代数的操作を用いた有限確率モデルの記述と多段階計算による推論コード生成

畠山 竜迅

本研究では，状態集合と観測集合が有限である隠れマルコフモデル（HMM）を対象とする．Forward 法と Viterbi 法では，遷移ごとの計算と遷移先ごとの集約を半環（semiring）の乗法と加法として扱うことにより，両者の更新をひとつの漸化式で記述できる．遷移関係が疎なら，正の遷移確率をもつ遷移だけを処理すればよい．ただし，この定式化は推論に用いる重みが与えられていることを前提としており，モデルの記述をもとに重みや疎な遷移関係を構成する過程は扱っていない．モデルの各分布を配列や疎形式で直接与えると，その記述は状態・観測の符号化や格納形式に依存する．そこで本研究では，任意の集合上の有限台の非負重み関数を構成・変換する weighted，η，map，map2，score および capp を代数的操作として用い，これらによって HMM の各分布を記述する．同じ記述を重み解釈と台解釈で評価し，各操作について，台解釈の結果が重み解釈によって定まる重み関数の台と一致することを示す．遷移分布について両解釈を照合し，正の遷移確率をもつ遷移とその重みから疎表現を構成する．さらに，BER MetaOCaml の型付き多段階計算を用い，この疎表現にもとづく半環上の推論コードをモデルごとに特殊化する．遷移先と遷移の重みを生成コード中の定数とする方式では，直接記述した実装に対する速度比は出次数 1 で約 2.45，32 で約 1.27 であり，出次数の増加にともなってコード生成時間と最大常駐集合サイズも増加した．

1 はじめに

隠れマルコフモデル（hidden Markov model; HMM）は，マルコフ連鎖にしたがう隠れ状態と，各状態に条件づけられた観測分布からなる時系列モデルである [16]．本論文では，隠れ状態集合と観測集合がともに有限である HMM を対象とする．扱う推論は，観測列の周辺尤度を計算する Forward 法と，観測列と状態列の同時確率を最大にする状態列を求める Viterbi 法であり，モデルパラメータの学習は対象外とする．

Forward 法と Viterbi 法では，いずれも，各遷移について，前時刻における遷移元の値，遷移の重み，および観測の重みを組み合わせて項を計算し，遷移先ごとにそれらの項を集約する．重みを組み合わせる演算と項を集約する演算とを，それぞれ半環（semiring）の乗法と加法として扱うことにより，両推論法の更新を，半環をパラメータとするひとつの漸化式として記述できる [7] [15]．ただし，Viterbi 経路を復元する場合には，最大値を与えた遷移元も集約時に記録する必要がある．また，遷移確率が 0 である状態対は更新に寄与しないため，遷移関係が疎な HMM では，正の遷移確率をもつ状態対だけを処理すればよい．この共通部分を一度だけ記述し，モデルごとに定まる遷移とその重みを用いて特殊化できれば，Forward 法と Viterbi 法の間で同じ走査処理を重複して実装せずに，推論関数の実行時に行う処理を削減できる．

このようなモデルごとの特殊化は，プログラムと，特殊化の時点で利用する入力とから，残りの入力を受けとる残余プログラムを生成する部分評価（partial evaluation）として捉えられる [10]．本研究では，BER MetaOCaml の型付き多段階計算（multi-stage programming）[2] [12] を用い，コード生成時に行う計算と，生成された推論関数の実行時に行う計算と

---

* Description of Finite Probabilistic Models Using Algebraic Operations and Inference Code Generation by Multi-stage Programming
This is an unrefereed paper. Copyrights belong to the Author.

Ryujin Hatakeyama, 東北大学情報科学研究科, Graduate School of Information Science, Tohoku University.

をプログラム上で区別する．本論文で扱う HMM では，状態集合，正の遷移確率をもつ状態対，および各遷移の重みをコード生成時に利用できる．一方，観測列は生成された推論関数の引数として与えられ，各時刻の観測値に応じた重みと状態ごとの値は，推論関数の実行時に計算される．したがって，状態集合，正の遷移確率をもつ状態対，および各遷移の重みにだけ依存する処理の一部をコード生成時に行い，観測列と状態ごとの値に依存する計算を生成コードに残すことができる．

しかし，モデルの各要素を整数添字付きの配列として直接記述すると，初期分布，遷移分布および観測分布の記述に，状態と観測の符号化方法が含まれる．さらに，正の遷移確率をもつ状態対を疎な配列として別に与える場合には，格納形式や走査順序もその記述に含めることになる．また，走査処理とその特殊化を推論法ごとに記述すれば，本来は共通する処理が Forward 法と Viterbi 法の間で重複する．そこで本論文では，モデルの各分布をこれらの実装上の選択から分離して記述し，その記述から正の遷移確率をもつ状態対と各遷移の重みを計算し，Forward 法と Viterbi 法に共通する推論コードを生成する．

提案手法では，任意の集合上の有限台の非負重み関数を構成・変換する weighted，η，map，map2，score および capp からなるモデルを記述するための操作インタフェースを定め，これらの操作を総称して代数的操作と呼ぶ．これらの操作を用いて HMM の初期分布，遷移分布および観測分布を記述し，同じ記述を，モジュール `Weighted` による重み解釈と，モジュール `Support` による台解釈で評価する．重み解釈では各要素の重みを計算し，台解釈では正の重みをもつ要素を列挙する．各操作について，台解釈の結果が重み解釈によって定まる重み関数の台と一致することを示し，初期分布，各遷移分布および各観測分布について，実装上も両者の対応を検査する．そのうえで，遷移分布に対する二つの評価結果を照合し，正の遷移確率をもつ状態対を，各遷移の重みとともに遷移元ごとに整理した表現を構成する．以下，これを遷移の疎表現と呼ぶ．

遷移の疎表現にもとづき，各遷移について，前時刻における遷移元の値，遷移の重み，および観測の重みを半環の乗法で組み合わせて項を計算し，遷移先ごとに半環の加法で集約するコードを生成する．コード生成器は，半環演算を表すコード値を構築するモジュール，集約時に必要な情報を記録するモジュール，および各時刻の計算結果の保存方法を定めるモジュールを引数として受けとる．これらのモジュールを推論の目的に応じて組み合わせることにより，周辺対数尤度，全時刻の Forward 値，経路の最大重み，および Viterbi 経路を計算するコードを，共通する走査処理と項の計算から生成する（表 1）．

ただし，遷移とその重みをコード生成時に利用できるとしても，それらに依存する処理をどこまでコード生成時に行うべきかは自明ではない．圧縮行格納（compressed sparse row; CSR）形式を基礎とする疎表現では，各遷移元に対応する格納区間，遷移先，および遷移の重みをコード生成時に利用できる．これらの情報に依存する配列参照やループ制御を生成コードから除けば，推論関数の実行時に行う処理を削減できる．その反面，遷移ごとの処理を複製するほど生成コードは大きくなり，コード生成時間とコード生成時の最大常駐集合サイズ（peak resident set size; peak RSS）も増加しうる．また，追加のコード生成時間に見合うだけの推論時間の短縮が生じるかどうかは，同じモデルに対して生成した推論関数を実行する回数にも依存する．

この関係を調べるため，正の遷移確率をもつ状態対を遷移元ごとに整理し，CSR 形式を基礎とする疎表現に格納する．各遷移元に対応する格納区間の端点をコード生成時に読み出すか，その区間を走査するループをコード生成時に展開するか，遷移先と遷移の重みを生成コード中に定数として記述するかを，異なる組み合わせで選択する複数のコード生成方式を実装した（表 2）．各方式について，推論の実行時間，コード生成時間，コード生成時の peak RSS，および損益分岐となる実行回数を比較する（第 5.5 節および第 5.6 節）．

生成された Forward 推論関数の実行時間を，同じ走査処理を直接記述した OCaml 実装と比較した．直接記述した実装の実行時間を生成された推論関数の

実行時間で除した速度比は，CSR 配列を推論関数の実行時に走査する `packed_dynamic` では約 1.14 から 1.20 であった．走査ループを展開し，遷移先と遷移の重みを生成コード中の定数とする `exact_embedding` では，速度比が出次数 1 で約 2.45，出次数 32 で約 1.27 であった．測定値の中央値にもとづき，生成と推論を合わせた `exact_embedding` の総実行時間が `packed_dynamic` 以下となる最小実行回数は，出次数 1 ではおおむね 1 回，出次数 32 ではおおむね 1305 回であった．本実験条件では，出次数の増加にともなって，遷移先と遷移の重みを生成コード中に定数として記述することによる推論時間の短縮幅が小さくなる一方，コード生成時間と peak RSS は増加した．

本論文の構成は次のとおりである．第 2 節では，本論文で扱う HMM，半環上の推論，遷移関係の疎性，および多段階計算を説明する．第 3 節では，有限台の非負重み関数に対する代数的操作と，重み解釈および台解釈を定める．第 4 節では，モデル記述から遷移の疎表現を構成し，Forward 法と Viterbi 法の推論コードを生成する処理を示す．第 5 節では，生成コードの数値的一致と性能を評価する．第 6 節で関連研究との関係を整理し，第 7 節でまとめと今後の課題を述べる．

2 背景

本節では，HMM の推論と多段階計算について整理する．はじめに，本論文で扱う HMM と Forward 法・Viterbi 法を定義し，具体例として Grid Sensors モデルを示す．続いて，両推論に共通する漸化式を半環上で定式化し，遷移関係の疎性を利用した計算と，多段階計算によるモデル特殊化を説明する．

2.1 隠れマルコフモデル

以下では，隠れ状態集合と観測集合がともに空でない有限集合であり，遷移確率と観測確率が時刻に依存しない斉時的（time-homogeneous）な HMM を扱う．

有限集合 U 上の確率分布の集合を

$$\mathcal{D}(U) = \left\{ p\colon U \to [0,1] \;\middle|\; \sum_{u\in U} p(u) = 1 \right\}$$

と書く．

定義 1 (HMM の仕様)．空でない有限集合 X を隠れ状態集合，空でない有限集合 Y を観測集合とする．HMM の仕様を $H = (X, Y, \pi, A, B)$ とする．ここで，$\pi \in \mathcal{D}(X)$，$A\colon X \to \mathcal{D}(X)$，$B\colon X \to \mathcal{D}(Y)$ である．π を初期分布，$A(x)$ を状態 x からの遷移分布，$B(x)$ を状態 x における観測分布と呼ぶ．また，$x, x' \in X$ および $y \in Y$ に対して $A(x,x') \coloneqq A(x)(x')$，$B(x,y) \coloneqq B(x)(y)$ と書き，それぞれ状態 x から x' への遷移確率，状態 x における観測 y の観測確率と呼ぶ．

