Fast manipulation of multi-dimensional arrays in Matlab and R

Kevin P. Murphy

10 July 2001

1 Introduction

Probabilistic inference in graphical models with discrete random variables requires performing various operations on multi-dimensional arrays (discrete potentials). (This is true whether we use an exact algorithm like junction tree [CDLS99, HD96] or an approximate algorithm like loopy belief propagation [AM00, KFL01].) These operations consist of element-wise multiplication of two arrays of potentially different sizes, and summation (marginalization) over a subset of the dimensions. This report discusses ways to implement these operations quickly in Matlab (extended with the BNT[1]) and R[2]. (See also Acklam's "MATLAB array manipulation tips and tricks" at `www.math.uio.no/~jacklam`. Unfortunately, none of his tips and tricks are relevant to the current problem!)

2 Multiplication

2.1 Example

Let us start with an example. Consider two arrays, A and B, where A represents a function over the variables X_1, X_2, X_3, X_4 and B represents a function over X_1, X_3. We will say A's domain is $\{1, 2, 3, 4\}$, and B's domain is $\{1, 3\}$. Let the size of variable X_i (i.e., the number of possible values it can have) be denoted by S_i. Here is a straighforward implementation of $A \times B$:

```
for i=1:S1
  for j=1:S2
    for k=1:S3
      for l=1:S4
        A[i,j,k,l] = A[i,j,k,l] * B[i,l];
      end
    end
  end
end
```

[1] BNT is an open-source Matlab package for Bayes Nets. See `www.cs.berkeley.edu/~murphyk/Bayes/bnt.html`.

[2] R is an open-source, scheme-like matrix-oriented language for statistical computing. It is very similar to Matlab, and is essentially identical to S and Splus. See `http://www.R-project.org/`.

2.2 Naive way (1998)

In the above example, we hard-coded the assumption that A is 4-dimensional, that B is 2-dimensional, and that the indices that they have in common are i and l. We could implement this for an arbitrary pair of arrays, T1 and T2, as follows. T1.T contains the array A, T1.domain contains the domain of A, T1.sizes contains the size of each dimension in A, and similarly for T2. T2.domain is guaranteed to be a subset of T1.domain.

```
mask = find_equiv_posns(T2.domain, T1.domain);
S1 = prod(T1.sizes);
for i1 = 1:S1
  sub1 = ind2subv(T1.sizes, i1);
  sub2 = sub1(mask);
  i2 = subv2ind(T2.sizes, sub2);
  T1.T(i1) = T1.T(i1) * T2.T(i2);
end
```

The BNT function `find_equiv_posns` (called `match` in R) determines where in T1's domain each element of T2's domain occurs. In the above example, mask = [1 3], since X_1 (the first element of T2.domain) occurs in the first position of T1.domain, and X_3 (the second element of T2.domain) occurs in the third position of T2.domain.

The function `ind2subv` converts an integer into a vector of subscripts, so for the above example,

```
for i1=1:prod(Asizes)
  [i,j,k,l] = ind2sub(Asizes, i1);
  assert(isequal(A[i1], A[i,j,k,l]))
end
```

`ind2subv` is like the built-in `ind2sub`, but returns the result as a vector, instead of as multiple arguments. sub2 extracts the indices that correspond to the entries in T2. These are then converted back to an integer using `subv2ind`.

2.3 Pre-computing the indices (1999)

It turns out that the above method is quite slow in Matlab because of the for-loop and the need to call ind2subv and subv2ind repeatedly. We can eliminate the latter inefficiency by precomputing the indices as follows:

```
mask = find_equiv_posns(T2.domain, T1.domain);
S = prod(T1.sizes);
subs = ind2subv(T1.sizes 1:S);
ndx = subv2ind(T2.sizes, subs(:,mask));
```

We can then do the multiplication thus:

```
T1.T(:) = T1.T(:) .* T2.T(ndx);
```

This is what `jtree_fast_inf_engine` in BNT2 did. It is about 45 times faster than the previous method on small problems. (See `BNT/examples/static/time_multiply_tables.m` for the code.) Unfortunately, computing all the indices for all the pairs of potentials that must be multiplied takes a lot of time and space.

