Vol.2012-GI-27 No.12
2012/3/2

完全プレイのためのデータベースのサイズの削減

田　中　哲　朗[†1]

完全に解析を終わったゲームで完全プレイをおこなうためのデータベースは，完全ハッシュ関数とウェーブレット木を用いて，小さな記憶容量で実現できる．本研究では局面のノードを一段展開することで，更に記憶容量を減らす方法を提案する．これにより,「どうぶつしょうぎ」の完全プレイのデータベースを 54.8MB の記憶容量で実現できた．これは，単純な実装と比較して 1/15 の容量となっている．

Reducing database size for perfect play

Tetsuro Tanaka[†1]

With small memory size, a database for perfect play of a strongly solved game can be constructed by a perfect hash function and a wavelet tree of ply numbers. In this article, we propose a method to reduce the database size by expanding the node of positions by one ply. As a result, the database size for “Dobutsu shogi” perfect play is reduced to 54.8MB, which is roughly one fifteenth of that of simple implementation.

1. はじめに

個人用 PC で数十 GB 単位のメモリと数 GHz のマルチコア CPU が利用できる時代になり，状態数の少ないゲームのいくつかは，後退解析により完全に解くことができるようになった．

完全に解析を終わったゲームにおいて，勝ち局面では最短手数の勝ちに繋がる手，負け局面では負けるまでの手数を最大にする手，引き分け局面では引き分けを維持する手を指すことを本稿では完全プレイと定義する．これは，局面から勝ち負けに関する情報を含めた手数へ写像があれば実現できる．

この写像は，局面からインデックスへの変換をおこなう完全ハッシュ関数のデータベースと，インデックスから手数への変換をおこなうデータベースがあれば実現できる．近年，これらのデータベースをより少ない記憶容量で表現する手法が開発されたので，それらを用いることで，単純な表現法よりもずっと小さい容量で実現できた．

また，局面のノードを一段展開することで更にデータベースの圧縮が可能になる.「どうぶつしょうぎ」の完全プレイのためのデータベースは単純な実装で 895MB のサイズだったが，これらの組み合わせにより，54.8MB まで減らすことができた．これは，単純な実装と比較して 1/16 の容量となっている．これにより，携帯用ゲーム機やスマートフォンで「どうぶつしょうぎ」の完全プレイをおこなうプログラムの作成も可能になったと考えられる．

2. 「どうぶつしょうぎ」の完全解析

本節では，本稿で対象とするゲームである「どうぶつしょうぎ」の完全解析 (ゲーム中に現れるすべての局面の勝敗を決定する) に関して説明する．

「どうぶつしょうぎ」は 2008 年に女流棋士の北尾まどか初段によって考案されたボードゲームである．このゲームは 2009 に後手 78 手勝ちであることが証明され，初期局面から開始して「手番のプレイヤーが相手のライオンを捕まえられる時は必ず捕まえて勝つ」,「ライオンがトライした時はゲームを終了する」として到達可能な局面（以下では単に初期局面から到達可能な局面と呼ぶ）すべてに関して，手番の勝ちか，負けか，引き分けかが判明している[?)]．

「どうぶつしょうぎ」の 8 個の駒を盤面上，および持ち駒に配置する組み合わせの数を数えたものを表 1 に示す．　ただし，左右を入れ替えた局面は同一と見なせるのにその重複を除いていないし，初期局面から到達可能な局面以外の局面も含んでいる．

それぞれのマスは空白か各プレイヤの 5 種類の駒が存在するかの 11 状態なので 4bit で表現可能である．盤面のマスの数は $3 = 12$ 個なので $4 \times 12 = 48$bit で盤面の状態は表せる．持ち駒は各プレイヤ 3 種類で駒の数は $0, 1, 2$ の 3 種類なのでそれぞれ 2bit の計 $6 \times 2 = 12$bit で表せる．手番は固定しても一般性を失わないので，計 60bit で盤面が表現できる．

図 1 のように左右を入れ替えた局面は同じと見なすことができるが，これは 64bit 整数として表現した時に小さい方の値で正規化することで自然に実現できる．

初期局面から開始して「手番のプレイヤーが相手のライオンを捕まえられる時は必ず捕ま

†1 東京大学情報基盤センター
Information Technology Center, The University of Tokyo
ktanaka@tanaka.ecc.u-tokyo.ac.jp