正の整数 T に対し，隠れ状態列を $x_{1:T} = (x_1, \ldots, x_T) \in X^T$，観測列を $y_{1:T} = (y_1, \ldots, y_T) \in Y^T$ とする．正の整数 t に対し，仕様 H が定める長さ t の状態列と観測列の同時確率を $p_{H,t}\colon X^t \times Y^t \to [0,1]$ とし，

$$p_{H,t}(x_{1:t}, y_{1:t}) = \pi(x_1)B(x_1, y_1) \times \prod_{s=2}^{t} A(x_{s-1}, x_s)B(x_s, y_s). \tag{1}$$

と定める．ただし，空積は 1 とする．

正の遷移確率をもつ状態対の集合を

$$E_A = \{(x,x') \in X \times X \mid A(x,x') > 0\} \tag{2}$$

とし，以下，これを遷移関係 E_A と呼ぶ．

2.1.1 Forward 法

T を固定する．観測列 $y_{1:T} \in Y^T$ に対する周辺尤度を

$$L_H(y_{1:T}) = \sum_{x_{1:T}\in X^T} p_{H,T}(x_{1:T}, y_{1:T})$$

と定める．各時刻 t について，Forward 値を表す関数 $\alpha_t\colon X \to [0,1]$ を

$$\alpha_t(x') = \sum_{\substack{x_{1:t}\in X^t \\ x_t = x'}} p_{H,t}(x_{1:t}, y_{1:t})$$

と定めると，$\alpha_1(x') = \pi(x')B(x', y_1)$ であり，$2 \le t \le T$ を満たす各整数 t について，次の漸化式を満たす．

$$\alpha_t(x') = B(x', y_t) \sum_{x\in X} \alpha_{t-1}(x)A(x,x') \tag{3}$$

周辺尤度は $L_H(y_{1:T}) = \sum_{x'\in X} \alpha_T(x')$ である．

2.1.2 Viterbi 法

観測列 $y_{1:T}$ が与えられたとき，Viterbi 法は，同

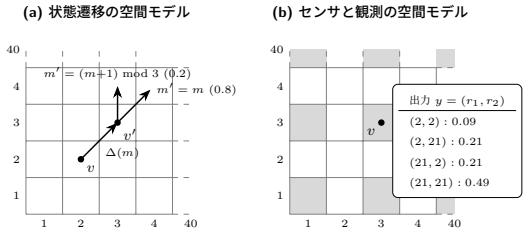

図 1　Grid Sensors モデルの位置と観測

時確率を最大にする状態列

$$x^*_{1:T} \in \arg\max_{x_{1:T} \in X^T} p_{H,T}(x_{1:T}, y_{1:T})$$

を求める．$L_H(y_{1:T}) > 0$ ならば，観測列 $y_{1:T}$ のもとでの状態列の事後確率は

$$p_H(x_{1:T} \mid y_{1:T}) = \frac{p_{H,T}(x_{1:T}, y_{1:T})}{L_H(y_{1:T})}$$

である．分母は状態列 $x_{1:T}$ に依存しないため，同時確率を最大にする状態列の集合と，事後確率を最大にする状態列の集合とは一致する．各時刻 t について，Viterbi 値を表す関数 $\delta_t \colon X \to [0,1]$ を

$$\delta_t(x') = \max_{\substack{x_{1:t} \in X^t \\ x_t = x'}} p_{H,t}(x_{1:t}, y_{1:t})$$

と定めると，$\delta_1(x') = \pi(x') B(x', y_1)$ であり，$2 \le t \le T$ を満たす各整数 t について，次の漸化式を満たす．

$$\delta_t(x') = B(x', y_t) \max_{x \in X} \delta_{t-1}(x) A(x, x') \qquad (4)$$

2.1.3　例：Grid Sensors

以下で用いる HMM の具体例として，EE365 の講義資料で示された Grid Sensors モデルを取り上げる [13]．

位置の集合を $G = \{1, \dots, 40\}^2$，進行方向の集合を $M = \{0, 1, 2\}$ とする．隠れ状態は位置 $v \in G$ と進行方向 $m \in M$ の組 $x = (v, m)$ であり，$|X| = 40 \cdot 40 \cdot 3 = 4800$ である．進行方向に対応する変位関数 $\Delta \colon M \to \mathbb{Z}^2$ を $\Delta(0) = (0,1)^\top, \Delta(1) = (1,0)^\top, \Delta(2) = (1,1)^\top$ とし，$\phi \colon \mathbb{Z}^2 \to G$ を各座標成分を 1 以上 40 以下に制限する関数とする．初期位置を $(1,1)$ とし，初期方向を一様とすると，$v = (1,1)$ のとき $\pi((v,m)) = 1/3$，それ以外では 0 である．次の位置 $v_t = \phi(v_{t-1} + \Delta(m_{t-1}))$ は一意に定まり，進行方向は確率 0.8 で維持され，確率 0.2 で $(m+1) \bmod 3$ へ変化する．

図 1 に示すように，各状態から正の確率で遷移できる状態はちょうど 2 状態である．状態数は 4800 であり，遷移行列の 4800^2 要素のうち正の要素は $4800 \cdot 2$ 個である．零要素の割合は約 99.958% である．

観測集合を $Y = \{1, \dots, 21\}^2$ （$|Y| = 441$）とする．各座標成分の観測確率を与える関数 $q \colon \{1, \dots, 40\} \times \{1, \dots, 21\} \to [0,1]$ を次のように定める．任意の $j \in \{1, \dots, 20\}$ について，$q(2j-1, j) = 0.3, q(2j-1, 21) = 0.7, q(2j, 21) = 1$ と定め，これら以外の組 (a, r) については $q(a, r) = 0$ とする．状態 $x = (v, m)$ における観測確率は進行方向 m には依存せず，二つの位置座標について定めた観測確率の積

$$B((v,m), (r_1, r_2)) = q(v_1, r_1) q(v_2, r_2)$$

によって与えられる．

2.2　半環上の HMM 推論

Forward 法と Viterbi 法では，各遷移について，前時刻における遷移元の値と遷移の重みとを組み合わせて項を計算し，遷移先ごとにそれらの項を集約する．Forward 法では各項を加算し，Viterbi 法では最大値をとる．この共通部分は，半環上の同じ漸化式として記述できる [7] [15]．

定義 2 (半環)**.** 半環 $\langle K; \oplus, \otimes, 0_K, 1_K \rangle$ は，集合 K，二項演算 $\oplus, \otimes : K \times K \to K$，および要素 $0_K, 1_K \in K$ からなり，次の条件を満たす．

1. $\langle K; \oplus, 0_K \rangle$ は可換モノイドである．
2. $\langle K; \otimes, 1_K \rangle$ はモノイドである．
3. 任意の $a, b, c \in K$ について，左右の分配則
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c),$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$
が成り立つ．
4. 0_K は $\otimes$ の吸収元である．すなわち，任意の $a \in K$ について
$$0_K \otimes a = 0_K = a \otimes 0_K$$
が成り立つ．

半環の一般的な議論については [5] を参照されたい．

半環 $\langle K; \oplus, \otimes, 0_K, 1_K \rangle$ をひとつとり，空の添字集合にわたる $\bigoplus$ を 0_K とする．HMM の初期状態，遷移および観測につける重みを，それぞれ $w_\pi \colon X \to K$，$w_A \colon X \times X \to K$，$w_B \colon X \times Y \to K$ とする．また，各時刻 t における状態ごとの値を表す関数を

$\rho_t\colon X \to K$ とする．初期時刻の値を

$$\rho_1(x') = w_\pi(x') \otimes w_B(x', y_1) \tag{5}$$

と定める．$2 \le t \le T$ を満たす各整数 t については，次の漸化式によって ρ_t を定める．

$$\rho_t(x') = \left(\bigoplus_{x \in X} \rho_{t-1}(x) \otimes w_A(x, x') \right) \otimes w_B(x', y_t). \tag{6}$$

確率半環 $\langle \mathbb{R}_{\ge 0}; +, \times, 0, 1 \rangle$ において $w_\pi(x) = \pi(x)$，$w_A(x, x') = A(x, x')$，$w_B(x, y) = B(x, y)$ とおけば，ρ_t は Forward 値 α_t に一致する．Viterbi 半環 $\langle \mathbb{R}_{\ge 0}; \max, \times, 0, 1 \rangle$ を用いれば，ρ_t は Viterbi 値 δ_t に一致する．

実装では，確率の積によるアンダーフローを避けるため，正の確率を対数へ変換して重みとし，対数領域で計算する．$a, b \in \mathbb{R} \cup \{-\infty\}$ に対して $\mathrm{logaddexp}(a, b) = \log(e^a + e^b)$ とし，$e^{-\infty} = 0$ および $\log 0 = -\infty$ と約束する．Forward 法には log-sum-exp 半環 $\langle \mathbb{R} \cup \{-\infty\}; \mathrm{logaddexp}, +, -\infty, 0 \rangle$ を，Viterbi 法には max-plus 半環 $\langle \mathbb{R} \cup \{-\infty\}; \max, +, -\infty, 0 \rangle$ を用いる．確率 0 は，いずれの半環でも零元 $-\infty$ に対応する．Viterbi 経路を復元する場合には，各遷移について計算した項が現在の最大値を上回ったときに，その遷移元を記録する．最大値を与える遷移元が複数ある場合には，走査順で最初に処理した遷移元を保持する．

以下では，任意の $(x, x') \notin E_A$ について，遷移の重みが $w_A(x, x') = 0_K$ を満たすものとする．また，$\rho_{t-1}(x) = 0_K$ であれば，0_K の吸収性により，x を遷移元とする各項も 0_K となる．したがって，式 (6) の有限和は，添字集合を X から $\{x \in X \colon (x, x') \in E_A\}$ へ制限しても変わらない．すなわち，

$$\rho_t(x') = \left(\bigoplus_{\substack{x \in X \\ (x, x') \in E_A}} \rho_{t-1}(x) \otimes w_A(x, x') \right) \otimes w_B(x', y_t). \tag{7}$$

