is some bank_account
   then check current_user`s access is above 99 otherwise
   write( 'Balance access denied to user ' ) and
   write( current_user`s name ).
```

## Exercises

1. Define a frame **Kettle** and make use of launch procedure to create some instances.

2. Write a program which checks different conditions and permits/restricts user from accessing the attribute values. Use the scenario of a Bank Account

# Lab Session 13

## *Developing Fuzzy Logic Based System*

## Introduction

Fuzzy Logic Toolbox™ software is a collection of functions built on the MATLAB® technical computing environment. It provides tools for you to create and edit fuzzy inference systems within the framework of MATLAB. This toolbox relies heavily on graphical user interface (GUI) tools to help you accomplish your work, although you can work entirely from the command line if you prefer. In this lab session, we will use GUI for building fuzzy logic based application.

## Fuzzy Inference Systems

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. The mapping then provides a basis from which decisions can be made, or patterns discerned. The process of fuzzy inference involves ―*Membership Functions*‖, ―*Logical Operations*‖, and ―*If-Then Rules*‖. Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Because of its multidisciplinary nature, fuzzy inference systems are associated with a number of names, such as fuzzy-rule-based systems, fuzzy expert systems, fuzzy modeling, fuzzy associative memory, fuzzy logic controllers, and simply (and ambiguously) fuzzy systems.

You can implement two types of fuzzy inference systems in the toolbox: *Mamdani-type* and *Sugeno-type*. These two types of inference systems vary somewhat in the way outputs are determined.

Mamdani-type inference, as defined for the toolbox, expects the output membership functions to be fuzzy sets. After the aggregation process there is a fuzzy set for each output variable that needs defuzzification. It is possible, and in many cases much more efficient, to use a single spike as the output membership function rather than a distributed fuzzy set. This type of output is sometimes known as a *singleton* output membership function, and it can be thought of as a pre-defuzzified fuzzy set. It enhances the efficiency of the defuzzification process because it greatly simplifies the computation required by the more general Mamdani method, which finds the centroid of a two-dimensional function. Rather than integrating across the two-dimensional function to find the centroid, you use the weighted average of a few data points. Sugeno-type systems support this type of model. In general, Sugeno-type systems can be used to model any inference system in which the output membership functions are either linear or constant.

## Overview of Fuzzy Inference Process

We‘ll start with a basic description of a two-input, one-output tipping problem (based on tipping practices in the United States).

## The Basic Tipping Problem

Given a number between 0 and 10 that represents the quality of service at a restaurant (where 10 is excellent), and another number between 0 and 10 that represents the quality of the food at that restaurant (again, 10 is excellent), what should the tip be?

The starting point is to write down the three golden rules of tipping.

1. *If the service is poor or the food is rancid, then tip is cheap.*
2. *If the service is good, then tip is average.*
3. *If the service is excellent or the food is delicious, then tip is generous.*

Assume that an average tip is 15%, a generous tip is 25%, and a cheap tip is 5%.

Now that you know the rules and have an idea of what the output should look like, begin working with the GUI tools to construct a fuzzy inference system for this decision process.

## The FIS Editor

The FIS Editor displays information about a fuzzy inference system. To open the FIS Editor, type the following command at the MATLAB prompt: `fuzzy`

The generic untitled FIS Editor opens, with one input labeled `input1`, and one output labeled `output1`.

In the given example, you construct a two-input, one output system. The two inputs are **`service`** and **`food`**. The one output is **`tip`**. To add a second input variable and change the variable names to reflect these designations:

**1.** Select **Edit > Add variable > Input**.
A second yellow box labeled **input2** appears.
