ベイジアンネットワークの機械語へのコンパイル

北海道大学情報科学研究科　高橋　渉
北海道大学情報科学研究科　湊　真一
○JST ERATO 研究員　川原　純

今回の話

- 今回の話は北大・情報M1の高橋さんの研究の話です。

ベイジアンネットワークとMLF式

- MLF式：ベイジアンネットワークの確率推論の計算式をすべて表したもの

$P(A = 0) = 0.8$
$P(A = 1) = 0.2$

$P(B = 0 \mid A = 0) = 0.7$
$P(B = 1 \mid A = 0) = 0.3$
$P(B = 0 \mid A = 1) = 0.6$
$P(B = 1 \mid A = 1) = 0.4$

$P(A = 0 \wedge B = 0) = 0.8 \times 0.7$
$P(A = 0 \wedge B = 1) = 0.8 \times 0.3$
$P(A = 1 \wedge B = 0) = 0.2 \times 0.6$
$P(A = 1 \wedge B = 1) = 0.2 \times 0.4$

このベイジアンネットワーク全体を表すMLF式

$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 + \lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3$$
$$+ \lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6 + \lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4$$

$P(A = 0) = 0.8$
$P(A = 1) = 0.2$

$P(B = 0 \mid A = 0) = 0.7$
$P(B = 1 \mid A = 0) = 0.3$
$P(B = 0 \mid A = 1) = 0.6$
$P(B = 1 \mid A = 1) = 0.4$

$P(A = 0 \wedge B = 0) = 0.8 \times 0.7$
$P(A = 0 \wedge B = 1) = 0.8 \times 0.3$
$P(A = 1 \wedge B = 0) = 0.2 \times 0.6$
$P(A = 1 \wedge B = 1) = 0.2 \times 0.4$

このベイジアンネットワーク全体を表すMLF式

$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 + \lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3$$
$$+ \lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6 + \lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4$$

観測値に矛盾する変数に0を代入

例えば A=0, B=0 が観測されたなら $\lambda_{A1} = 0, \lambda_{B1} = 0$

$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 + \cancel{\lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3}$$
$$+ \cancel{\lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6} + \cancel{\lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4}$$

$\lambda_{A0} = 1, \lambda_{B0} = 1$ を代入 0.8×0.7

$P(A = 0) = 0.8$
$P(A = 1) = 0.2$

$P(B = 0 \mid A = 0) = 0.7$
$P(B = 1 \mid A = 0) = 0.3$
$P(B = 0 \mid A = 1) = 0.6$
$P(B = 1 \mid A = 1) = 0.4$

$P(A = 0 \wedge B = 0) = 0.8 \times 0.7$
$P(A = 0 \wedge B = 1) = 0.8 \times 0.3$
$P(A = 1 \wedge B = 0) = 0.2 \times 0.6$
$P(A = 1 \wedge B = 1) = 0.2 \times 0.4$

このベイジアンネットワーク全体を表すMLF式

$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 + \lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3$$
$$+ \lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6 + \lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4$$

観測値に矛盾する変数に0を代入

例えば B=0 が観測されたなら $\lambda_{B1} = 0$

$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 + \cancel{\lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3}$$
$$+ \lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6 + \cancel{\lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4}$$