In R, this can be implemented as follows.

```
S <- prod(T1.sizes);
ndx <- array(0, c(S, length(T2.domain)))
for (i in 1:length(T2.domain)) {
  stride <- prod(T1.sizes[1:mask[i]-1])
  ndx[,i] <- gl(T2.sizes[i], stride, S) # generate levels for a discrete factor
}
T1.T <- T1.T * T2.T[ndx]
```

This takes about the same amount of time as the Matlab version. (See `BNT/examples/static/time.mult.tables.r` for the code.)

2.4 Vectorizing everything (2000)

We can eliminate the for-loop and the need for indices as follows. We make T2 the same size as T1 by replicating entries where necessary, and then just doing an element-wise multiplication:

```
temp = extend_domain(T2.T, T2.domain, T2.sizes, T1.domain, T1.sizes);
T1.T = T1.T .* temp;
```

We explain `B = extend_domain_table(A, Adom, Asizes, Bdom, Bsizes)` using an example. Let Adom = [1 3], Asizes = [2 2], Bdom = [1 2 3 4] and Bsizes = [2 2 2 2] (so all variables are binary). First we reshape A to make it have size [2 1 2 1], so it has the same number of dimensions as T1. Call the result B.

```
map = find_equiv_posns(Adom, Bdom);
sz = ones(1, length(Bdom));
sz(map) = Asizes;
B = reshape(A, sz);
```

Now we replicate B along dimensions 2 and 4, to make it the same size as Bsizes.

```
sz = Bsizes;
sz(map) = 1;
B = repmat(B, sz(:)');
```

Now we are ready to multiply: `B .* A`. This is about 6 times faster than the previous method.

Doug Schwarz wrote a genops class using C that can avoid the repmat above. See `http://myweb.servtech.com/~schwarz` This is about 1.4 times faster than the pure matlab version.

The file `BNT/examples/static/time_multiply_tables.m` compares 3 methods: pre-computing indices, the repmat version and the Genops version. To multiply two tables 10 times, one with domain 1:10 and the other with domain 1:2:10, where each node has size 3, takes 8.3632, 0.1471 and 0.1049s respectively.

Unfortunately, R does not seem to have a way to do repmat. Instead, we must first make many copies of B to make it the same size as A, then permute the dimensions, and then multiply:

```
B <- array(B, Asz) # duplicate elements of B to make it same size as A
diffdom <- Adom[-Bdom] # Adom \ Bdom
B2dom <- c(Bdom, diffdom)
perm <- match(Adom, B2dom)
B <- aperm(B, perm) # make B2 have the same order as A (VERY SLOW)
A <- A * B  # A = T1.T, B = T2.T
```

To do the same computation as described in `time_multiply_tables.m` above takes 1.42 in R version 1.2.3, and 0.42s in R version 1.3.0 (which has a much faster implementation of aperm). So it seems that R is about 4 times slower than Matlab, at least on this test.

2.5 Generating C code (2001)

Unfortunately, even the vectorized solution is quite slow. So Wei Hu is currently writing Matlab/C code that will generate customized C code to implement the method in Section 2.3. (By "customized", we mean that the model compiler will generate new C code every time the model changes.) The resulting routine in BNT3 is called `jtree_compiled_inf_engine`. The indices can be computed quickly in C, and then hard-coded directly into the generated C file instead of being stored in memory. Early experiments indicate a speedup of a factor of 3–10 over the method in Section 2.4. (The idea of generating compiled code from a model has also been proposed in [Bun94, FS01, DP97].)

3 Summation/Marginalization

This section to be written.

References

[AM00] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Trans. Info. Theory*, 46(2):325–343, March 2000.

[Bun94] W. L. Buntine. Operations for learning with graphical models. *J. of AI Research*, pages 159–225, 1994.

[CDLS99] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer, 1999.

[DP97] Adnan Darwiche and Gregory Provan. Query DAGs: A practical paradigm for implementing belief-network inference. *J. of AI Research*, 6:147–176, 1997.

[FS01] B. Fischer and J. Schumann. Generating data analysis programs from statistical models. *J. Functional Programming*, 2001. Submitted.

[HD96] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *Intl. J. Approx. Reasoning*, 15(3):225–263, 1996.

[KFL01] F. Kschischang, B. Frey, and H-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans Info. Theory*, February 2001.