**2.** Click the yellow box **input1**. This box is highlighted with a red outline.
**3.** Edit the **Name** field from input1 to service, and press **Enter**.
**4.** Click the yellow box **input2**. This box is highlighted with a red outline.
**5.** Edit the **Name** field from input2 to food, and press **Enter**.
**6.** Click the blue box **output1**.
**7.** Edit the **Name** field from output1 to tip, and press **Enter**.
**8.** Select **File > Export > To Workspace**.
**9.** Enter the **Workspace variable** name tipper, and click **OK**.

The diagram is updated to reflect the new names of the input and output variables. There is now a new variable in the workspace called tipper that contains all the information about this system. By saving to the workspace with a new name, you also rename the entire system. Your window looks something like the following diagram.

Leave the inference options in the lower left in their default positions for now. You have entered all the information you need for this particular GUI. Next, define the membership functions associated with each of the variables. To do this, open the Membership Function Editor. You can open the Membership Function Editor in one of three ways:

- Within the FIS Editor window, select **Edit > Membership Functions.**
- Within the FIS Editor window, double-click the blue icon called **tip**.
- At the command line, type `mfedit`.

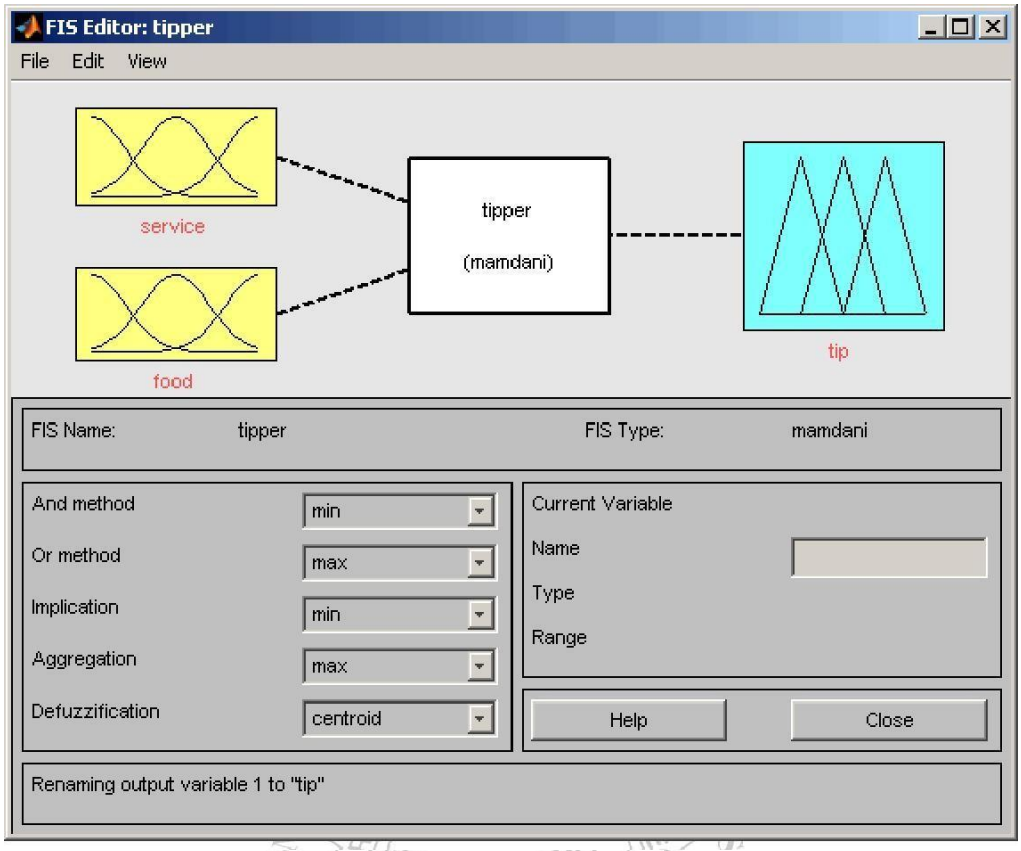



**Figure 13.1** *Fuzzy Toolbox Window*

## The Membership Function Editor

The Membership Function Editor is the tool that lets you display and edit all of the membership functions associated with all of the input and output variables for the entire fuzzy inference system. The Membership Function Editor shares some features with the FIS Editor, as shown in *Figure 2*. In fact, all of the five basic GUI tools have similar menu options, status lines, and **Help** and **Close** buttons.