表 1 持ち駒総数ごとの局面数

持ち駒総数	局面数
0	638,668,800
1	638,668,800
2	242,161,920
3	44,098,560
4	4,134,240
5	190,080
6	3,564
Sum	1,567,925,964

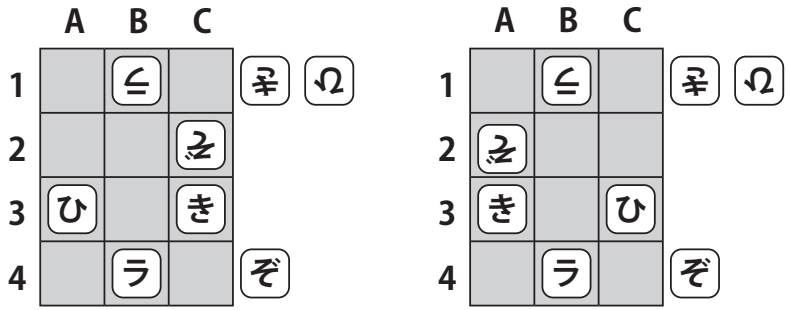

図 1 左右入れ替えると等しくなる局面

えて勝つ」,「ライオンがトライした時はゲームを終了する」として到達可能な局面（以下では単に初期局面から到達可能な局面と呼ぶ）の局面数のうち末端局面を除いた局面数は 99,485,568 となった．後退解析の結果末端局面を除いた 99,485,568 局面のうち，手番の勝ちは 56,474,473 局面，引き分けは 2,682,700 局面，負けは 40,328,395 局面と分かった．

また，後退解析の結果，勝ち負けに関しては局面の手数も求まっている．以下では，対象とする局面全体の集合を S と呼び, 局面 $s(\in S)$ から引き分け局面の手数 0，勝ち局面の手数を $1, 3, \ldots$, 負け局面の手数を負の値を用いて $-2, -4, -6, \ldots$ と表現した手数を $wl(s)$ と

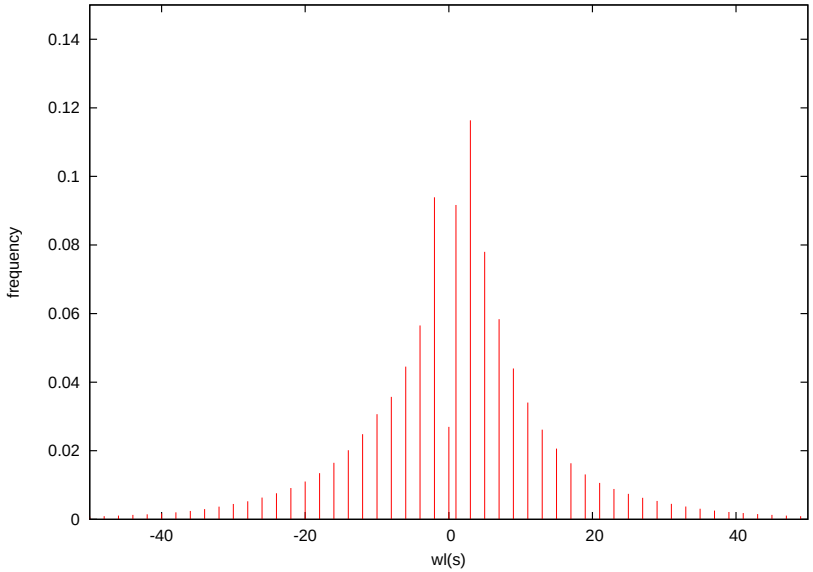

図 2 wl(s) の分布

呼ぶことにする．$wl(S)$ の分布は図 2 のようになっている[⋆1]．

負け局面の最大手数は 172, 勝ち局面の最大手数は 173 だが，負け局面の手数は偶数，勝ち局面の手数は奇数なので，手数として現れる数字は 174 通りであり，1 バイトで表現可能である．また，出現頻度は，勝ち局面と負け局面共に手数が大きくなるほど減る傾向がある[⋆2]．