さらに，右分配則により，

$$\rho_t(x') = \bigoplus_{\substack{x \in X \\ (x, x') \in E_A}} \big(\rho_{t-1}(x) \otimes w_A(x, x') \otimes w_B(x', y_t) \big). \tag{8}$$

各状態対 $(x, x') \in E_A$ に対応する項は

$$\rho_{t-1}(x) \otimes w_A(x, x') \otimes w_B(x', y_t)$$

である．式 (8) のように，各項の計算時に観測の重みまで組み合わせれば，前時刻の値と遷移の重みだけを組み合わせた中間値を別の配列に保存する必要はない．

2.3 疎な遷移関係を利用した計算

遷移関係が疎な HMM に対して，Forward 法の一時刻分の計算を密な行列ベクトル積として実装すると，プログラム 1 のようになり，状態数 n に対して $\Theta(n^2)$ の時間を要する．

```
let forward_step_dense alpha a b_y =
  let n = Array.length alpha in
  let next_alpha = Array.make n 0.0 in
  for j = 0 to n - 1 do
    let sum = ref 0.0 in
    for i = 0 to n - 1 do
      sum := !sum +. alpha.(i) *. a.(i).(j)
    done;
    next_alpha.(j) <- !sum *. b_y.(j)
  done;
  next_alpha
```

プログラム 1　密行列による一時刻分の Forward 計算

この実装では，確率が 0 である $n^2 - |E_A|$ 個の状態対についても配列参照と演算を行うため，遷移関係の疎性を計算量に反映できない．第 2.1.3 節の Grid Sensors モデルのように，状態数に比べて各状態からの遷移先が少ない HMM では，式 (7) にしたがい，E_A に属する $|E_A|$ 個の状態対だけを処理すればよい．また，遷移元の値が半環の零元 0_K であれば，その遷移元に対応する処理も省略できる．

実装では，E_A の各状態対を有向グラフの辺として格納する．以下，この疎表現に関する説明では，格納したひとつの遷移を辺と呼ぶ．辺を走査する処理を各推論法へ直接記述すると，同じ走査処理と特殊化処理が重複するため，本研究ではこれらを共通のコード生成器にまとめる．

2.4 多段階計算による特殊化

モデルごとに定まる遷移とその重みをコード生成時に利用できれば，推論コードをそのモデルに特殊化できる．部分評価は，プログラムと，特殊化の時点

で利用できる入力とから，残りの入力を受けとる残余プログラムを生成する変換である [10]．本処理系では，型付き多段階計算によってこの特殊化を記述できる BER MetaOCaml を用いる [12]．型 τ の値を計算するプログラム片を表すコード値の型を τ code と書く．コード括弧 $.\langle e \rangle.$ は，後の段階で評価される式 e を表すコード値を構築する．コード括弧内のエスケープ $.\sim e$ は，現在の段階で構築したコード値をその位置へ挿入する．

対数領域の Forward 法では，同じ状態に到達する複数の項を logaddexp によって集約する．この演算を行うコード値は，たとえば次のように構築できる．

```
let add x y =
  .<
    let vx = .~x
    and vy = .~y in
    if vx = neg_infinity then vy
    else if vy = neg_infinity then vx
    else if vx > vy then
      vx +. log (1.0 +. exp (vy -. vx))
    else
      vy +. log (1.0 +. exp (vx -. vy))
  >.
```

プログラム 2　対数領域の Forward 法に対応する半環の加法の構築

ここで，`x` と `y` の型は `float code` である．二つのコード値をエスケープによってコード括弧内へ挿入しているため，logaddexp 自体は，生成された推論関数の実行時に計算される．

3　代数的操作によるモデル記述

本節では，まず，任意の集合上の有限台の非負重み関数に対して，モデルの記述に用いる代数的操作のインタフェースを定める．次に，各操作の重み解釈と台解釈を与え，台解釈の結果が重み関数の台と一致することを示す．最後に，これらの操作を用いて HMM の初期分布，遷移分布および観測分布を記述し，Forward 漸化式との関係を説明する．

本実装では，同じ記述を二つのモジュールで評価する．`Weighted` による評価では各要素の重みを計算し，`Support` による評価では正の重みをもつ要素を列挙する．以下，前者を重み解釈，後者を台解釈と呼ぶ．

3.1　有限台の重み関数と操作インタフェース

集合 U 上の重み関数とは，非負実数値をとる関数 $u\colon U \to \mathbb{R}_{\geq 0}$ である．その台を $\mathrm{supp}(u) = \{x \in U : u(x) > 0\}$ と定め，台が有限である重み関数全体を $\mathcal{W}_{\mathrm{fin}}(U)$ と書く．また，$u \in \mathcal{W}_{\mathrm{fin}}(U)$ の総質量を $\|u\|_1 = \sum_{x \in \mathrm{supp}(u)} u(x)$ と定める．

$u \in \mathcal{W}_{\mathrm{fin}}(U)$ が $\|u\|_1 = 1$ を満たすとき，u を U 上の有限台の確率分布と呼ぶ．有限集合 U については，この集合は第 2.1 節で定めた $\mathcal{D}(U)$ に一致する．

本処理系では，有限台の重み関数を構成・変換する操作のインタフェースを，OCaml シグネチャ`ALGEBRA` として与える（プログラム 3）．型 `'a repr` は，型 `'a` の値の集合上の有限台の重み関数を記述するための抽象型である．その具体的な表現と各操作の解釈は，このシグネチャを実装するモジュールごとに定める．以下，混同のおそれがない場合には，これらを単に操作と呼ぶ．

```
1  type weight = float
2
3  module type ALGEBRA = sig
4    type 'a repr
5
6    val weighted : ('a * weight) list -> 'a repr
7    val eta : 'a -> 'a repr
8    val map : ('a -> 'b) -> 'a repr -> 'b repr
9    val map2 : ('a -> 'b -> 'c) -> 'a repr -> 'b repr -> 'c
         repr
10   val score : ('a -> weight) -> 'a repr -> 'a repr
11   val capp : ('a -> 'b repr) -> 'a repr -> 'b repr
12 end
```

プログラム 3　有限台の重み関数の操作インタフェース

型 `weight` は OCaml 上では `float` の別名であり，型だけでは非負性や有限性を表せない．このため，各操作の実装では，操作に与えられる重みを検査し，負値，NaN および無限大を拒否する．

3.2　重み関数とその台

重み解釈では，型 `'a repr` を，型 `'a` の要素と非負重みとの組からなる有限リストとして実装する．同

じ要素が複数回現れる場合には，その要素に付された重みを加算する．

集合 U, V, Z をとる．$a \in U$，$u \in \mathcal{W}_{\mathrm{fin}}(U)$，$v \in \mathcal{W}_{\mathrm{fin}}(V)$ とし，$f\colon U \to V$，$g\colon U \times V \to Z$，$q\colon U \to \mathbb{R}_{\geq 0}$ とする．また，各 $x \in U$ に有限台の重み関数 $k(x)$ を対応させる関数を $k\colon U \to \mathcal{W}_{\mathrm{fin}}(V)$ とする．さらに，$L = [(x_1, w_1), \ldots, (x_n, w_n)]$ を，$x_i \in U$，$w_i \in \mathbb{R}_{\geq 0}$ を満たす有限リストとする．空の添字集合にわたる和を 0 と約束し，各操作の重み解釈を次のように定める．

$$\mathrm{weighted}(L)(x) = \sum_{\substack{1 \leq i \leq n \\ x_i = x}} w_i \tag{9}$$

$$\eta_U(a)(x) = \begin{cases} 1 & (x = a) \\ 0 & (x \neq a) \end{cases} \tag{10}$$

$$\mathrm{map}(f, u)(y) = \sum_{\substack{x \in \mathrm{supp}(u) \\ f(x) = y}} u(x) \tag{11}$$

$$\mathrm{map2}(g, u, v)(z) = \sum_{\substack{x \in \mathrm{supp}(u),\, y \in \mathrm{supp}(v) \\ g(x,y) = z}} u(x)v(y) \tag{12}$$

$$\mathrm{score}(q, u)(x) = u(x)q(x) \tag{13}$$

$$\mathrm{capp}(k, u)(y) = \sum_{x \in \mathrm{supp}(u)} u(x)k(x)(y). \tag{14}$$

式 (11) の map は，各 $y \in V$ について，$f(x) = y$ を満たす要素 x の重みを加算する．式 (12) の map2 は，各組 $(x, y) \in \mathrm{supp}(u) \times \mathrm{supp}(v)$ に重み $u(x)v(y)$ を割り当てたうえで，各 $z \in Z$ について，$g(x, y) = z$ を満たす組の重みを加算する．

$\mathrm{score}(q, u)$ は，各 $x \in U$ に対する重み $u(x)$ に $q(x)$ を乗じる．一般に

$$\|\mathrm{score}(q, u)\|_1 = \sum_{x \in \mathrm{supp}(u)} u(x)q(x)$$

であり，この総質量は必ずしも 1 ではない．したがって，score は正規化を行う操作ではなく，非負係数 $q(x)$ によって各要素の重みを変える操作である．HMM の記述では，$q(x)$ を尤度または観測の重みとして用いる．$k_q(x)(x') = q(x)\eta_U(x)(x')$ と定めれば，$\mathrm{score}(q, u) = \mathrm{capp}(k_q, u)$ であり，score はこの処理を直接記述するための操作でもある．

関数 $k\colon U \to \mathcal{W}_{\mathrm{fin}}(V)$ と $u \in \mathcal{W}_{\mathrm{fin}}(U)$ に対し，$\mathrm{capp}(k, u)(y)$ は，各 $x \in \mathrm{supp}(u)$ から y への寄与 $u(x)k(x)(y)$ の和である．各 $k(x)$ の総質量が 1 ならば，$\mathrm{capp}(k, u)$ の総質量は u の総質量に等しい．

これらの操作と確率分布との関係は次のとおりである．$\sum_{i=1}^{n} w_i = 1$ ならば，$\mathrm{weighted}(L)$ は確率分布であり，$\eta_U(a)$ は常に確率分布である．また，u と v が確率分布ならば，$\mathrm{map}(f, u)$ と $\mathrm{map2}(g, u, v)$ も確率分布である．さらに，u が確率分布であり，各 $x \in \mathrm{supp}(u)$ について $k(x)$ が確率分布ならば，$\mathrm{capp}(k, u)$ も確率分布である．一方，$\mathrm{score}(q, u)$ は一般には正規化を保存しない．このため，本稿では各操作の値域を確率分布に限定せず，有限台の重み関数全体にとる．