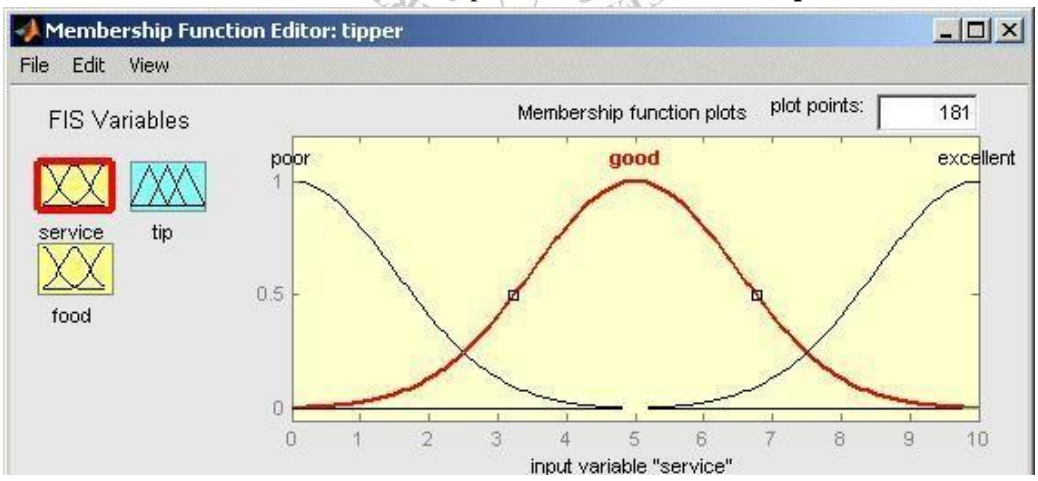



**Figure 13.2** *Output Window*

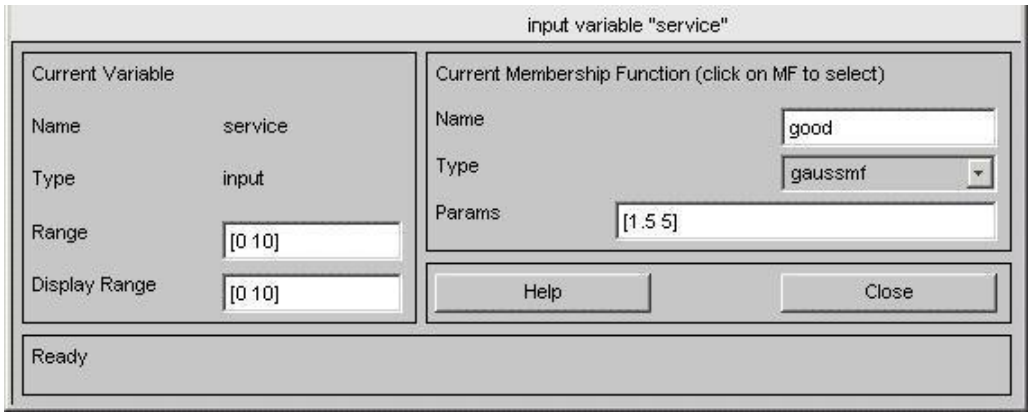



**Figure 13.3** *Configuration Window*

When you open the Membership Function Editor to work on a fuzzy inference system that does not already exist in the workspace, there is no membership function associated with the variables that you defined with the FIS Editor. On the upper-left side of the graph area in the

Membership Function Editor is a ―Variable Palette‖ that lets you set the membership functions for a given variable. To set up the membership functions associated with an input or an output variable for the FIS, select a FIS variable in this region by clicking it. Next select the **Edit** pull-down menu, and choose **Add MFs...** A new window appears which allows you to select both the membership function type and the number of membership functions associated with the selected variable.

In the lower-right corner of the window are the controls that let you change the name, type, and parameters (shape), of the membership function, after it is selected.

