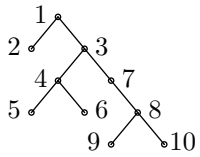
(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. I'm experimenting with what may be a new twist(?) on dynamic programming. It's motivated by "Bayesian networks" that form a binary tree. With this method we can answer queries that are much different from the usual "marginal" distributions. For example, with binary states, we can determine the probability that exactly half of the nodes are 1, in $O(n^3)$ steps. In general we can determine the probability that a Boolean function $f(x_1,\ldots,x_n)$ is true, as long as the BDD for that function is small when the nodes appear in arbitrary order. (More precisely, I have a particular order in mind, for each binary tree; the function should have a small BDD when the variables are inspected in that order.)

Here's the problem: Given a binary tree of n nodes, with $n-1$ weight functions $w_k(x_j,x_k)$ on the edge from node j to a child node k. Assign binary values $(x_1,\ldots,x_n)$ to the nodes. Every such state occurs with relative weight $W(x_1,\ldots,x_n)=\prod w_k(x_j,x_k)$, where the product is over all edges.

For example, the binary tree



has the weight function

$$W(x_1,\ldots,x_{10}) = w_2(x_1,x_2)\,w_3(x_1,x_3)\,w_4(x_3,x_4)\,w_5(x_4,x_5)\\ w_6(x_4,x_6)\,w_7(x_3,x_7)\,w_8(x_7,x_8)\,w_9(x_8,x_9)\,w_{10}(x_8,x_{10}).$$

Without loss of generality we can assume that the left subtree of each node has at most as many nodes as the corresponding right subtree, and that the nodes have been numbered in preorder. Both of these assumptions are in fact fulfilled in the example above. (Surprise!)

If the QDD for a Boolean function $f(x_1,\ldots,x_n)$ has N nodes, a variant of the algorithm below computes $\sum\{W(x_1,\ldots,x_n)\mid f(x_1,\ldots,x_n)=1\}$ in $O(nN)$ steps. Here I demonstrate the idea when f is the symmetric function $S_m(x_1,\ldots,x_n)=[x_1+\cdots+x_n=m]$.

2. For $1\le i\le n$, let $W_i(x_i,\ldots,x_n)=\prod w_k(x_j,x_k)$, where the product includes only edges jk with $j\ge i$. Thus, for instance, $W_7(x_7,x_8,x_9,x_{10})$ in the example above is $w_8(x_7,x_8)\,w_9(x_8,x_9)\,w_{10}(x_8,x_{10})$. In general we have $W_n(x_n)=1$ and $W_1(x_1,\ldots,x_n)=W(x_1,\ldots,x_n)$.

If node j has no children, $W_j(x_j,\ldots,x_n)=W_{j+1}(x_{j+1},\ldots,x_n)$. If node j has one child, it's the right child, and it's node $j+1$; hence $W_j(x_j,\ldots,x_n)=w_{j+1}(x_j,x_{j+1})\,W_{j+1}(x_{j+1},\ldots,x_n)$ in that case. Otherwise node j has two children, $j+1$ and k; then we have

$$W_j(x_j,\ldots,x_n)=w_{j+1}(x_j,x_{j+1})\,w_k(x_j,x_k)\,W_{j+1}(x_{j+1},\ldots,x_n).$$

Let S_j be the set of all x_k such that $k>j$ and k is the right child of i for some $i<j$. For example, the S's corresponding to the tree above are

$$\begin{aligned} S_1&=\emptyset, & S_6&=\{x_7\},\\ S_2&=\{x_3\}, & S_7&=\emptyset,\\ S_3&=\emptyset, & S_8&=\emptyset,\\ S_4&=\{x_7\}, & S_9&=\{x_{10}\},\\ S_5&=\{x_6,x_7\}, & S_{10}&=\emptyset. \end{aligned}$$

These sets are easy to compute, for increasing values of j:

$$S_{j+1}=\begin{cases} S_j\setminus\{x_{j+1}\}, & \text{if node } j \text{ is childless;}\\ S_j, & \text{if it has just the right child } j+1;\\ S_j\cup\{x_k\}, & \text{if its children are } j+1 \text{ and } k. \end{cases}$$

3. What's the point? Well, the S's allow us to compute the functions

$$T_j(s,x_j,S_j)=\sum\{W_j(x_j,\ldots,x_n)\mid x_j+\cdots+x_n=s\},$$

where the sum is over all variables x_k with $k>j$ that are not in S_j. For example, in the tree above,

$$T_6(2,0,1)=\sum_{p=0}^{1}\sum_{q=0}^{1}\sum_{r=0}^{1}W_6(0,1,p,q,r)[p+q+r=1].$$

The overall answer that we're trying to compute is $T_1(m,0)+T_1(m,1)$.