命題 3. 式 (9)–(14) によって定めた各重み関数について，次が成り立つ．

$$\mathrm{supp}(\mathrm{weighted}(L)) = \left\{ x_i \,\middle|\, {}^{1 \leq i \leq n}_{w_i > 0} \right\}, \tag{15}$$

$$\mathrm{supp}(\eta_U(a)) = \{a\}, \tag{16}$$

$$\mathrm{supp}(\mathrm{map}(f, u)) = f[\mathrm{supp}(u)], \tag{17}$$

$$\mathrm{supp}(\mathrm{map2}(g, u, v)) = g[\mathrm{supp}(u) \times \mathrm{supp}(v)], \tag{18}$$

$$\mathrm{supp}(\mathrm{score}(q, u)) = \left\{ x \,\middle|\, {}^{x \in \mathrm{supp}(u)}_{q(x) > 0} \right\}, \tag{19}$$

$$\mathrm{supp}(\mathrm{capp}(k, u)) = \bigcup_{x \in \mathrm{supp}(u)} \mathrm{supp}(k(x)). \tag{20}$$

Proof. 非負実数からなる有限和が正であるのは，その和に正の項が含まれる場合に限る．

たとえば，$\mathrm{map}(f, u)(y)$ は，$f(x) = y$ を満たす $x \in \mathrm{supp}(u)$ の重み $u(x)$ の和である．したがって，この値が正であるのは，そのような x が少なくともひとつ存在する場合に限る．よって，$\mathrm{supp}(\mathrm{map}(f, u)) = f[\mathrm{supp}(u)]$ が成り立つ．

weighted，map2 および capp についても，それぞれの定義に現れる有限和に同じ議論を適用する．map2 では，$x \in \mathrm{supp}(u)$ かつ $y \in \mathrm{supp}(v)$ ならば $u(x)v(y) > 0$ である．capp では，$x \in \mathrm{supp}(u)$ ならば $u(x) > 0$ であるため，$u(x)k(x)(y) > 0$ となるのは $k(x)(y) > 0$ の場合に限る．η に関する等式は定義からしたがう．また，$u(x)q(x) > 0$ であるのは $u(x) > 0$ かつ $q(x) > 0$ の場合に限るため，score に関する等式も成り立つ． □

非負性は，重み解釈と台解釈との対応に本質的にかかわる．ブール半環を $\mathbb{B} = \langle \{0, 1\}; \vee, \wedge, 0, 1 \rangle$ とし，

$\chi\colon \mathbb{R}_{\geq 0} \to \{0,1\}$ を $\chi(0) = 0$，$\chi(r) = 1$ $(r > 0)$ と定める．非負実数の間では相殺が生じないため，任意の $a, b \in \mathbb{R}_{\geq 0}$ について，$\chi(a+b) = \chi(a) \vee \chi(b)$ および $\chi(ab) = \chi(a) \wedge \chi(b)$ が成り立ち，また $\chi(0) = 0$，$\chi(1) = 1$ である．したがって，χ は半環準同型であり，台解釈は，非負実数係数を，その係数が零であるか正であるかを表すブール係数へ写す解釈としても捉えられる．負の重みを許す場合には相殺が生じうるため，台だけから和の台を計算することは一般にはできない．

系 4. 式 (9)–(14) によって定めた各重み関数について，

$$\mathrm{weighted}(L),\ \eta_U(a),\ \mathrm{score}(q,u) \in \mathcal{W}_{\mathrm{fin}}(U),$$
$$\mathrm{map}(f,u),\ \mathrm{capp}(k,u) \in \mathcal{W}_{\mathrm{fin}}(V),$$
$$\mathrm{map2}(g,u,v) \in \mathcal{W}_{\mathrm{fin}}(Z).$$

Proof. 命題 3 に現れる各右辺が有限集合であることからしたがう． □

台解釈では，型 `'a repr` を，型 `'a` の要素からなる重複のない有限リストとして実装する．`Support` における各操作は，式 (15)–(20) の右辺を重複のない有限リストとして計算する．命題 3 により，各操作への入力について台解釈の結果が重み解釈の台と一致していれば，その操作の出力についても両者は一致する．

実装では，浮動小数点演算のアンダーフローなどによって，数学上の対応が計算結果に反映されない可能性がある．そこで，`Compile_description` では，初期分布，各遷移分布および各観測分布について，`Weighted` が正の重みを返した要素の集合と，`Support` が列挙した要素の集合とを照合する．

また，同じ記述を `Weighted` と `Support` で別々に評価するため，その記述は純粋であり，同じ引数に対して同じ結果を返すものとする．乱数生成や可変状態の参照のように評価結果が実行時状態に依存する処理，および可変状態の更新や外部入出力のように副作用をともなう処理は用いない．この条件は OCaml の型だけでは保証されないため，モデルを記述する側で満たすものとする．

3.3 HMM の記述と漸化式

初期分布，遷移分布および観測分布を，有限台の非負重み関数に対する代数的操作によって記述する．第 2.1.3 節の Grid Sensors モデルでは，map によって進行方向の分布を次状態の分布へ写し，map2 によって二つの座標に対する観測分布を観測対の分布へまとめる．Grid Sensors モデルの遷移分布と観測分布は，プログラム 4 のように記述される．

```
module Make (A : Finite_weighted.ALGEBRA) = struct
  let transition s =
    let next_mode =
      A.weighted [s.m, 0.8; (s.m + 1) mod 3, 0.2]
    in
    A.map (fun m -> next_state s m) next_mode

  let sensor_channel coordinate =
    if coordinate mod 2 = 0 then A.eta 21
    else
      A.weighted [((coordinate + 1) / 2, 0.3); (21, 0.7)]

  let emission s =
    A.map2 (fun y1 y2 -> { y1; y2 })
      (sensor_channel s.x1) (sensor_channel s.x2)
end
```

プログラム 4　Grid Sensors モデルの遷移分布と観測分布の記述

観測 $y \in Y$ に対し，$b_y\colon X \to \mathbb{R}_{\geq 0}$ を $b_y(x) = B(x,y)$ で定める．また，$k_A\colon X \to \mathcal{W}_{\mathrm{fin}}(X)$ を $k_A(x)(x') = A(x,x')$ で定める．前時刻の状態ごとの重みを $\alpha_{t-1} \in \mathcal{W}_{\mathrm{fin}}(X)$ とする．$\mathrm{capp}(k_A, \alpha_{t-1})(x')$ は，各遷移元 x から状態 x' への寄与 $\alpha_{t-1}(x)A(x,x')$ を加算した重みである．さらに，状態 x' における観測 y_t の重み $B(x', y_t)$ を乗じれば，

$$\alpha_t = \mathrm{score}\big(b_{y_t}, \mathrm{capp}(k_A, \alpha_{t-1})\big)$$

となる．これを点ごとに展開すれば，

$$\alpha_t(x') = B(x', y_t) \sum_{x \in \mathrm{supp}(\alpha_{t-1})} \alpha_{t-1}(x) A(x, x')$$

であり，通常の Forward 漸化式に一致する．

したがって，capp と score の合成は，Forward 法の一時刻分の更新を表す．実装では，モデル記述から計算した遷移関係 E_A と各遷移の重みを用い，観測列の各要素に応じた観測の重みを推論関数の実行時に読み出して，この更新を行うコードを生成する．

4 推論コード生成

本処理系では，HMM の記述を評価して，正の遷移確率をもつ状態対と各遷移の重みを計算し，これらを遷移元ごとに整理した疎表現を構成する．その疎表現にもとづいて，Forward 法と Viterbi 法に共通する漸化式を実行するコードを生成する．半環演算は，BER MetaOCaml のコード値を構築する操作として実装する．

4.1 モデル記述から遷移の疎表現への変換

`Compile_description` では，同じ記述を `Weighted` と `Support` でそれぞれ評価する．`Weighted` による評価では，要素と重みの組からなる有限リストを計算し，`Support` による評価では正の重みをもつ要素を重複なく列挙する（プログラム 6）．初期分布，各遷移分布および各観測分布について，重みの総和と 1 との差の絶対値が 10^{-12} 以下であること，および二つの評価による台が一致することを検査する．その後，状態集合と観測集合の各要素を整数に符号化し，推論コード生成器へ渡す仕様モジュールを構築する．

遷移関係 E_A の各状態対を遷移元ごとの行として整理し，各行を，出次数と設定にもとづいて `Empty`，`Singleton`，`SmallStatic` または `CsrSlice` に分類する（プログラム 7）．空行は `Empty` とする．遷移をひとつだけ含む行は，設定に応じて `Singleton` とするか，`CsrSlice` に含める．残る行のうち，出次数が閾値以下のものを `SmallStatic`，それ以外を `CsrSlice` とする．

`Singleton` に分類した行は，ひとつの遷移先と重みの組として保持し，`SmallStatic` に分類した行は，遷移先と重みの配列として保持する．`CsrSlice` に分類した行の遷移は，遷移先配列と重み配列に遷移元ごとに連続して格納し，各行について，遷移元と格納区間の開始位置・終了位置を配列に保持する．以下，この半開区間をその遷移元の辺区間と呼ぶ．

4.2 遷移ごとの項の計算と集約

コード生成器が要求する半環演算のインタフェースをプログラム 5 に示す．本実装でコード生成器へ渡す具体的なモジュールでは，`t = float code`，`inp = float`，`boolt = bool code` とする．

表 1　出力と構成モジュールの対応

出力	使用する半環	集約時に記録する情報	保存する結果
周辺対数尤度	log-sum-exp	なし	最終時刻の値
全時刻の Forward 値	log-sum-exp	なし	各時刻の値
経路の最大重み	max-plus	なし	最終時刻における最大値
Viterbi 経路	max-plus	遷移元	バックポインタ

コード生成器は，半環演算を表すコード値を構築するモジュール，集約時に遷移元を記録するかどうかを定めるモジュール，および各時刻の計算結果の保存方法を定めるモジュールを引数にとる OCaml ファンクタである．これらのモジュールの組み合わせと出力との対応を表 1 に示す．

四種類の出力を計算するコードは，辺の走査と遷移ごとの項の計算を共有し，使用する半環，集約時に遷移元を記録するかどうか，および保存する結果が異なる．コード生成器は，前時刻における遷移元の値が半環の零元でなければ，その遷移元の辺区間を走査するコードを生成する．各辺について，前時刻における遷移元の値，遷移の重み，および観測の重みを半環の乗法で組み合わせて項を計算し，遷移先ごとに半環の加法で集約する．Viterbi 経路を計算する場合には，最大値を与えた遷移元もあわせて記録する．

4.3 特殊化方式

表 2 に，`CsrSlice` に対する 6 種類の特殊化方式を示す．`packed_dynamic` では，辺区間の端点，遷移先および遷移の重みを推論関数の実行時に配列から読み出し，辺を実行時に反復処理する．`hoist_rows` では，辺区間の開始位置と終了位置をコード生成時に配列から読み出し，その値によって生成コードのループ範囲を定める．`unroll_edges` では，さらに辺を走査するループをコード生成時に展開する．ループを展開したうえで，`embed_dest` は遷移先を，`embed_weights` は遷移の重みを生成コード中に定数として記述し，`exact_embedding` は両方を定数として記述する．なお，`Singleton` と `SmallStatic` に分類した行については，いずれの方式でもコード生成時に辺の走査を展開する．`packed_dynamic` と `exact_embedding` が生成するコードの模式例を，それぞれプログラム 9 お

よびプログラム 10 に示す.