$\lambda_{A0} = 1, \lambda_{A1} = 1, \lambda_{B1} = 1$ を代入

$$0.8 \times 0.7$$
$$+ 0.2 \times 0.6$$

MLF式の計算

- MLF式を高速に計算する方法を考える

  要は足し算、掛け算（積和形）の高速化

  特に、λ 変数の 0 or 1 を変えて繰り返し計算したい

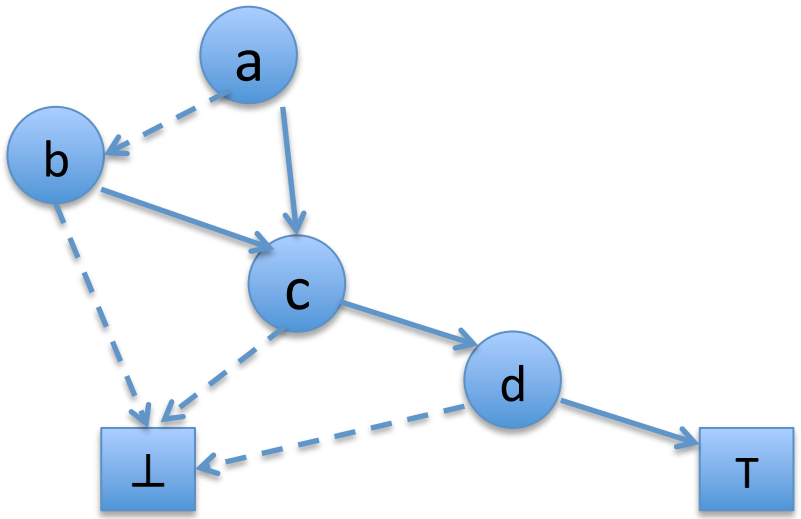
$$\lambda_{A0} \times \lambda_{B0} \times 0.8 \times 0.7 \quad + \quad \lambda_{A0} \times \lambda_{B1} \times 0.8 \times 0.3$$
$$+ \ \lambda_{A1} \times \lambda_{B0} \times 0.2 \times 0.6 \quad + \quad \lambda_{A1} \times \lambda_{B1} \times 0.2 \times 0.4$$

算術計算とZDD(1)

- 積和式の各項を組合せ集合として表す

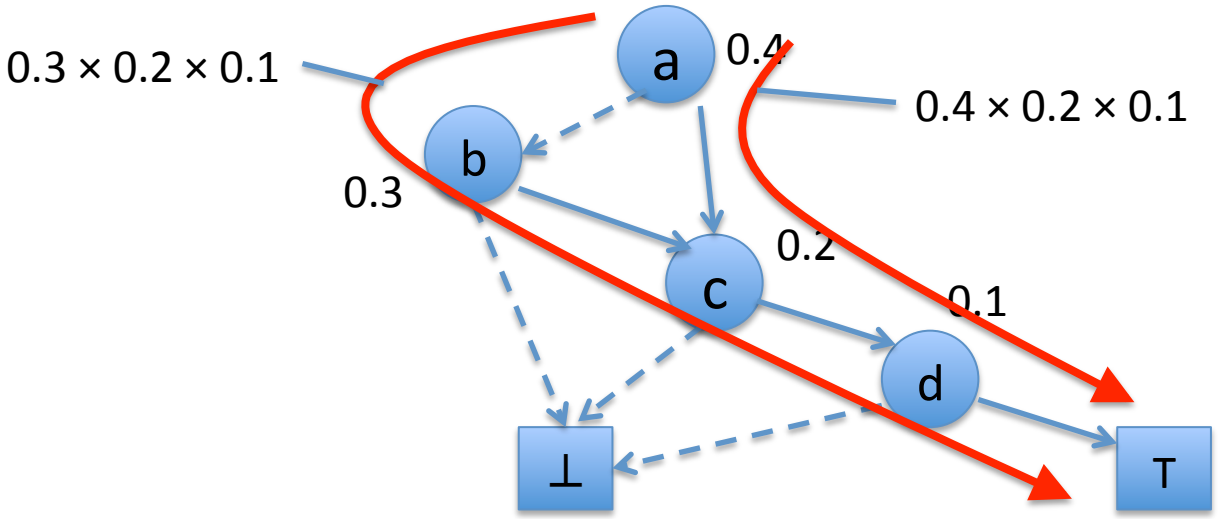
$$a \times c \times d \;+\; b \times c \times d \quad \rightarrow \quad \{\ \{a, c, d\}, \{b, c, d\}\ \}$$

- 組合わせ集合をZDDで表す



算術計算とZDD(2)

- ZDD として表すことができれば
積和式の計算を効率良く行える $a \times c \times d \ + \ b \times c \times d$