3. 解析結果を用いた完全プレイ

「どうぶつしょうぎ」のように解析が終わったゲームで，対象とするすべての局面 s に対して，$wl(s)$ が十分短い時間で求められると仮定する．この時,「完全プレイ」するプログラムを作成することを考える.「完全プレイ」として必要な要件は一般的には,

⋆1 負け局面の最大手数は 172, 勝ち局面の最大手数は 173 だが，図は-50 から 50 のみを切り出している．
⋆2 勝ち手数に関しては 1 よりも 3 の頻度が高くなっているが，勝ち手数 1 のうち，敵ライオンをキャッチできる局面は含まれていないためであると考えられる．

- 手番の勝ち局面からスタートしてプレイをした時には必ず勝つ
- 手番の引き分け局面からスタートしてプレイをして負けることはない.

と考えられるが，本稿では，

- 勝ち局面 s からスタートした時は必ず $wl(s)$ 手以下で勝つ.
- 負け局面 s からスタートした時は $-wl(s)$ 手未満には負けない.

という条件も課したものと定義する.

なお，以下ではある局面 s で可能な合法手の集合を $moves(s)$，ある局面 s で，その局面での合法手 m を指した時に遷移する局面 (手番のプレイヤで正規化したもの) を $apply(s,m)$ と定義する.

自明な勝ちがない局面 s を与えられた時に「完全プレイ」をおこなうプログラムは以下のアルゴリズムで実現できる.

(1) $wl(s) > 0$ の時は，$m \in moves(s)$ かつ $wl(apply(s,m)) = -(wl(s)-1)$ となる m が一つ以上存在するはずなので，その m を指す.
(2) $wl(s) < -2$ の時は，$m \in moves(s)$ かつ $wl(apply(s,m)) = -wl(s)-1$ となる m が一つ以上存在するはずなので，その m を指す.
(3) $wl(s) = -2$ の時は，何を指しても次に相手が 1 手で勝つので，任意の合法手を指す.
(4) $wl(s) = 0$ の時は，$m \in moves(s)$ かつ $wl(apply(s,m)) = 0$ となる m が一つ以上存在するはずなので，その m を指す.

写像 $wl(s)$ は以下のように単純に実装することができる．なお，以下では $N = |S|$ とする.

- 局面 $s(\in S)$ を $l$bit で表現できるものとする．局面全体の集合 S を大きさ N の配列 sa で表す.
- 配列 sa 中には局面をソートして入れておく．これにより，局面から $0..N-1$ の番号付けをおこなう関数 $index(s)$ は，二分探索を用いれば $log_2 N$ 回のアクセスで実現できる.
- 手数を m bit で表現できるとして，各要素が m ビットでサイズが N の配列 cs を用意する．すべての $s(\in S)$ について，$cs[index(s)] = wl(s)$ となるように，cs の中身を初期化しておく.

この方法で実装すると，$wl(s)$ は 1 回の参照あたり $O(logN)$ の時間で実行できる．一方，実装に要するメモリ容量は $(l+m) \times N$ bit となる.「どうぶつしょうぎ」のデータベースでは，$l = 60, m = 8, N = 99485568$ なので，$6765018624bit \simeq 846MB$ となる.

4. データベースサイズの削減

単純な実装によるデータベースの 846MB というサイズは，PC 等では問題にならないサイズだが，携帯用ゲーム機やスマートフォンで実現するには大きすぎる．そこで，データベースサイズの削減を検討する.

4.1 完全ハッシュ関数

データベースは，局面 s から $wl(s)$ への写像 (map) と考えれば良い．しかし，一般の map のようなデータベースに登録されていないキーが入力として入ってきた時に，未登録であることを検出する必要はなく，登録されているキーしか入力されない仮定できる．このようなケースでは完全ハッシュ関数 (PHF, perfect hash function) を用いることができる.

N 個のキーから $M(M \geq N)$ として，$0..(M-1)$ の重複のないインデックスを得る関数を完全ハッシュ関数と呼び，特に $M = N$ の場合は，最小完全ハッシュ関数 (MPHF,minimal perfect hash function) と呼ぶ．理論的には，最小完全ハッシュ関数の表現には $1.44N$ bit が必要なことが知られているが[1].