5 実験と考察

本節では，まず第 5.2 節で，生成コードと参照実装とで計算結果が数値的に一致することを確認する．次に，第 5.3 節で，密行列を走査する実装と二種類の CSR 実装とを比較する．第 5.4 節では，直接記述した実装と生成された推論関数とを比較する．さらに，第 5.5 節および第 5.6 節では，辺区間の端点をコード生成時に配列から読み出すこと，辺を走査するループを展開すること，ならびに遷移先と遷移の重みを生成コード中に定数として記述することが，推論の実行時間，コード生成時間，およびコード生成時の peak RSS に与える影響を調べる．性能評価は，対数領域の Forward 法を対象とする．

5.1 実験方法

実験環境

OS	：	macOS
CPU	：	2.0 GHz Intel Core i5
主記憶	：	16 GB
OCaml	：	5.3.0
MetaOCaml	：	BER MetaOCaml N153 (`5.3.0+BER`)

実行時間とコード生成コストの測定方法

推論の実行時間については，各条件に対して乱数シードを 3 通り設定し，各シードについて別個のプロセスを 3 回起動した．各プロセスでは，各実装を 5 回ウォームアップした後，推論の実行時間を 30 回測定した．したがって，各条件・各実装の測定値は 270 個である．図中の点または棒はこれらの中央値を，誤差棒は第 1 四分位数から第 3 四分位数までを表す.

コード生成時間およびコード生成時の peak RSS については，各特殊化方式と出次数の組み合わせごとに，乱数シード 17 を用いて，方式ごとに独立したプロセスを 10 回起動した．各プロセスではウォームアップを行わず，ひとつの特殊化方式についてだけコードを生成した．コード生成時間は，コード値を構築する時間と `Runcode.run` の実行時間との和とする．コード生成時間と peak RSS については，それぞれ 10 回の中央値を用いる．

密行列実装および CSR 参照実装の比較にはネイティブコードを用いた．MetaOCaml によって生成された推論関数を含む比較にはバイトコードを用いた．このため，異なる実験群の絶対実行時間は比較せず，各実験群の内部で速度比を比較する．

入力条件

- 第 5.3 節では，状態数を $|X| = 100$，観測集合の要素数を $|Y| = 16$，遷移行列の零要素の割合を 0.95 とした．観測列の長さは
$$T \in \{100, 500, 1000, 2000, 4000\}$$
とした.
- 第 5.4 節では，状態数 $|X| = 500$，観測集合の要素数 $|Y| = 16$，観測列の長さ $T = 1000$，遷移行列の零要素の割合を 0.95 とした.
- 第 5.5 節および第 5.6 節では，状態数 $|X| = 100$，観測集合の要素数 $|Y| = 16$ とし，各状態の出次数を $d \in \{1, 2, 4, 8, 16, 32\}$ とした．各出次数 d に対する観測列の長さを $T_d = \left\lfloor \frac{5\times 10^6}{|X|d} \right\rfloor$ とした．これにより，時刻 $2, \ldots, T_d$ の Forward 更新において処理する遷移の総数 $|X|d(T_d - 1)$ を約 5×10^6 にそろえた.

第 5.3 節および第 5.4 節で用いる HMM については，乱数から生成した正の値をそれぞれ正規化して，初期分布，遷移分布および観測分布を構成した．観測列は，構成した HMM から標本化した.

出次数を固定して特殊化方式を比較する実験では，状態を $0, \ldots, |X| - 1$ の整数で表し，出次数 d の遷移関係を
$$E_d = \left\{ \big(i, (i + k) \bmod |X|\big) \;\middle|\; \begin{smallmatrix} 0 \le i < |X|, \\ 0 \le k < d \end{smallmatrix} \right\}$$
とした．各状態からの d 個の遷移には，それぞれ遷移確率 $1/d$ を与えた．同じ乱数シードについて，初期分布と観測分布はすべての出次数について共通とした．観測列は，これらの分布の生成とは独立な疑似乱数系列から生成し，同じ乱数シードでは，各出次数条件について同じ疑似乱数系列の先頭 T_d 要素を用いた.

表 2　特殊化方式ごとの辺区間，辺の走査および配列要素の扱い

特殊化方式	辺区間の端点	辺の走査	遷移先	遷移の重み
`packed_dynamic`	実行時に配列から読み出す	実行時に反復	実行時に配列から読み出す	実行時に配列から読み出す
`hoist_rows`	コード生成時に配列から読み出す	実行時に反復	実行時に配列から読み出す	実行時に配列から読み出す
`unroll_edges`	コード生成時に配列から読み出す	コード生成時に展開	実行時に配列から読み出す	実行時に配列から読み出す
`embed_dest`	コード生成時に配列から読み出す	コード生成時に展開	生成コード中の定数	実行時に配列から読み出す
`embed_weights`	コード生成時に配列から読み出す	コード生成時に展開	実行時に配列から読み出す	生成コード中の定数
`exact_embedding`	コード生成時に配列から読み出す	コード生成時に展開	生成コード中の定数	生成コード中の定数

5.2　数値的一致

計算結果が一致することは，まず参照実装どうしで確認し，ついで生成された Forward 推論関数を含む比較でも確認した．まず，確率領域で各時刻の状態に対する重みを正規化する参照実装と，対数領域の密行列実装，遷移元ごとに出辺を走査する CSR 実装（CSR push），および遷移先ごとに入辺を走査する CSR 実装（CSR pull）とを 5 条件で比較した．最終対数尤度の最大絶対差は約 9.09×10^{-13} であった．

次に，生成された Forward 推論関数を，同じ辺の走査を直接記述した参照実装および Bigarray 実装と比較した．3 条件において，最終対数尤度の最大絶対差は約 1.14×10^{-13}，最終状態ベクトルの要素ごとの最大絶対差は約 1.34×10^{-14} であった．辺区間の端点，辺の走査ループ，遷移先および遷移の重みの扱いが異なる各特殊化方式についても，同じ条件で計算した対数尤度は一致した．また，辺区間が空である行や，遷移先が少ない行を含む条件についても，生成実装と直接記述した実装との一致を確認した．

5.3　密行列走査と CSR 走査の比較

全状態対を走査する密行列実装，CSR push 実装（`csr_push`），および CSR pull 実装（`csr_pull`）の実行時間を図 2 に示す．`dense` の実行時間を各 CSR 実装の実行時間で除した速度比は，`csr_push` で約 3.29 から 3.37，`csr_pull` で約 5.91 から 6.15 であった．本実験条件では，正の遷移確率をもつ状態対だけを処理する二つの CSR 実装の実行時間が，密行列実装より一貫して短かった．今回の参照実装では pull 方式の実行時間が短かったが，現行のコード生成器は push 方式を採用している．入辺を走査する pull 方式のバックエンドは今後の課題である．

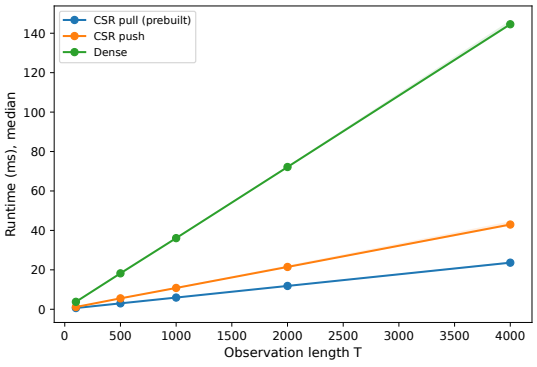

図 2　観測列の長さと Forward 計算時間

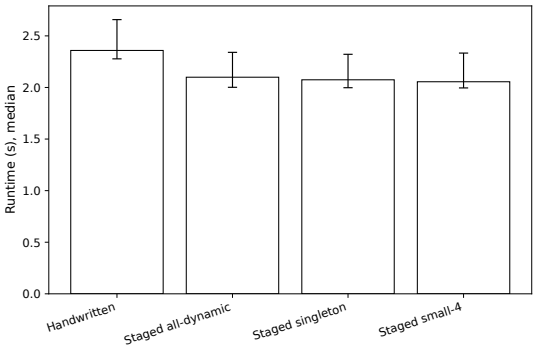

図 3　直接記述した実装と生成された推論関数の Forward 計算時間（棒は中央値，誤差棒は第 1 四分位数から第 3 四分位数）

5.4　直接記述した実装と生成された推論関数の比較

図 3 には，直接記述した実装と，行を分類するための設定を三通りに変えて生成した推論関数との実行時間を示す．直接記述した実装の中央値は約 2.36 秒，生成された推論関数の中央値は約 2.10 秒であった．