The membership functions from the current variable are displayed in the main graph. These membership functions can be manipulated in two ways. You can first use the mouse to select a particular membership function associated with a given variable quality, (such as poor, for the variable, service), and then drag the membership function from side to side.

The process of specifying the membership functions for the two input tipping example, tipper, is as follows:

**1.** Double-click the input variable service to open the Membership Function Editor.
**2.** In the Membership Function Editor, enter [0 10] in the **Range** and the **Display Range** fields.
**3.** Create membership functions for the input variable service.
   **a.** Select **Edit > Remove All MFs** to remove the default membership functions for the input variable service.
   **b.** Select **Edit > Add MFs.** to open the Membership Functions dialog box.
   **c.** In the Membership Functions dialog box, select gaussmf as the **MF Type**.
   **d.** Verify that 3 is selected as the **Number of MFs**.
   **e.** Click **OK** to add three Gaussian curves to the input variable service.
**4.** Rename the membership functions for the input variable service, and specify their parameters.
   **a.** Click on the curve named mf1 to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
      • In the **Name** field, enter poor.
      • In the **Params** field, enter [1.5 0].

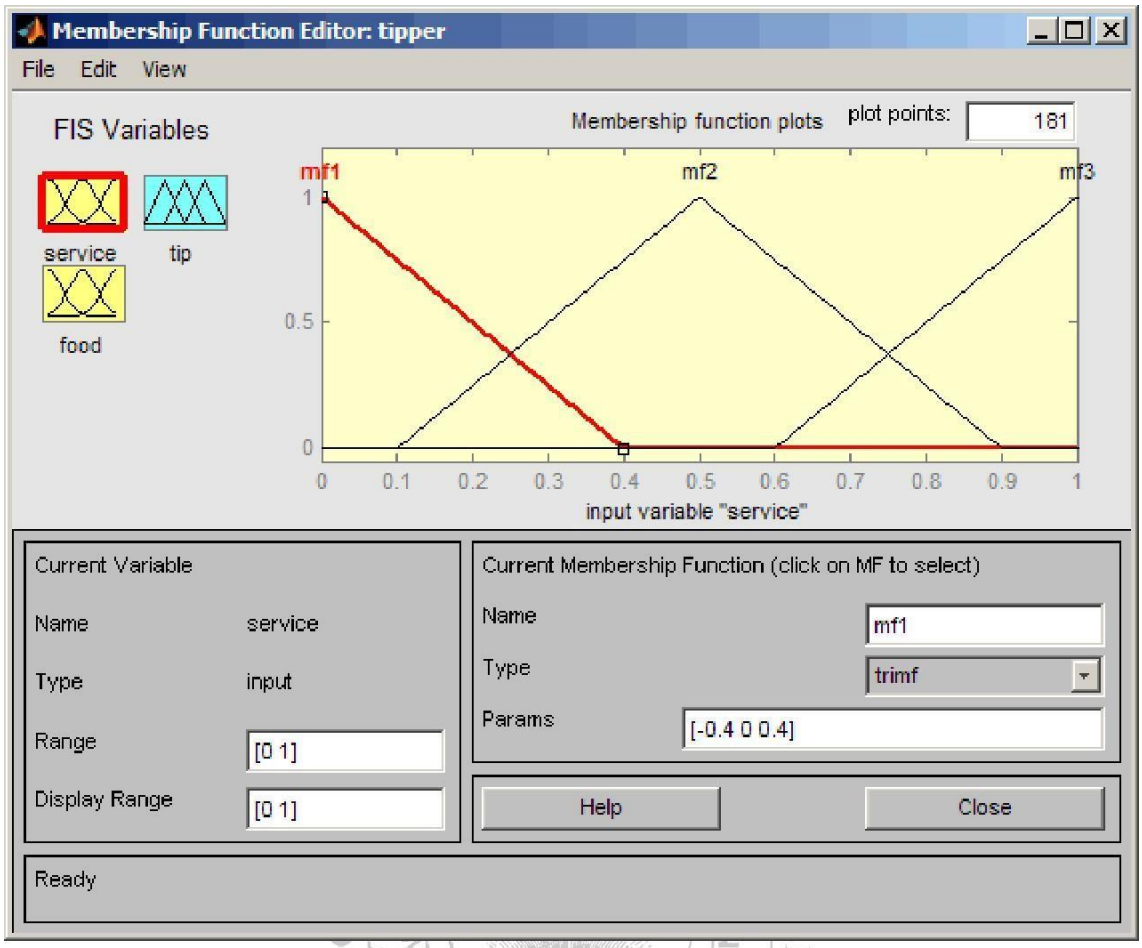



**Figure 13.4** *Membership function window*

The two inputs of **Params** represent the standard deviation and center for the Gaussian curve. **Tip** to adjust the shape of the membership function, type in desired parameters.