大きな N に関して実用的な完全ハッシュ関数を作成する研究は近年，大きく発展を遂げている．文献 2) の手法 (以下では BDZ) では漸近的に MPH を $2.6N$ bit, $M = 1.23N$ の時に PH を $1.95N$ bit で表現できる．また，これの改良版の文献 3) 中のアルゴリズム (以下では CHD) は $M = 2N$ の PH を $0.67N$bit, $M = 1.23N$ の PH を $1.4N$bit で表現できるとしている.

特に，CHD では loadfactor(N/M) として，50% ～ 99% の任意の値を指定することができる，MPHF と比較すると，PHF の方がより少ないビット数で実現できるので，この時点では PHF の方を用いることも可能である.

4.2 手数の表現

完全ハッシュ関数で得られたインデックスから手数を得るためのデータベースは，8 ビット長の配列を利用すれば $8N$ bit で実現できる．しかし，図 2 のように，$w(s)$ の出現頻度に大きな偏りがあることを利用するとサイズを減らせる可能性がある．たとえば,「どうぶつしょうぎ」の $w(s)$ の出現頻度を元に経験的エントロピー H_0 を求めると 4.76 bit となるので，理論的には $4.76N$ bit まで減らせる可能性がある.

逐次アクセスをおこなう場合には，Huffman coding[5] を用いて符号化することで $(H_0+1)N$ bit 以下で表現することができることは知られている．また，Arithmetic coding[6] などを

用いると，漸近的に H_0N bit に近づけることは可能である．[⋆1]

しかし，完全プレイをおこなう場合には，逐次アクセスではなくランダムアクセスをおこなう必要がある．大規模データに対して圧縮したままでランダムアクセスをおこなうための実用的な方法は，最近になっていくつも開発された．

その中の一つが簡潔データ構造[7] の一つである索引付きのビットベクトルを用いて，ハフマン符号化に対応したウェーブレット木を作成する方法である．これを用いると，逐次アクセス用のハフマン符号化に rank 操作のための補助的な記憶容量を少量加えるだけで，ランダムアクセスが可能になる．

インデックスを得る時に PHF を用いた場合には，キーが対応しないインデックスに対応するテーブルには，もっとも頻度が高い値を入れることで H_0 を最小にすることができる．一般には PHF の load factor の値が小さいほど (M が大きいほど), より少ないビット数で PHF を表現できるが，その分ウェーブレット木のサイズは大きくなるので，どの位の load factor の値が良いかは実験が必要になる．

4.3 ゲームの性質を利用した削減

$wl(s)$ から負け局面の手数だけを切り出した $l(s)$ を以下のように定義する．

$$l(s) = ws(s), if ws(s) < 0$$
$$l(s) = 0, otherwise$$

局面 s からノードを一段だけ展開することを許すと，$l(s)$ があれば，$wl(s)$ を得ることができることを示す．

- s が末端局面ならば終了
- $l(s) < 0$ ならば $wl(s) = l(s)$
- それ以外の時は，
  - 子局面 $c = apply(s, m), m \in moves(s)$ の中に手番の負けの末端局面がある時は $wl(s) = 1$
  - 子局面 $c = apply(s, m), m \in moves(s)$ の中に $wl(c) < 0$ なる c がある場合は，$wl(s) = -max_{wl(c)} wl(c) - 1$
  - それ以外の時は $wl(s) = 0$

「どうぶつしょうぎ」の $l(s)$ の分布は，図 2 の左半分を切り出したものになる．この経験的エントロピー H_0 を求めると 2.5 bit となり，ウェーブレット木のサイズを $wl(s)$ の半分近くまで減らせる可能性があることが分かる．

5. 実　　験

どうぶつしょうぎの手数データベースを提案した手法で表現する実験をおこなった．

完全ハッシュ関数の実現には CHD, BDZ を共に実装してある CMPH (C Minimal Perfect Hashing Library)[4] のバージョン 1.1 を用いた[⋆2]．CMPH では CHD, BDZ 以外に様々なアルゴリズムによる完全ハッシュ関数をサポートしているが，記憶容量に関しては実現されている他のアルゴリズムは大きく劣るため，用いなかった．

