Documentation for the Uncertainty Classes

Shawn Eastwood

1 Text File Data Formats

Data related to the variables and factors will be read from `.txt` files. The data formats are listed below:

1.1 Tabular Classical Factor

```
<number of variables N (int)>
<variable name 1> <variable name 2> ... <variable name N>
<factor value 1> <factor value 2> ... <factor value M>
```

2 Variable_List

The class `Variable_List` is a database of all existing variables, storing their names along with their domain sizes. A single global instance of `Variable_List` named `the_vars` is utilized by all subsequent classes.

Within the scope of `Variable_List` there is a private structure type `Variable` that simple stores a variable name (string) and the variable's domain size (int). The domain of a variable is simply

$$\{0, 1, 2, \ldots, \texttt{domain\_size} - 1\}$$

The domain size must be ≥ 1.

`Variable_List` maintains the variable list as a pointer to an array of pointers to `Variable` structures. Variables are sequentially added to the list, and when the list is full, its length is increased by `LIST_SIZE_INCREMENT = 50`. Unused slots in the array store the `NULL` pointer.

Public methods used to interface with `Variable_List` include:

`add_variable(string name, int domain_size):` Inserts a variable into the `Variable_List`.

`get_domain_size(string name):` Extracts the domain size of the provided variable name.

2.1 Data Format

When a variable list is stored in ".txt" file to be read into the program, the data format used is (the format is not sensitive to the amount or type of whitespace, as long as whitespace is present where indicated):

```
<number of variables N (int)>
<variable name 1 (string)> <domain size 1 (int)>
<variable name 2 (string)> <domain size 2 (int)>
...
<variable name N (string)> <domain size N (int)>
```

3 Factor_Template

The class `Factor_Template` is a superclass for all factor types. The fields of `Factor_Template` include:

`int num_of_vars`: The number of variables that comprise the factor. A factor can have 0 variables.

`string* factor_var_names`: The array of variable names. The order in which the names occur affects how data is organized within the factor.

`int* var_domain_sizes`: The array of domain sizes. The order corresponds to the order of `factor_var_names`. Even though the domain sizes can be looked up from `the_vars`, storing the sizes locally within the factor is more computationally efficient.

`long int* var_index_increments`: A tabular array indexed by the variables is stored in a linear fashion. `var_index_increments[i]` refers to the index increment that corresponds to a $+1$ increment in the value of variable `factor_var_names[i]`. In the linear array, the value of the last variable in `factor_var_names` cycles the fastest. Even though these increments can be computed by any method that needs them, storing these increments within the factor speeds up said methods.

$$\texttt{var\_index\_increments[num\_of\_vars - 1]} = 1$$
$$\forall i \in \{0, 1, \ldots, \texttt{num\_of\_vars} - 2\} : \texttt{var\_index\_increments[i]} = \texttt{var\_domain\_sizes[i+1]} \times \texttt{var\_index\_increments[i+1]}$$

`long int num_of_vals`: The total number of assignments to the factor's variables.

$$\texttt{num\_of\_vals} = \texttt{var\_domain\_sizes[0]} \times \texttt{var\_index\_increments[0]}$$

Constructors implemented for `Factor_Template` include:

`Factor_Template()`: Default constructor, creates a template factor with no variables.

`Factor_Template(int new_num_of_vars, string* new_factor_var_names)`: Creates a template factor from the provided parameters.

`Factor_Template(istream& in)`: Creates a template factor from the provided input data stream (more on ".txt" data formats below).

`Factor_Template(Factor_Template& existing_factor)`: The copy constructor.

`Factor_Template& operator=(Factor_Template& existing_factor)`: The overloaded assignment operator.

Public methods implemented for `Factor_Template` include:

`void extract_vars(int& int_out, string*& array_out)`: Extracts both the number of variables and the variable array and returns them via the reference parameters.

`bool has_var(string sought_var)`: Returns true if and only if the provided variable is present in the factor.

3.1 Data Format

Importing data from a ".txt" file is the most common way to generate factors. The format of the data that describes the contents of a `Factor_Template` has the following form (the format is not sensitive to the amount or type of whitespace, as long as whitespace is present where indicated):

```
<num_of_vars (int)>
<factor_var_names[0] (string)> <factor_var_names[1] (string)> ...
        <factor_var_names[num_of_vars-1] (string)>
```

4 Tabular_Factor

The class `Tabular_Factor` inherits from the `Factor_Template` class. A tabular factor stores its data in a tabular array indexed by its variables. The array is stored in a linear format and the index increments stored by the array `var_index_increments` are used to navigate the array. The new fields introduced by the `Tabular_Factor` class include:

`float* factor_vals`: The array of factor values. The array size is given by `num_of_vals`.

The same constructors for `Factor_Template` exist with `Tabular_Factor`. In cases where the initial factor values are not specified, all of the factor values are assigned 1.0. The additional constructors listed below also import data related to the factor values:

`Tabular_Factor(istream& in)`: Creates a `Tabular_Factor` from the provided input stream.