**b.** Click on the curve named mf2 to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `good`.
- In the **Params** field, enter `[1.5 5]`.

**c.** Click on the curve named **mf3**, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `excellent`.
- In the **Params** field, enter `[1.5 10]`.

**5.** In the **FIS Variables** area, click the input variable `food` to select it.

**6.** Enter `[0 10]` in the **Range** and the **Display Range** fields.

**7.** Create the membership functions for the input variable `food`.

**a.** Select **Edit > Remove All MFs** to remove the default Membership Functions for the input variable `food`.

**b.** Select **Edit > Add MFs** to open the Membership Functions dialog box.

**c.** In the Membership Functions dialog box, select `trapmf` as the **MF Type**.

**d.** Select 2 in the **Number of MFs** drop-down list.

**e.** Click **OK** to add two trapezoidal curves to the input variable `food`.

**8.** Rename the membership functions for the input variable food, and specify their parameters:

**a.** In the **FIS Variables** area, click the input variable food to select it.

**b.** Click on the curve named **mf1**, and specify the following fields in the **Current Membership Function (click on MF to select)** area:

- In the **Name** field, enter `rancid`.
- In the **Params** field, enter `[0 0 1 3]`.

**c.** Click on the curve named **mf2** to select it, and enter `delicious` in the **Name** field.
**d.** Reset the associated parameters if desired.

**9.** Click on the output variable `tip` to select it.

**10.** Enter `[0 30]` in the **Range** and the **Display Range** fields to cover the output range. The input ranges from 0 to 10, but the output is a `tip` between 5% and 25%.

**11.** Rename the default triangular membership functions for the output variable `tip`, and specify their parameters.

**a.** Click the curve named **mf1** to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `cheap`.
- In the **Params** field, enter `[0 5 10]`.

**b.** Click the curve named **mf2** to select it, and specify the following fields in the **Current Membership Function (click on MF to select)** area:
- In the **Name** field, enter `average`.
- In the **Params** field, enter `[10 15 20]`.

**c.** Click the curve named **mf3** to select it, and specify the following:
- In the **Name** field, enter `generous`.
- In the **Params** field, enter `[20 25 30]`.

Now that the variables have been named and the membership functions have appropriate shapes and names, you can enter the rules. To call up the Rule Editor, go to the **Edit** menu and select **Rules**, or type `ruleedit` at the command line.

Constructing rules using the graphical Rule Editor interface is fairly self evident. Based on the descriptions of the input and output variables defined with the FIS Editor, the Rule Editor allows you to construct the rule statements automatically, From the GUI, you can create rules by selecting an item in each input and output variable box, selecting one **Connection** item, and clicking **Add Rule**. You can choose none as one of the variable qualities to exclude that variable from a given rule and choose not under any variable name to negate the associated quality.

- Delete a rule by selecting the rule and clicking **Delete Rule**.
- Edit a rule by changing the selection in the variable box and clicking **Change Rule**.
- Specify weight to a rule by typing in a desired number between 0 and 1 in **Weight**. If you do not specify the weight, it is assumed to be unity (1).