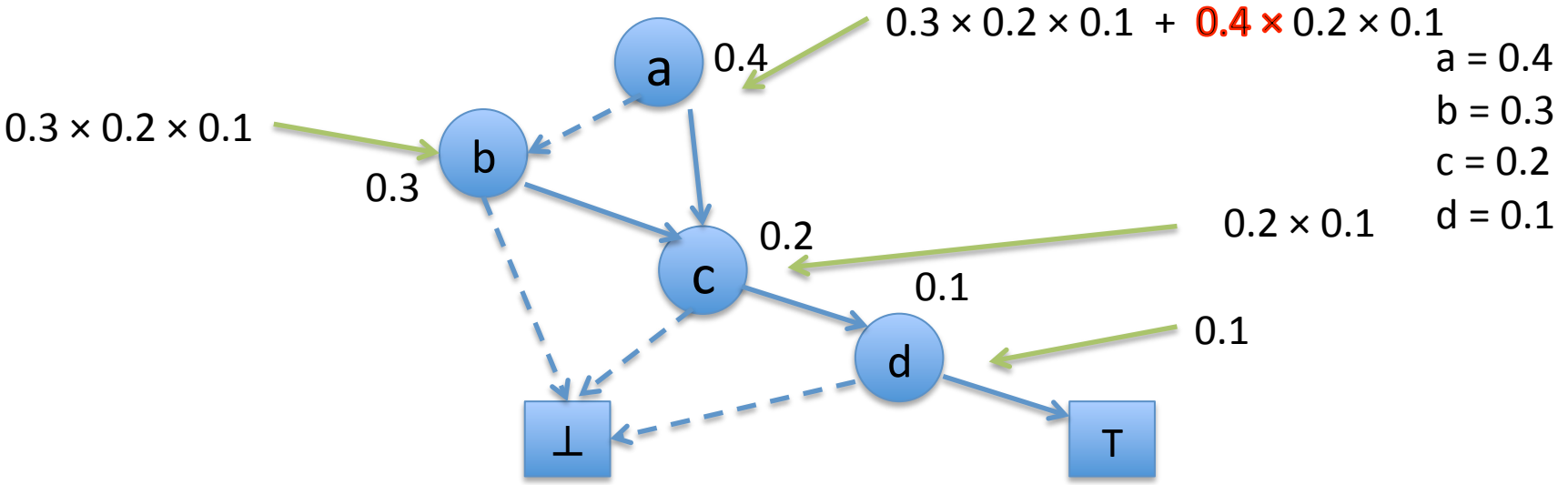
a = 0.4
b = 0.3
c = 0.2
d = 0.1

T に到達するパスが項に対応する。
HI枝を選んだときの節点の数字を掛け合わせる。

算術計算とZDD(3)

- 下から上へ向かって計算する

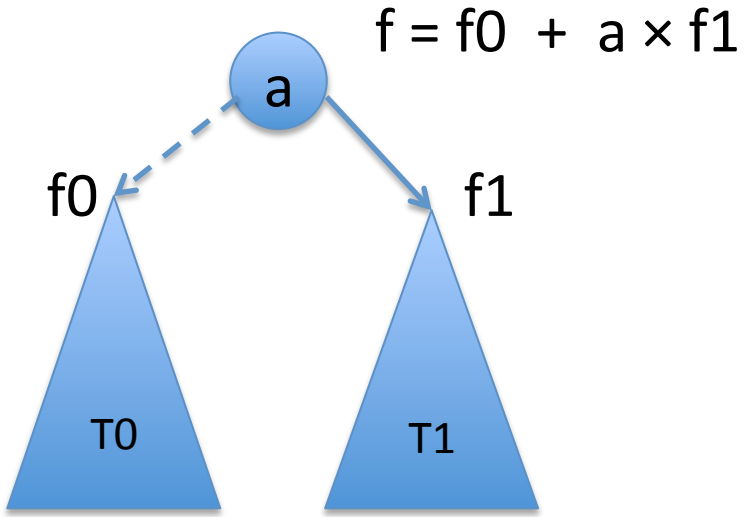
$$a \times c \times d \;+\; b \times c \times d$$