`Tabular_Factor(int new_num_of_vars, string* new_factor_var_names, float* new_vals)`: A constructor that also requests information related to the factor values.

New public methods implemented for `Tabular_Factor` include:

`void read_vals(float* new_vals)`: Updates the factor's values using data from the provided array.

`void print(ostream& out)`: Prints the factor into the given output stream.

`void extract_vals(long int& long_int_out, float*& array_out)`: Extracts both the number of factor values and the value array and returns them via reference parameters.

`Tabular_Factor& multiply(int num_of_factors, Tabular_Factor** factor_array)`: Forms the product factor by multiplying together all factors referenced by the pointer array "`factor_array`".

`void marginalize(string elim_var_name)`: Sums the stated variable out of the factor. Throws an error if the variable does not exist in the factor.

`void condition(string cond_var_name, int cond_val)`: Removes the variable "`cond_var_name`" by fixing it to the value "`cond_val`". Does nothing if the variable does not exist in the factor.

`void normalize()`: Scales all factor values so that the values sum to 1.

4.1 Data Format

Within a .txt file, the format expected for a `Tabular_Factor` is (the format is not sensitive to the amount or type of whitespace, as long as whitespace is present where indicated):

```
<Factor_Template>
<factor_vals[0] (float)> <factor_vals[1] (float)> ...
        <factor_vals[num_of_vals-1] (float)>
```

The `<Factor_Template>` field denotes the text representation for the data in the `Factor_Template` superclass.

5 Tabular_DS_Factor

The class `Tabular_DS_Factor` inherits from the `Factor_Template` class. A tabular DS factor stores each focal element as a tabular array of boolean values indexed by its variables. A value of `true` means that the variable assignment that indexes the current position in the array is contained in the focal element. Each array is stored in a linear format and the index increments stored in the array var_index_increments are used to navigate the array. The focal elements are themselves accessed from an array of pointers. The new fields introduced by the `Tabular_DS_Factor` class include:

`int num_of_focal_elements`: The number of focal elements contained by the current DS factor. There must always be at least 1 focal element.

`bool** focal_elements`: The array of pointers that reference the boolean arrays that denote the focal elements.

`float* focal_element_weights`: The array of weights that correspond to the focal elements.

The same constructors for `Factor_Template` exist with `Tabular_DS_Factor`. In cases where the initial focal elements are not specified, the vacuous DS model will be assumed. The additional constructors listed below also import data related to the focal elements:

`Tabular_DS_Factor(istream& in)`: Creates a `Tabular_DS_Factor` from the provided input stream.

`Tabular_DS_Factor(int new_num_of_vars, string* new_factor_var_names, int new_num_of_focal_elements, bool** new_focal_elements, float* new_focal_element_weights)`: A constructor that also requests information related to the focal elements.

New public methods implemented for `Tabular_DS_Factor` include:

`void print(ostream& out)`: Prints the DS factor into the given output stream.

`void extract_focal_elements(long int& domain_size_out, int& num_of_focal_elements_out, bool**& focal_elements_out, float*& focal_element_weights_out)`: Extracts the number of possible variable assignments; the number of focal elements; the focal elements themselves; and the focal element weights.

`void extend(int num_of_new_vars, string* new_vars)`: Extends the factor to incorporate the new variables listed in the input array. Duplicate variables, or variables that are already part of the factor are simply ignored.

`Tabular_DS_Factor& multiply(int num_of_factors, Tabular_DS_Factor** factor_array)`: Forms the product factor by multiplying together all factors referenced by the pointer array "`factor_array`".

`void marginalize(string elim_var_name)`: Marginalizes the stated variable out of the factor. Throws an error if the variable does not exist in the factor.

`void condition(string cond_var_name, int cond_val)`: Removes the variable "`cond_var_name`" by fixing it to the value "`cond_val`". Does nothing if the variable does not exist in the factor.

`void normalize()`: Scales all the focal element weights so that the weights all sum to 1.

5.1 Data Format

Within a .txt file, the format expected for a `Tabular_DS_Factor` is (the format is not sensitive to the amount or type of whitespace, as long as whitespace is present where indicated):

```
<Factor_Template>
<num_of_focal_elements (int)>
<focal_element_weights[0] (float)> <focal_elements[0][0] (bool)>
        <focal_elements[0][1] (bool)> ... <focal_elements[0][num_of_vals-1] (bool)>
<focal_element_weights[1] (float)> <focal_elements[1][0] (bool)>
        <focal_elements[1][1] (bool)> ... <focal_elements[1][num_of_vals-1] (bool)>
<focal_element_weights[num_of_focal_elements-1] (float)>
        <focal_elements[num_of_focal_elements-1][0] (bool)>
        <focal_elements[num_of_focal_elements-1][1] (bool)> ...
        <focal_elements[num_of_focal_elements-1][num_of_vals-1] (bool)>
```

The `<Factor_Template>` field denotes the text representation for the data in the `Factor_Template` superclass.