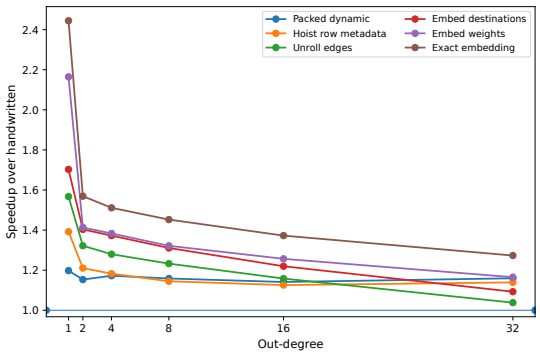

図 4　出次数と実行時間の速度比

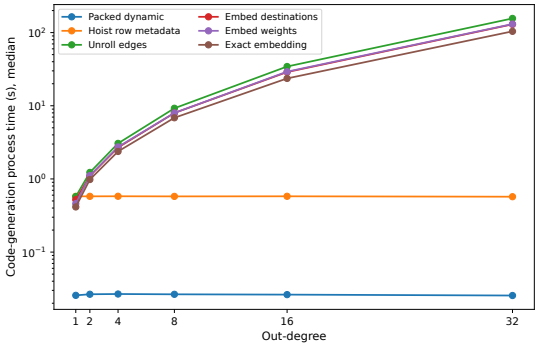

図 5　出次数とコード生成時間

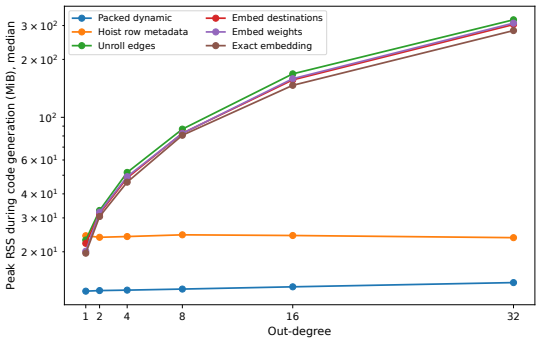

図 6　出次数とコード生成時の peak RSS

この実験条件では，500 個の行がすべて `CsrSlice` に分類されたため，いずれの設定でも CSR 配列を実行時に走査する同じコードが生成された．そのため，三つの生成実装の実行時間はほぼ一致した．生成コードには，使用する半環や項の集約方法を実行時に選択するディスパッチは含まれていない．

5.5　出次数と特殊化方式による実行時間の変化

図 4 の横軸は各行の出次数 d である．縦軸は，直接記述した実装の実行時間を，各方式で生成した推論関数の実行時間で除した速度比である．速度比が 1 より大きい場合には，生成された推論関数の実行時間が，直接記述した実装の実行時間より短い．

CSR 配列を推論関数の実行時に走査する `packed_dynamic` について，速度比は約 1.14 から 1.20 であり，出次数による大きな変化はなかった．一方，遷移先と遷移の重みを生成コード中に定数として記述する `exact_embedding` について，速度比は出次数 1 で約 2.45，出次数 32 で約 1.27 であった．

5.6　コード生成時間・peak RSS・再利用回数

`exact_embedding` のコード生成時間と peak RSS の中央値は，出次数 1 ではそれぞれ約 0.414 秒と 19.7 MiB，出次数 32 では約 104.0 秒と 282.5 MiB であった．

図 5 および図 6 に示すように，`packed_dynamic` と `hoist_rows` では，出次数を増やしても，コード生成時間と peak RSS は大きく変化しなかった．一方，辺を走査するループをコード生成時に展開する四方式では，出次数が大きくなるにつれて，コード生成時間と peak RSS が増加した．また，遷移先と遷移の重みを生成コード中に定数として記述することによる実行時間の短縮幅は，出次数が大きいほど小さかった．

測定値の中央値にもとづく損益分岐回数

`exact_embedding` と `packed_dynamic` のコード生成時間の中央値を，それぞれ G_{e} と G_{p} とする．また，一回の推論の実行時間の中央値を，それぞれ R_{e} と R_{p} とする．$G_{\mathrm{e}} > G_{\mathrm{p}}$ かつ $R_{\mathrm{e}} < R_{\mathrm{p}}$ であるとき，N 回の推論を含む総実行時間について

$$G_{\mathrm{e}} + N R_{\mathrm{e}} \le G_{\mathrm{p}} + N R_{\mathrm{p}}$$

が初めて成り立つ整数 N を，測定値の中央値にもとづく損益分岐回数 N_{be} とする．この値は

$$N_{\mathrm{be}} = \left\lceil \frac{G_{\mathrm{e}} - G_{\mathrm{p}}}{R_{\mathrm{p}} - R_{\mathrm{e}}} \right\rceil$$

である．出次数をそれぞれ $1, 2, 4, 8, 16, 32$ としたとき，N_{be} は同じ順に，おおむね $1, 5, 12, 38, 155, 1305$ 回であった．したがって，本実験条件では，遷移先と遷移の重みを生成コード中に定数として記述したときに総実行時間が短くなるかどうかは，行の出次数と，同じモデルに対して推論関数を実行する回数とに依存した．

6 関連研究

本研究には，有限台の重み関数によるモデル記述と同一の記述に対する複数の解釈，半環上の推論計算，およびモデルごとのコードの特殊化という三つの側面がある．以下では，まずモデル記述と意味解釈に関する研究を整理し，ついで推論計算の代数的抽象化，高水準な仕様からの計算の導出，および多段階計算による特殊化との関係を述べる．

可換半環 S に対して，各集合 U に有限台をもつ S 値関数の集合を対応させる構成はモナドをなし，その要素は S を係数とする有限形式和とみなせる [4]．本論文の $\mathcal{W}_{\mathrm{fin}}(U)$ は，係数半環を $\mathbb{R}_{\geq 0}$ とした場合に対応する．この見方では，η はモナドの単位，map は写像による押し出し（pushforward），capp はクライスリ拡張（Kleisli extension）に対応する．また，各集合 U に総質量が 1 である重み関数の集合を対応させる構成は，有限台の分布モナドをなす [9]．本論文の map2 は，直積上に重み $u(x)v(y)$ をもつ重み関数を構成した後，写像 g によって押し出す操作である．本論文では，これらの構造にもとづく操作のうち，モデル記述に用いるものをひとつのインタフェースにまとめる．

確率プログラミングでは，score と呼ばれる操作により，実行に付された重みに非負係数を乗じて，ソフト制約（soft constraint）や尤度を表す [17]．本論文の $\mathrm{score}(q, u)(x) = u(x)q(x)$ は，有限台の重み関数上で，各要素の重みに非負係数 $q(x)$ を乗じる操作である．また，Coumans と Jacobs が扱う半環準同型による係数変換を本論文の設定に適用すると，非負実数係数を，その係数が零であるか正であるかを表すブール係数へ写すことにより，重み解釈から台解釈への移行を説明できる [4]．命題 3 では，本論文の操作インタフェースについてこの対応を直接証明し，実装では浮動小数点演算によって対応が崩れていないことを，初期分布，各遷移分布および各観測分布について検査する．

同じ高水準な記述に複数の意味を与える実装形式として，tagless-final 形式がある．Carette，Kiselyov，Shan [3] は，共通の操作インタフェースに対して記述された型付き対象言語の項に，評価，コンパイル，部分評価などの異なる解釈を与えた．この形式では，対象言語の項を特定の構文木として固定せず，操作の解釈を替えることによって，同じ記述から異なる処理を構成する．本論文でも，モデルを共通の操作インタフェースに対して記述し，`Weighted` と `Support` によって異なる解釈を与える．

半環を用いて推論や動的計画計算を代数的に扱う研究では，すでに与えられた計算構造上の重みの組み合わせと集約とを，半環演算によって統一的に記述する．Goodman [7] は構文解析の項と推論規則に対する重み計算を，Mohri [15] は重み付き有向グラフ上の経路の重みを対象とし，半環を替えることによって異なる計算を同じ構造上で実行する枠組みを与えた．Aji と McEliece [1] は，分配則にもとづく計算を一般的なメッセージ伝播として定式化し，Baum–Welch 法や Viterbi 法をその具体例として位置づけた．本論文では，この系譜にしたがい，Forward 法と Viterbi 法の更新を半環上の漸化式として共通化する．

高水準な問題記述から効率的な動的計画計算を構成する研究もある．Algebraic Dynamic Programming [6] は，解候補の構造を定める文法と，候補を評価・選択する評価代数（evaluation algebra）とを分離し，同じ探索空間に異なる評価方法を適用できるようにする．また，文法と評価代数から，候補を明示的に列挙しない動的計画計算を構成する．Little，He，Kayas [14] は，組合せ的な解候補を列挙して評価する仕様から出発し，半環多相性と shortcut fusion によって，中間的な候補集合を構築せずに目的の半環上で実行される計算を導出した．これらの研究が高水準な仕様から動的計画計算そのものを構成するのに対し，本論文では Forward 法と Viterbi 法の漸化式を所与とし，モデル記述からその漸化式に必要な疎な遷移構造を

計算する．

確率プログラムから効率的な推論計算を生成する研究では，高水準なモデル記述または推論記述を別の計算表現へ変換する．Walia ら [18] は，配列を用いた高水準な推論アルゴリズムをループからなるコードへ変換し，領域固有の不変条件を用いた最適化と，各実行に応じた特殊化を行った．Holtzen，Van den Broeck，Millstein [8] の Dice は，離散確率プログラムを重み付きモデル計数（weighted model counting）へ合成的に変換し，プログラムが表す構造を厳密推論に利用する．これらに対し，本論文では一般の確率プログラムや推論アルゴリズムをコンパイルするのではなく，有限台の重み関数による HMM の記述から遷移関係 E_A を計算し，HMM に固有の漸化式を実行するコードを生成する．

コード生成時に利用できる情報にもとづいて数値計算のコードを特殊化する研究として，Kiselyov [11] は，HMM の計算から抽象化された行列ベクトル積を扱っている．そこでは，行列をコード生成時に利用でき，ベクトルを生成されたコードの実行時に与える場合について，疎行列による表現，代数的な簡約，行を走査するループの展開，および各行に含まれる非零要素の個数に応じた内側のループの選択的な展開を，MetaOCaml 上の独立したモジュールとして構成している．本論文の特殊化方式も，行列の構造をコード生成時に利用してループや配列参照を特殊化する点でこの研究と共通する．ただし，Kiselyov の処理がすでに与えられた行列から始まるのに対し，本論文では，同じモデルの記述に対する重み解釈と台解釈から遷移の疎表現を構成する処理も対象とする．

以上より，本研究の特徴は，有限台の重み関数によるモデル記述，重み解釈と台解釈との照合，遷移関係 E_A の抽出と疎形式への格納，半環上の推論計算，および多段階計算によるモデルごとの特殊化を，一連の処理として構成する点にある．また，特殊化の効果を，推論の実行時間だけでなく，コード生成時間，コード生成時の peak RSS，および測定値の中央値にもとづく損益分岐回数によって評価する．