To insert the first rule in the Rule Editor, select the following:

- `poor` under the variable **service**
- `rancid` under the variable **food**
- The **or** radio button, in the **Connection** block
- `cheap`, under the output variable, **tip**.

Then, click **Add rule**.

The resulting rule is
*1. If (service is poor) or (food is rancid) then (tip is cheap) (1)*

The numbers in the parentheses represent weights.

Follow a similar procedure to insert the second and third rules in the Rule Editor to get:

***1.*** *If (service is poor) or (food is rancid) then (tip is cheap) (1)*
***2.*** *If (service is good) then (tip is average) (1)*
***3.*** *If (service is excellent) or (food is delicious) then (tip is generous) (1)*

At this point, the fuzzy inference system has been completely defined, in that the variables, membership

functions, and the rules necessary to calculate tips are in place. Now, look at the fuzzy inference diagram presented at the end of the previous section and verify that everything is behaving the way you think it should. You can use the Rule Viewer, the next of the GUI tools we'll look at. From the **View** menu, select **Rules**.

## The Rule Viewer

The Rule Viewer displays a roadmap of the whole fuzzy inference process. It is based on the fuzzy inference diagram described in the previous section. You see a single figure window *(Figure 5)* with 10 plots nested in it. The three plots across the top of the figure represent the antecedent and consequent of the first rule. Each rule is a row of plots, and each column is a variable. The rule numbers are displayed on the left of each row. You can click on a rule number to view the rule in the status line.

- The first two columns of plots (the six yellow plots) show the membership functions referenced by the antecedent, or the if-part of each rule.
- The third column of plots (the three blue plots) shows the membership functions referenced by the consequent, or the then-part of each rule.

Notice that under **food**, there is a plot which is blank. This corresponds to the characterization of none for the variable **food** in the second rule.

- The fourth plot in the third column of plots represents the aggregate weighted decision for the given inference system. This decision will depend on the input values for the system. The **defuzzified** output is displayed as a bold vertical line on this plot. The variables and their current values are displayed on top of the columns. In the lower left, there is a text field **Input** in which you can enter specific input values. For the two-input system, you will enter an input vector, `[9 8]`, for example, and then press **Enter**. You can also adjust these input values by clicking on any of the three plots for each input. This will move the red index line horizontally, to the point where you have clicked. Alternatively, you can also click and drag this line in order to change the input values. When you release the line, (or after manually specifying the input), a new calculation is performed, and you can see the whole fuzzy inference process take place.

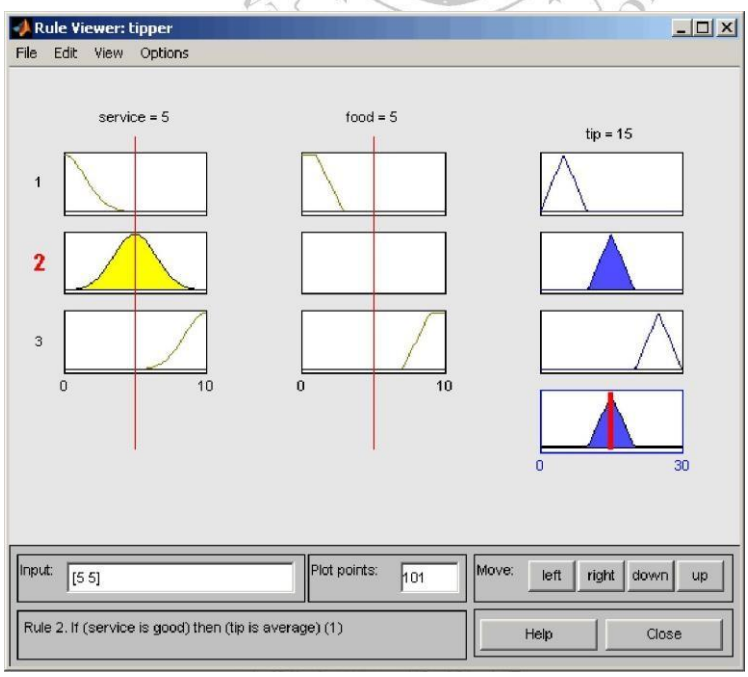



**Figure 13.5** *Final Selection Window*

- Where the index line representing service crosses the membership function line —`service is`

`poor` || in the upper-left plot determines the degree to which rule one is activated.