T に到達するパスが項に対応する。
HI枝を選んだときの節点の数字を掛け合わせる。

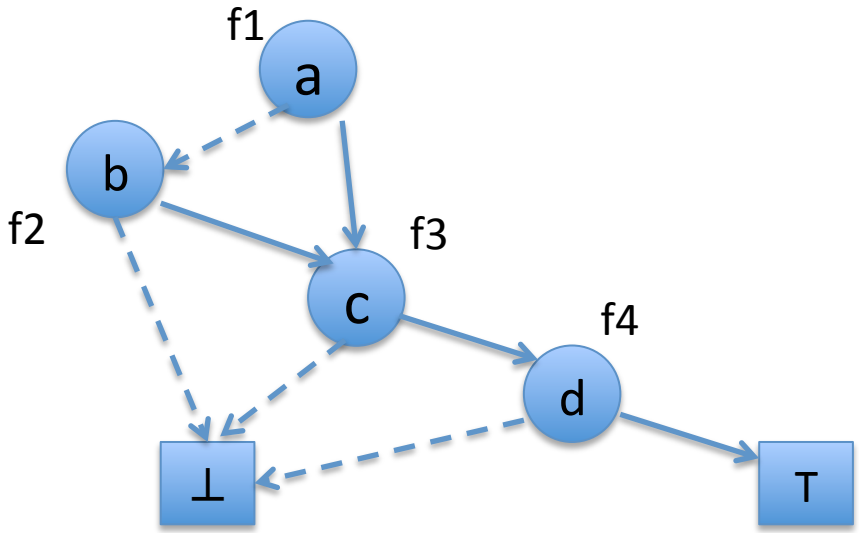
算術計算とZDD(4)

- 以下の計算を節点ごとに、下から上に向かって行う



MLFを計算するC言語コード生成

- 以下のC言語のコードを生成




```
double f1, f2, f3, f4;
double a, b, c, d;

f4 = 0 + d × 1;
f3 = 0 + c × f4;
f2 = 0 + b × f3;
f1 = f2 + a × f3;

return f1;
```

gccコンパイルエラー

- 数十万行を超えたあたりからコンパイルエラーが発生

  → 原因を探すため実験

n 個

double f1, f2, f3,…;

m と n の値を色々変えてみると
n の値によらず、m の値が 30万を超えると
コンパイル時にメモリ不足（4G超え）でエラー
→計算式の最適化でメモリを食う

m 個

f4 = 0 + d × 1;
f3 = 0 + c × f4;
f2 = 0 + b × f3;
…

m の値によらず、n の値が5万を超えると
スタックオーバーフローでエラー
→ malloc を使えば解決

for ループでまわす

double f1, f2, f3,…;

m 個

f4 = 0 + d × 1;
f3 = 0 + c × f4;
f2 = 0 + b × f3;
…

```
for ( i = 0; i < 100000; ++i ) {
    f[a[i]] = f[b[i]] + x[c[i]] * f[d[i]];
}
```

a[i], b[i], d[i] ： 節点番号
c[i] ： 変数番号
これらはテキストを読み込んで用意する

実験結果

m=100,000

	式を並べる	forループ
コンパイル時間	54	0.036
計算時間	0.0064	0.033

sec.

m=1,000,000

	式を並べる	forループ
コンパイル時間	エラー	0.037
計算時間	---	0.30

sec.

m=200,000

	式を並べる	forループ
コンパイル時間	115	0.039
計算時間	0.013	0.062

sec.

n = 100 個

double f1, f2, f3,…;

m 個

f4 = 0 + d × 1;
f3 = 0 + c × f4;
f2 = 0 + b × f3;
…

Intel Core 2 Duo 2.13GHz　2GB メモリ、Mac OS X 10.6

今後の課題

- 高速な計算アルゴリズム（理論的、数値的）を考える
- 一般にグラフ構造上での計算
- FPU の構造をうまく利用した計算
- 並列化、GPGPU

```
fld  a     // スタックにプッシュ
fadd b   // 足し算（スタックの先頭との）
fst  c     // スタックからポップ
```

スタックのサイズは8