7　まとめと今後の課題

本論文では，任意の集合上の有限台の非負重み関数を構成・変換する代数的操作と，その重み解釈および台解釈を定めた．これらの操作を用いて HMM の初期分布，遷移分布および観測分布を記述し，各操作について，台解釈の結果が重み解釈によって定まる重み関数の台と一致することを示した．実装では，初期分布，各遷移分布および各観測分布について二つの評価結果を照合したうえで，遷移関係 E_A を計算し，各遷移に重みを対応させて遷移元ごとに整理した疎表現を構成した．

この疎表現にもとづき，Forward 法と Viterbi 法に共通する辺の走査と，遷移ごとの項の計算とを行うコード生成器を実装した．使用する半環，集約時に遷移元を記録するかどうか，および各時刻の計算結果を保存するかどうかを，モジュールの組み合わせによって切り替える．これにより，周辺対数尤度，全時刻の Forward 値，経路の最大重み，および Viterbi 経路を計算するコードを，同じ走査処理から生成する．

また，`CsrSlice` に分類した行について，辺区間の端点をコード生成時に配列から読み出すか，辺を走査するループを展開するか，遷移先と遷移の重みを生成コード中に定数として記述するかを，個別に選択できるようにした．本実験条件では，CSR 配列を推論関数の実行時に走査する `packed_dynamic` について，直接記述した実装に対する速度比は約 1.14 から 1.20 であった．`exact_embedding` では，出次数が小さい条件で推論時間がさらに短くなったが，出次数の増加にともなって短縮幅が小さくなり，コード生成時間とコード生成時の peak RSS が増加した．

本論文で用いた代数的操作は，任意の集合 U 上の有限台の非負重み関数に対して定義される．重み解釈では有限個の要素とその重みを保持し，台解釈では重複のない有限個の要素を保持するため，操作の数学的な定義は，U が無限集合である場合にも適用できる．また，系 4 を各時刻の更新に帰納的に適用すれば，初期状態に対する重み関数が有限台をもち，各状態から正の重みで到達できる状態が有限個であれば，各時刻の状態に対する重み関数も有限台をもつ．

一方，現在の HMM を記述するためのインタフェースとコード生成器は，有限な状態集合と観測集合の全要素を，コード生成に先立って整数へ符号化する．無限の状態集合をもつ HMM を扱うには，全状態を事前に列挙せず，初期分布の有限台から到達可能な状態を順次列挙し，各状態に対する有限台の遷移分布と，入力された観測に対する各状態の重みとを推論の進行に応じて計算できるインタフェースが必要となる．

さらに，そのようなインタフェースのもとでモデル固有の遷移構造を生成コードへ反映するには，状態を列挙する過程で各状態に整数を割り当て，正の重みをもつ遷移を疎形式で格納し，列挙にともなって追加される有限個の遷移をコード生成の対象に加える仕組みが必要である．これらの拡張は今後の課題である．

謝辞

本研究を進めるにあたり，Oleg Kiselyov 先生には，研究方針の検討から実装，評価，原稿の取りまとめまで，長いあいだ折に触れてご指導いただきました．議論や実装が行き詰まったときには，問題を具体的に捉え直すための助言をいただき，そのたびに研究を前へ進めることができました．励ましや細やかなお心遣いも含め，これまでいただいたご指導に心より感謝申し上げます．

また，住井英二郎先生，松田一孝先生，西田雄気先生，ならびに研究室の皆様には，ゼミや日々の議論を通じて，多くのご意見と助言をいただきました．この場を借りて感謝いたします．

生成 AI の利用について　本研究では，実験作業の一部におけるプログラム作成の補助，関連する評価方法を調査する際の情報収集補助，および図の作成方法の点検のため ChatGPT 上の GPT-5.5 Thinking および GPT-5.6 Sol（推論レベル High）を利用した．ChatGPT が提示したプログラムは著者が内容を確認し，修正および検証を行った上で使用した．

参 考 文 献

[1] Aji, S. and McEliece, R.: The generalized distributive law, *IEEE Transactions on Information Theory*, Vol. 46, No. 2(2000), pp. 325–343.

[2] Carette, J. and Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code, *Science of Computer Programming*, Vol. 76, No. 5(2011), pp. 349–375. Special Issue on Generative Programming and Component Engineering (Selected Papers from GPCE 2004/2005).

[3] Carette, J., Kiselyov, O., and Shan, C.-c.: Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages, *Journal of Functional Programming*, Vol. 19, No. 5(2009), pp. 509–543.

[4] Coumans, D. and Jacobs, B.: Scalars, Monads, and Categories, *Quantum Physics and Linguistics: A Compositional, Diagrammatic Discourse*, Heunen, C., Sadrzadeh, M., and Grefenstette, E.(eds.), Oxford University Press, 2013, pp. 184–216.

[5] Droste, M., Kuich, W., and Vogler, H.(eds.): *Handbook of Weighted Automata*, Springer Publishing Company, Incorporated, 1st edition, 2009.

[6] Giegerich, R., Meyer, C., and Steffen, P.: A discipline of dynamic programming over sequence data, *Science of Computer Programming*, Vol. 51, No. 3(2004), pp. 215–263.

[7] Goodman, J.: Semiring parsing, *Comput. Linguist.*, Vol. 25, No. 4(1999), pp. 573–606.

[8] Holtzen, S., Van den Broeck, G., and Millstein, T.: Scaling Exact Inference for Discrete Probabilistic Programs, *Proceedings of the ACM on Programming Languages*, Vol. 4, No. OOPSLA(2020), pp. 140:1–140:31.

[9] Jacobs, B.: Probabilities, Distribution Monads, and Convex Categories, *Theoretical Computer Science*, Vol. 412, No. 28(2011), pp. 3323–3336.

[10] Jones, N. D., Gomard, C. K., and Sestoft, P.: *Partial evaluation and automatic program generation*, Prentice-Hall, Inc., USA, 1993.

[11] Kiselyov, O.: Reconciling Abstraction with High Performance: A MetaOCaml Approach, *Foundations and Trends in Programming Languages*, Vol. 5, No. 1(2018), pp. 1–101.

[12] Kiselyov, O.: MetaOCaml: ten years later – System description, *Science of Computer Programming*, Vol. 250(2026), pp. 103397.

[13] Lall, S.: EE365: Hidden Markov Models, Lecture Notes, Stanford University, `https://stanford.edu/class/ee365/lectures/hmm.pdf`, 2014.

[14] Little, M., He, X., and Kayas, U.: Polymorphic dynamic programming by algebraic shortcut fusion, *Formal Aspects of Computing*, Vol. 36, No. 2(2024), pp. 1–32.

[15] Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems, *J. Autom. Lang. Comb.*, Vol. 7, No. 3(2002), pp. 321–350.

[16] Rabiner, L.: A tutorial on hidden Markov models and selected applications in speech recognition, *Proceedings of the IEEE*, Vol. 77, No. 2(1989), pp. 257–286.

[17] Staton, S., Yang, H., Wood, F., Heunen, C.,

and Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, New York, NY, USA, Association for Computing Machinery, 2016, pp. 525–534.

[18] Walia, R., Narayanan, P., Carette, J., Tobin-Hochstadt, S., and Shan, C.-c.: From high-level inference algorithms to efficient code, *Proc. ACM Program. Lang.*, Vol. 3, No. ICFP(2019).

A 半環演算のコード生成インタフェース

```
module type SEMIRING = sig
  type t         (* float code *)
  type inp       (* float *)
  type boolt     (* bool code *)

  val zero : t
  val one : t
  val zero_inp : inp
  val one_inp : inp
  val add : t -> t -> t
  val mul : t -> t -> t
  val lift : inp -> t
  val is_zero : t -> boolt
  val sum_list : t list -> t
  val is_zero_inp : inp -> bool
  val is_one_inp : inp -> bool
end
```

プログラム 5　半環演算のコード生成インタフェース

B 内部実装の主要な箇所

B.1 モデル記述に対する評価

`Weighted` と `Support` による評価と検査の主要部分を簡略化してプログラム 6 に示す．遷移分布および観測分布についても，各状態に対して同じ検査を行う．

```
module Compile_description (D : DESCRIPTION) = struct
  module W = D.Make(Weighted)
  module S = D.Make(Support)

  let require_positive_elements (w_list : ('a * float) list)
      : 'a list =
    w_list
    |> List.filter (fun (_, w) -> w > 0.0)
    |> List.map fst

  let check_support_agreement (name : string) (w_list : ('a
      * float) list) (s_list : 'a list) =
    let w_set =
      require_positive_elements w_list |> List.sort_uniq
      compare
    in
    let s_set = s_list |> List.sort_uniq compare in
    if w_set <> s_set then
      failwith (name ^ " support mismatch")

  let check_mass (name : string) (w_list : ('a * float) list
      ) =
    let mass = List.fold_left (fun acc (_, w) -> acc +. w)
      0.0 w_list in
    if abs_float (mass -. 1.0) > 1e-12 then
      failwith (name ^ " mass is not 1.0")

  let check_distribution name (w_list, s_list) =
    check_mass name w_list;
    check_support_agreement name w_list s_list

  let compile () =
    check_distribution "initial" (W.initial, S.initial);
    (* 検査済みの分布から仕様を構築する HMM *)
end
```

プログラム 6　モデル記述から計算した重みと，台解釈が列挙した要素とを照合する処理の概略

B.2 遷移元ごとの分類

行の分類処理をプログラム 7 に示す．

```
match weighted_row with
| [] ->
    Empty
| [dst, p] when static_singleton ->
    Singleton (dst, weight_of_prob p)
| [_] ->
    append_loop_row src weighted_row
| _ when List.length weighted_row <= small_threshold ->
    SmallStatic
      (weighted_row
       |> List.map (fun (dst, p) ->
            dst, weight_of_prob p)
       |> Array.of_list)
| _ ->
    append_loop_row src weighted_row
```

プログラム 7　遷移先の個数にもとづく行の分類

B.3 最大値と遷移元の記録

```
let update buf backptrs target weight src =
  let old_val = L.get buf target in
  .<
    let w = .~weight in
    let o = .~old_val in
    if .~(S.better .<w>. .<o>.) then begin
      .~(L.set buf target .<w>.);
      Array1.unsafe_set .~backptrs .~target .~src
    end
  >.
```