- A yellow patch of color under the actual membership function curve is used to make the fuzzy membership value visually apparent. Each of the characterizations of each of the variables is specified with respect to the input index line in this manner. If you follow rule 1 across the top of the diagram, you can see the consequent —`tip is cheap` || has been truncated to exactly the same degree as the (composite) antecedent—this is the implication process in action. The aggregation occurs down the third column, and the resultant aggregate plot is shown in the single plot appearing in the lower right corner of the plot field. The **defuzzified** output value is shown by the thick line passing through the aggregate fuzzy set. You can shift the plots using **left**, **right**, **down**, and **up**. The menu items allow you to save, open, or edit a fuzzy system using any of the five basic GUI tools.

The Rule Viewer allows you to interpret the entire fuzzy inference process at once. The Rule Viewer also shows how the shape of certain membership functions influences the overall result. Because it plots every part of every rule, it can become unwieldy for particularly large systems, but, for a relatively small number of inputs and outputs, it performs well (depending on how much screen space you devote to it) with up to 30 rules and as many as 6 or 7 variables.

When you save a fuzzy system to a file, you are saving an ASCII text FIS file representation of that system with the file suffix **.fis**. This text file can be edited and modified and is simple to understand. When you save your fuzzy system to the MATLAB workspace, you are creating a variable (whose name you choose) that acts as a MATLAB structure for the FIS system. FIS files and FIS structures represent the same system.

If you do not save your FIS to a file, but only save it to the MATLAB workspace, you cannot recover it for use in a new MATLAB session.

## Exercises

1) Implement the following rule set in matlab, to control the mechanism of a crane.

   - IF Distance is far AND Angle is zero THEN apply pos_medium Power
   - IF Distance is far AND Angle is neg_small THEN apply pos_high Power
   - IF Distance is medium AND Angle is neg_small THEN apply pos_high Power
   - IF Distance is medium AND Angle is neg_big THEN apply pos_medium Power
   - IF Distance is close AND Angle is pos_small THEN apply neg_medium Power
   - IF Distance is close AND Angle is neg_small THEN apply pos_medium Power
   - IF Distance is close AND Angle is zero THEN apply zero Power
   - IF Distance is zero AND Angle is zero THEN apply zero Power
   - IF Distance is zero AND Angle is pos_small THEN apply neg_medium Power
   - Develop a fuzzy logic based application of your choice.

2) Attach the screenshots of FIS editor, membership functions of all input & output parameters, rule editor and rule viewer.

3) Implement the following rule set in matlab, to control the mechanism of a fuzzy logic based washing machine.

   - If clothe material is soft and status is clean then apply low power for less cycle time.
   - If clothe material is soft and status is dirty then apply low power for long cycle time.
   - If clothe material is medium and status is dirty then apply medium power for long cycle time.
   - If clothe material is hard and status is clean then apply medium power for long cycle time.
   - If clothe material is hard and status is dirty then apply high power for long cycle time.

4) Attach the screenshots of FIS editor, membership functions of all input & output parameters, rule editor and rule viewer.

Attach screenshots here.

# Lab Session 14

## *Complex Engineering Activity*

## PROBLEM STATEMENT

Attach problem statement here as assigned by teacher.

Problem statement includes;

1. Task to be accomplished.
2. Complex Problem-Solving attributes covered in this CEP. (As per OBA manual)
3. Rubrics with explicit reference to attributes mentioned in 2.
4. Deliverables.
    a. Technical Documentation
    b. Hardcopy of code
    c. Softcopy of code
    d. GitHub repository link of your code

## GRADING RUBRIC

## SOLUTION

Create a GitHub repository of assigned project and share URL here.