And the T's satisfy a simple bottom-up recursion, starting with

$$T_n(s,x_n)\;=\;[x_n=s].$$

Namely, if node j is childless, for $j<n$, we have $T_j(s,x_j,S_j)=T_{j+1}(s-x_j,x_{j+1},S_{j+1})$; notice that this formula makes sense, because $x_{j+1}\in S_j$ by the definition of preorder. On the other hand if node j has the unique child $j+1$, we have $T_j(s,x_j,S_j)=\sum_{p=0}^{1}w_{j+1}(x_j,p)\,T_{j+1}(s-x_j,p,S_j)$. And finally if node j has both $j+1$ and k as children, the formula is

$$T_j(s,x_j,S_j)=\sum_{p=0}^{1}w_{j+1}(x_j,p)\sum_{q=0}^{1}w_k(x_j,q)\,T_{j+1}(s-x_j,p,q,S_j).$$

In this case x_k is the "leftmost" element of S_{j+1}, because the S's grow in a last-in-first-out manner.

It suffices to restrict the value of s to the range $\max(0,m+1-j)\le s\le\min(m,n+1-j)$, because no other values of s at step j contribute to the final $T_1(m,0)$ and $T_1(m,1)$.

4. Still we might ask, what's the point? We've computed each function value for T_j with only a few multiplications and additions, but the number of such function values is potentially huge. If S_j has r elements, we need to keep 2^{r+1} values of $T_j(s,x_j,S_j)$ for each relevant value of s.

Fortunately, r cannot become very large; and that, in fact, is the real point of this whole method. The value of $r+1$ cannot exceed $\lfloor\lg(n+1)\rfloor$ (and it is often much smaller).

Proof. Let M_n be the largest value of $|S_j|+1$, over all binary trees with n vertices; we shall show that $M_n=\lfloor\lg(n+1)\rfloor$, by induction. Clearly $M_0=0$, if we understand that case properly. When $n\ge0$, it's not difficult to see that $M_{n+1}=\max_{0\le k\le n-k}(\max(M_k+1,M_{n-k}))$; and this will exceed M_n only if $M_k=M_{n-k}$, because $M_k\le M_{n-k}$ whenever $k\le n-k$. QED.

(Even more is true, in fact: The total of 2^{r+1} over all levels j is always $O(n^{\lg3})=O(n^{1.585})$. Thus the total running time for the algorithm is $O(n^{2.585})$, not merely $O(n^3)$; in general, for functions with at most M nodes per level in their QDD, the running time is $O(n^{1.585}M)$. I have some notes on this, and have submitted it as OEIS sequence A193494.)

5. Implementation. Instead of allocating storage and computing the results myself, I'm just testing the formulas today. So this program simply outputs a Mathematica program that does the actual computation.

The input on *stdin* is supposed to be a list of edge pairs "j k", one per line, in lexicographic order. The program doesn't check carefully for bad input, but it does panic if something unexpected happens.

The command line should contain the parameter m.

```
#define maxn 100
#define bufsize 50
#include <stdio.h>
#include <stdlib.h>
  char buf[bufsize];
  int edgej[maxn], edgek[maxn];
  int S[maxn][maxn];      /* overkill, but we accept left-heavy trees */
  int kids[maxn];
  int where[maxn];
  int x[maxn];
  main(int argc, char *argv[])
  {
    register int j, k, n, p, q, r, s;
    int m;
    ⟨ Parse the command line 6 ⟩;
    ⟨ Input the tree 7 ⟩;
    ⟨ Compute the S's 8 ⟩;
    ⟨ Output the necessary computations 9 ⟩;
  }
```

6. ⟨ Parse the command line 6 ⟩ ≡

```
  if (argc ≠ 2 ∨ sscanf(argv[1], "%d", &m) ≠ 1) {
    fprintf(stderr, "Usage: %s m\n", argv[0]);
    exit(−99);
  }
```

This code is used in section 5.

7.

```
#define panic(mess)
        { fprintf(stderr, "%s!\n", mess); exit(−1); }
```

⟨ Input the tree 7 ⟩ ≡

```
  for (n = 1; ; n++) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (n ≡ maxn) panic("too many edges");
    if (sscanf(buf, "%d %d", &edgej[n], &edgek[n]) ≠ 2) panic("bad input line");
    kids[edgej[n]]++;
    where[edgej[n]] = n;
  }
  if (edgek[n − 1] ≠ n) panic("inconsistent input");
```

This code is used in section 5.