プログラム 8　新たに計算した値が現在値より大きい場合の値と遷移元の記録

C 一時刻分の計算を行う生成コードの例

特殊化の程度の違いを示すため，実際の出力を簡略化した模式的なコード断片を以下に示す．

C.1 `packed_dynamic` の場合（実行時ループ）

```
for k = start_idx to stop_idx - 1 do
  let dst = col_idx.(k) in
  let w   = weights.(k) in
  next.(dst) <- logaddexp next.(dst) (src_val +. w +. obs.(
      dst))
done
```

プログラム 9　推論関数の実行時に辺区間を反復するコード

C.2 `exact_embedding` の場合（$d = 2$，ループ展開あり）

```
(* 遷移先と重みを定数として直接記述 *)
next.(5) <- logaddexp next.(5) (src_val +. (-0.223) +. obs
      .(5));
next.(8) <- logaddexp next.(8) (src_val +. (-1.609) +. obs
      .(8))
```

プログラム 10　遷移先と重みを定数として含む生成コード

D OCaml 実装と C 実装の参考比較

密行列による確率領域の Forward 法について，直接記述した OCaml ネイティブ実装と 2 種類の C 実装の実行時間を比較した．

現在の状態に対する重みを $a = (a_0, \ldots, a_{n-1})$ とする．いずれの実装も，Forward 更新 $a'_j = B(j, o_t) \sum_{i=0}^{n-1} a_i A(i, j)$ を計算した後，状態に対する重みを正規化する．遷移確率と観測確率を，それぞれ次の位置に格納した．

$$\texttt{trans_prob.(i * n + j)} = A(i, j),$$
$$\texttt{emit_prob.(j * n_obs + o)} = B(j, o).$$

OCaml 実装は通常の浮動小数点配列を，C 実装は C レイアウトの一次元 Bigarray を入力とする．

C 実装では，この Forward 更新を二つのループ順で実装した．一方はプログラム 1 と同様に，各時刻について，外側のループで遷移先の状態 j を走査し，内側のループで遷移元の状態 i を走査して，$\sum_i a_i A(i, j)$ を計算する．もう一方は，次時刻の配列を 0 で初期化した後，外側のループで遷移元の状態 i を走査し，内側のループで遷移先の状態 j を走査して，$a_i A(i, j)$ を各遷移先について累積する．最後に，各遷移先について観測確率を乗じる．

遷移行列は `trans_prob.(i * n + j)` に行優先で格納されている．このため，外側のループで遷移先の状態 j を走査する実装では，内側のループが一回進むたびに，遷移行列を n 要素おきに参照することになる．これに対して，外側のループで遷移元の状態 i を走査する実装では，i を固定したまま，`trans_prob.(i * n + j)` を $j = 0, \ldots, n-1$ の順に連続して参照する．

C 用 Bigarray への入力変換は測定前に一度だけ行い，測定時間には含めなかった．一回の Forward 推論につき C の外部関数を一回呼び出し，C 関数内で観測列全体を処理した．したがって，外部関数呼出しは各時刻には行われない．測定時間には，C カーネルの計算に加え，呼出しごとの作業配列の確保と解放，Bigarray の次元検査，および最終時刻の状態分布を OCaml 配列へコピーする処理を含む．入力モデルの生成と両実装の出力の一致検査も，測定時間には含めなかった．OCaml 側の入力検査も測定前に行い，測定中は `Array.unsafe_get` および `Array.unsafe_set` を用いる検査済みの実装を使用した．

比較条件を表 3 に示す．図 7 および図 8 の各比較について，3 つの乱数シードを用い，各シードにつき

表 3　OCaml 実装と C 実装の比較条件

条件	$\lvert X\rvert$	$\lvert Y\rvert$	T	反復数
small	10	16	100	1000
medium	50	16	1000	10
large	100	16	1000	5

3 つの独立したプロセスを実行した．各プロセスでは 5 回のウォームアップ後に 30 回測定したため，各条件について 270 組の対応する測定値を得た．短時間の処理では，ひとつの観測列に対する Forward 推論を表中の反復数だけ繰り返し，その合計時間を反復数で割った値を一回当たりの実行時間とした．測定順序による偏りを抑えるため，OCaml 実装と対象とする C 実装を測定する順序を反復ごとに入れ替えた．

OCaml コードは OCaml 5.3.0 の `ocamlopt -O2` によってネイティブコードへコンパイルした．C コードは Apple Clang 12.0.5 を用い，`-W-Wall-Wextra-O3-m64-march=native-fno-trapping-math-fomit-frame-pointer-std=c11` を指定してコンパイルした．C カーネルでは，各入力配列および出力配列が別領域にあることを `restrict` によってコンパイラへ与えた．また，各時刻の状態に対する重みを正規化する処理をインライン関数として実装し，状態に対する非負の重みの総和が 0 の場合には対数正規化定数として $-\infty$ を返し，正の場合には各要素をその総和で除した．

同じ乱数シードを用い，同じプロセスの同じ測定回で得られた OCaml 実装と C 実装の測定値を組にし，各組の速度比を $R = t_{\mathrm{OCaml}}/t_{\mathrm{C}}$ とした．速度比が 1 より大きい場合には，C 実装の実行時間が OCaml ネイティブコード実装より短いことを表す．

図 7 は，外側のループで遷移先の状態を走査する C 実装との比較結果である．点は 270 組の速度比の中央値，誤差棒は第 1 四分位数から第 3 四分位数までを表す．速度比の中央値は，small，medium，large の各条件について，それぞれ約 2.18，1.91，1.59 であった．

図 8 は，外側のループで遷移元の状態を走査する C 実装との比較結果である．測定条件と方法は図 7 と同じである．速度比の中央値は，small，medium，large の各条件について，それぞれ約 1.77，6.30，7.14 であった．

図 7 と図 8 を比較すると，small では二つの C 実装で得られた速度比の差が medium および large より小さい．一方，medium および large では，遷移元の状態を外側のループで走査する C 実装との速度比

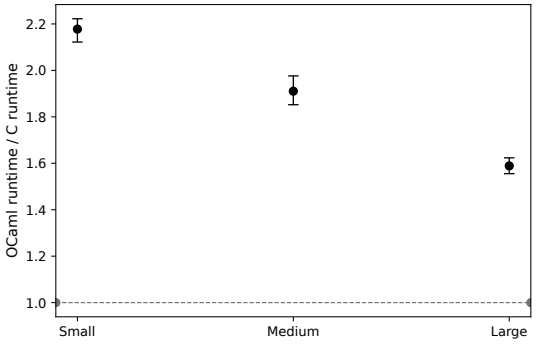

図 7　密行列を用いる確率領域 Forward 法における OCaml ネイティブ実装と，外側のループで遷移先の状態を走査する C 実装の速度比
（点は中央値，誤差棒は第 1 四分位数から第 3 四分位数，水平線は速度比 1）

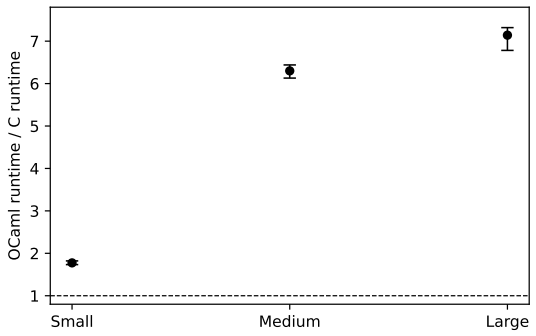

図 8　密行列を用いる確率領域 Forward 法における OCaml ネイティブ実装と，外側のループで遷移元の状態を走査する C 実装の速度比
（点は中央値，誤差棒は第 1 四分位数から第 3 四分位数，水平線は速度比 1）

が大きかった．このように，同じ Forward 更新式を計算する C 実装でも，ループ順によって OCaml 実装との速度比は大きく変わりうる．したがって，これらの速度比は，今回用いた実装どうしを比較して得られた値であり，OCaml と C という言語の一般的な性能の差や，C で実装した場合に得られる性能の上限とみなすことはできない．

なお，第 5.4 節の比較では，遷移の疎表現を用いて対数領域で計算しており，本節の実装とは計算方法とデータ表現が異なる．

E　出次数ごとの測定結果

図 4，図 5 および図 6 の基礎となる測定値を，以下の表に示す．測定方法は第 5.1 節に示したとおりである．

表 4　出次数ごとの実行時間（秒）

実装	$d=1$	$d=2$	$d=4$	$d=8$	$d=16$	$d=32$
`handwritten`	0.971	1.017	1.060	1.036	1.011	1.012
`packed_dynamic`	0.812	0.878	0.903	0.890	0.889	0.875
`hoist_rows`	0.696	0.840	0.895	0.901	0.899	0.892
`unroll_edges`	0.620	0.765	0.826	0.837	0.874	0.975
`embed_dest`	0.571	0.726	0.769	0.787	0.829	0.927
`embed_weights`	0.449	0.716	0.766	0.781	0.803	0.867
`exact_embedding`	0.397	0.649	0.701	0.710	0.737	0.796

表 5　各特殊化方式のコード生成時間（秒）

実装	$d=1$	$d=2$	$d=4$	$d=8$	$d=16$	$d=32$
`packed_dynamic`	0.026	0.027	0.027	0.026	0.026	0.026
`hoist_rows`	0.579	0.578	0.580	0.577	0.579	0.572
`unroll_edges`	0.574	1.227	3.059	9.242	34.39	156.2
`embed_dest`	0.521	1.097	2.687	7.969	28.86	129.7
`embed_weights`	0.459	1.110	2.738	7.986	29.26	130.9
`exact_embedding`	0.414	0.982	2.375	6.852	23.64	104.0

表 6　各特殊化方式のコード生成時の peak RSS（MiB）

実装	$d=1$	$d=2$	$d=4$	$d=8$	$d=16$	$d=32$
`packed_dynamic`	12.5	12.6	12.6	12.8	13.1	13.8
`hoist_rows`	24.2	23.8	24.0	24.5	24.3	23.7
`unroll_edges`	23.0	32.8	51.7	86.8	168.2	321.4
`embed_dest`	22.1	31.6	48.4	82.6	156.6	303.6
`embed_weights`	20.1	32.1	49.2	82.5	158.8	308.7
`exact_embedding`	19.7	30.5	46.0	80.7	146.7	282.5

表 7　`exact_embedding` を `packed_dynamic` と比較した損益分岐回数

	$d=1$	$d=2$	$d=4$	$d=8$	$d=16$	$d=32$
N_{be}	1	5	12	38	155	1305