---

Attach your code and relevant screenshots here

**NED University of Engineering & Technology**
**Department of Computer and Information Systems Engineering**

Course Code and Title: <CS-323 Artificial Intelligence>

Laboratory Session No._____ Date: _____

| **Software Use Rubric** | | | | |
|---|---|---|---|---|
| Skill Sets | Extent of Achievement | | | |
| | 0 | 1 | 2 | 3 |
| **To what extent has the student implemented the solution?** | The solution has not been implemented. | The solution has syntactic and logical errors. | The solution has syntactic or logical errors. | The solution is syntactically and logically sound for the stated problem parameters. |
| **How efficient is the proposed solution?** | The solution does not address the problem adequately. | The solution exhibits redundancy and partially covers the problem. | The solution exhibits redundancy or partially covers the problem. | The solution is free of redundancy and covers all aspects of the problem. |
| **How did the student answer questions relevant to the solution?** | The student answered none of the questions. | The student answered less than half of the questions. | The student answered more than half but not all of the questions. | The student answered all the questions. |
| **To what extent is the student familiar with the scripting/ programming interface?** | The student is unfamiliar with the interface. | The student is familiar with few features of the interface. | The student is familiar with many features of the interface. | The student is proficient with the interface. |

| | |
|---|---|
| Weighted CLO Score | |
| Remarks | |
| Instructor's Signature with Date | |

**NED University of Engineering & Technology**
**Department of Computer and Information Systems Engineering**

Course Code and Title: <CS-323 Artificial Intelligence>

Laboratory Session No._____ Date: _____

| **Software Use Rubric** | | | | |
|---|---|---|---|---|
| Skill Sets | Extent of Achievement | | | |
| | 0 | 1 | 2 | 3 |
| **To what extent has the student implemented the solution?** | The solution has not been implemented. | The solution has syntactic and logical errors. | The solution has syntactic or logical errors. | The solution is syntactically and logically sound for the stated problem parameters. |
| **How efficient is the proposed solution?** | The solution does not address the problem adequately. | The solution exhibits redundancy and partially covers the problem. | The solution exhibits redundancy or partially covers the problem. | The solution is free of redundancy and covers all aspects of the problem. |
| **How did the student answer questions relevant to the solution?** | The student answered none of the questions. | The student answered less than half of the questions. | The student answered more than half but not all of the questions. | The student answered all the questions. |
| **To what extent is the student familiar with the scripting/ programming interface?** | The student is unfamiliar with the interface. | The student is familiar with few features of the interface. | The student is familiar with many features of the interface. | The student is proficient with the interface. |

| | |
|---|---|
| Weighted CLO Score | |
| Remarks | |
| Instructor's Signature with Date | |

**NED University of Engineering & Technology**
**Department of Computer and Information Systems Engineering**

Course Code and Title: <CS-323 Artificial Intelligence>

Laboratory Session No._____ Date: _____

| **Software Use Rubric** | | | | |
|---|---|---|---|---|
| Skill Sets | Extent of Achievement | | | |
| | 0 | 1 | 2 | 3 |
| **To what extent has the student implemented the solution?** | The solution has not been implemented. | The solution has syntactic and logical errors. | The solution has syntactic or logical errors. | The solution is syntactically and logically sound for the stated problem parameters. |
| **How efficient is the proposed solution?** | The solution does not address the problem adequately. | The solution exhibits redundancy and partially covers the problem. | The solution exhibits redundancy or partially covers the problem. | The solution is free of redundancy and covers all aspects of the problem. |
| **How did the student answer questions relevant to the solution?** | The student answered none of the questions. | The student answered less than half of the questions. | The student answered more than half but not all of the questions. | The student answered all the questions. |
| **To what extent is the student familiar with the scripting/ programming interface?** | The student is unfamiliar with the interface. | The student is familiar with few features of the interface. | The student is familiar with many features of the interface. | The student is proficient with the interface. |

| | |
|---|---|
| Weighted CLO Score | |
| Remarks | |
| Instructor's Signature with Date | |