7. EXACT INFERENCE: VARIABLE ELIMINATION

RENATO LUI GEH
NUSP: 8536030

Abstract. This document describes how variable elimination was implemented, showing its usage and results.

1. Migration from C++ to Lua

From the beginning of the course it was decided that code would be implemented in C++11. However, my experience throughout the semester has shown that code in C++ can become quite cumbersome. Its verbosity can be at times too much, leading to confusing and cluttered code. Although its lower level of abstraction can be seen as a powerful feature, the short deadlines mean that there is little time to implement the required auxiliary modules. Furthermore, since C++ is statically typed, certain coding shortcuts are not available, which means even more code. The lack of straightforward dictionaries (i.e. maps) is also a disadvantage.

Taking all this into account, I decided to migrate all code from C++ to Lua, a dynamically typed script language. The ratio of actual coding to headaches is now greater than one, since native features in Lua allowed for simpler implementations.

In this assignment I have decided to still keep the C++ files for documentation. The header files explain somewhat clearly what each class and method does, whilst the implementation files (i.e. dot ccs) have some comments on why the methods work. Although this migration is not a port per se (as in there are some differences on how I perform some operations), most of what I coded in C++ is similar to the equivalent Lua code.

2. Code

2.1. *Factor*

The Lua file `factor.lua` contains the following public functions.

(1) `Factor.create(S, Pr)`
(2) `Factor.sum(phi, var)`
(3) `Factor.multiply(P)`

2.1.1. *`Factor.create(S, PR)`.* This function creates a new potential with scope `S` and ordered set of probabilities `Pr`. The potential is a Lua table with fields:

- `scope` : Scope of variables in the potential.
- `map` : The potential table.
- `Pr` : Set of probabilities ordered according to the variables in `S`.
- `_v` : Array of "variances".

The potential table `map` is a d-dimensional table, where $d = |\texttt{S}|$. To find the value of a complete state $\mathbf{X}$:

1 Let $\mathbf{X} = \{X_1 = x_1, \ldots, X_n = x_n\}$ be a complete state over the variables in potential ϕ, then the value of the potential $\phi(\mathbf{X})$ is:

2

3 $\phi(X_1 = x_1, \ldots, X_n = x_n)$ = ϕ.map$[x_1][x_2]\ldots[x_n]$

The value of an incomplete state comes naturally if it is a contiguous subset of $\mathbf{X}$ starting from the first variable. That is, $\mathbf{Y} = \{X_1, X_2, \ldots, X_m\}$, where $m < n$. This is true because the value of $\phi[x_1]\ldots[x_m]$ is a table that contains the remaining values of the probability distribution.

The set of probabilities `Pr` is ordered according to the scope `S`. That is, the scope attributes an order to each variable, where the "leftmost" variable equals to the most significant bit and the rightmost variable the least significant bit. The array `Pr` is then set as the value of each of the bits set in an increasing order according to their bit significance.

The array `_v` shows how much each bit varies in `Pr`. The least significant bit always has variance 1. Each subsequent bit is the product between the previous least significant bit's variance and the number of possible values that bit may take.

This function calls a private function `combine` that recursively constructs `map`.

2.1.2. *`Factor.sum(phi, var)`.* This function sums out the variable `var` from `phi`.

This is done by iterating over clusters of values and then summing all the values from `var` by adding an offset `variance` to the indeces.

This offset is chosen according to its position in the table (bit position) and the number of possible values that variable may take.

2.1.3. *`Factor.multiply(P)`.* Multiplies all potentials in the set `P`.

2.2. *Interaction Graph*

The Lua file `interaction_graph.lua` contains the following public functions.

(1) `InteractionGraph.create_graph(data)`
(2) `InteractionGraph.eliminate_var(v, G, P)`
(3) `InteractionGraph.min_degree(S, G, P)`

(4) `InteractionGraph.min_fill(S, G, P)`

2.2.1. *`InteractionGraph.create_graph(data)`.* Creates an interaction graph from a Bayesian Network. The argument `data` is a table that contains a list of variables `vars` in the bayesian network, and a list of potentials `pots`.

It generates an interaction graph as an adjacency list H where every key in H is a variable. Each pair `(k, v)` in H is a list with bayesian variable `k` and list of neighbours `v`. There exists an edge `var_i`→`var_j` in H iff H`[var_i][var_j] = H[var_j][var_i] = true`. Since Lua allows tables as keys, it was decided that a variable be used as key. This allows for more intuitive and readable code. The total number of neighbours a variable `var_i` is represented by H`[var_i][1]`.

2.2.2. *`InteractionGraph.eliminate_var(v, G, P)`.* Eliminates variable `v` from the interaction graph `G` given the potentials `P`.

This function constructs a set of potentials Φ, where every potential in Φ contains `v` in its scope, transforms $Ne($`v`$)$ into a clique, which equates to multiplying all factors, and then removes node `v` and all associated edges (i.e. sums `v` out). After working on the graph `G`, it removes all potentials in Φ that are also in `P` and adds the potential π to `P`, where π is the result of the multiplication and summing out of all potentials Φ.

2.2.3. *`InteractionGraph.min_degree(S, G, P)`.* Applies the min-degree heuristic to a Bayesian Network $\mathcal{N} = (S, G, P)$, where S is the scope, G is the interaction graph and P is the set of potentials in $\mathcal{N}$.

The min-degree heuristic tries to find a policy (i.e. order of elimination) π where we construct the smallest potentials. This means we must find the variables with the smallest neighbourhoods. This can be accomplished by always choosing a variable `x` in which `G[x][1]` is minimum.

The function returns two values. The first the an order of elimination that agrees with the min-degree criteria. The second is the resulting potential of the variable elimination.

2.2.4. *`InteractionGraph.min_fill(S, G, P)`.* Following the same pattern as min-degree, this function accepts a scope, interaction graph and a set of potentials of a bayesian network.

The min-fill heuristic selects a policy π that minimizes the number of edges constructed after we form a clique from nodes $Ne(X)$, where X is the variable we intend on eliminating. This is done by iterating over all nodes and checking whether each node from $Ne(X)$ is already connected to their would-be neighbours. We then find a variable that minimizes this criterion.

The function returns two values. First the order of elimination π and second the resulting potential of all operations involved in the variable elimination procedure.