8. ⟨ Compute the S's 8 ⟩ ≡

```
for (j = 1; j < n; j++) {
  switch (kids[j]) {
  case 2: S[j+1][0] = edgek[where[j]];
    for (k = 0; S[j][k]; k++) S[j+1][k+1] = S[j][k];
    if (edgek[where[j]-1] ≠ j+1) panic("bad edge for two-kid node");
    break;
  case 1: for (k = 0; S[j][k]; k++) S[j+1][k] = S[j][k];
    if (edgek[where[j]] ≠ j+1) panic("bad edge for one-kid node");
    break;
  case 0: if (S[j][0] ≠ j+1) panic("bad preorder for no-kid node");
    for (k = 1; S[j][k]; k++) S[j+1][k-1] = S[j][k];
    break;
  default: panic("too many kids");
  }
}
if (S[n][0]) panic("S[n] not empty");
```

This code is used in section 5.

9. ⟨ Output the necessary computations 9 ⟩ ≡

```
for (s = 0; s < 2; s++)
  for (k = 0; k < 2; k++) printf("T[%d,%d,%d]=%d\n", n, s, k, s ≡ k);
for (j = n-1; j; j--) {
  for (s = 0; s ≤ m; s++) {
    if (s < m+1-j) s = m+1-j;
    if (s > n+1-j) break;
    for (k = 0; S[j][k]; k++) ;
    r = k;
    while (1) {
      ⟨ Output T[j, s, x[0], ..., x[r]] 10 ⟩;
      for (k = 0; x[k]; k++) {
        x[k] = 0;
      }
      if (k > r) break;
      x[k] = 1;
    }
  }
}
printf("ans=T[1,%d,0]+T[1,%d,1]\n", m, m);
```

This code is used in section 5.

10. ⟨ Output $T[j,s,x[0],\ldots,x[r]]$ 10 ⟩ ≡

```
printf("T[%d,%d",j,s);
for (k=0; k≤r; k++) printf(",%d",x[k]);
printf("]=");
if (s−x[0]<0 ∨ s−x[0]>n−j) printf("0");
else
  switch (kids[j]) {
  case 0: ⟨ Output the no-kid case 11 ⟩; break;
  case 1: ⟨ Output the one-kid case 12 ⟩; break;
  case 2: ⟨ Output the two-kid case 13 ⟩; break;
  }
printf("\n");
```

This code is used in section 9.

11. ⟨ Output the no-kid case 11 ⟩ ≡

```
printf("T[%d,%d",j+1,s−x[0]);
for (k=1; k≤r; k++) printf(",%d",x[k]);
printf("]");
```

This code is used in section 10.

12. ⟨ Output the one-kid case 12 ⟩ ≡

```
for (p=0; p<2; p++) {
  if (p) printf("+");
  printf("w[%d,%d,%d]",j+1,x[0],p);
  printf("T[%d,%d,%d",j+1,s−x[0],p);
  for (k=1; k≤r; k++) printf(",%d",x[k]);
  printf("]");
}
```

This code is used in section 10.

13. ⟨ Output the two-kid case 13 ⟩ ≡

```
for (p=0; p<2; p++) {
  if (p) printf("+");
  printf("w[%d,%d,%d](",j+1,x[0],p);
  for (q=0; q<2; q++) {
    if (q) printf("+");
    printf("w[%d,%d,%d]",edgek[where[j]],x[0],q);
    printf("T[%d,%d,%d,%d",j+1,s−x[0],p,q);
    for (k=1; k≤r; k++) printf(",%d",x[k]);
    printf("]");
  }
  printf(")");
}
```

This code is used in section 10.

14. Index.

⟨ Compute the S's 8 ⟩ Used in section 5.
⟨ Input the tree 7 ⟩ Used in section 5.
⟨ Output $T[j, s, x[0], \ldots, x[r]]$ 10 ⟩ Used in section 9.
⟨ Output the necessary computations 9 ⟩ Used in section 5.
⟨ Output the no-kid case 11 ⟩ Used in section 10.
⟨ Output the one-kid case 12 ⟩ Used in section 10.
⟨ Output the two-kid case 13 ⟩ Used in section 10.
⟨ Parse the command line 6 ⟩ Used in section 5.

TREEPROBS