ウェーブレット木の実装としては，SDSL[8] のバージョン 0.9.8 の中の wt_huff を用いた[⋆3]．

SDSL では wt_huff の内部で用いるビットベクトルの rank, select 操作に用いる補助データ構造の与え方を指定することができるが，予備実験の結果，最も記憶容量を節約できた

```
wt_huff<rank_support_v5<>,select_support_dummy,select_support_dummy >
```

を用いた．この指定法では select 操作を定数時間でおこなうことはできないが，今回はこのデータ構造に対して select をおこなうことは想定していないので問題ない．

なお，CMPH, SDSL 共に使用中の記憶容量と保存 (CMPI では dump, SDSL では serialize) したファイルのサイズが，ほぼ一致していることはソース上で確かめてあるので，以下では保存したファイルサイズで計測する．

まず，CHD, BDZ で MPH を実現した結果を保存したファイルの大きさはそれぞれ，26.9MB, 34.4MB となった．MPH に対応する SDSL のウェーブレット木を保存したファイルの大きさは 63.4MB となった．それぞれ，合計して　26.9MB+63.4MB=90.3MB, 34.4MB+63.4MB=97.8MB で表現できていることが分かる．

BDZ で $M = 1.23N$ とした PH ではそれぞれ，24.5MB, 70.4MB で合計して 94.8MB で MPH よりは改善しているが，CHD+MPH には及ばなかった．CHD+PH では load factor の値によって，ファイルの大きさが異なる．load factor ごとのそれぞれのファイルのサイズの変化を図 3 に示す．load factor が 88%の時が最小で 88.5MB となっている．

⋆1 インデックス化された後の $wl(s)$ はランダムに並んでいると考えられているので，block-sorting 等の文脈を利用するような圧縮法を用いても H_0Nbit 以下に圧縮することは期待できない

⋆2 完全ハッシュ関数の実装としては岡野原大輔氏の作成した Bep(http://www-tsujii.is.s.u-tokyo.ac.jp/ hillbig/bep-j.htm) が知られているが，これはアルゴリズムは BDZ と同等で，また集合に含まれないキーが与えられた時に，検知するためにキー自体を保持する構造になっているため，今回の目的には使えなかった．

⋆3 ウェーブレット木の実装としては，岡野原大輔氏の作成した wat-array(http://code.google.com/p/wat-array/wiki/WatArrayIntro) が知られているが，これは圧縮をおこなっていないので，今回の目的には使えなかった．

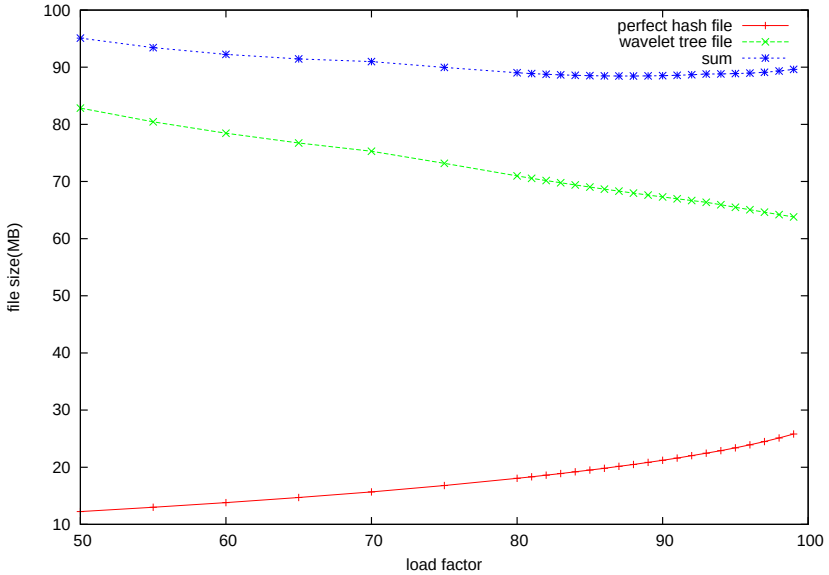

図 3 load factor ごとのファイルサイズ (wl(s))

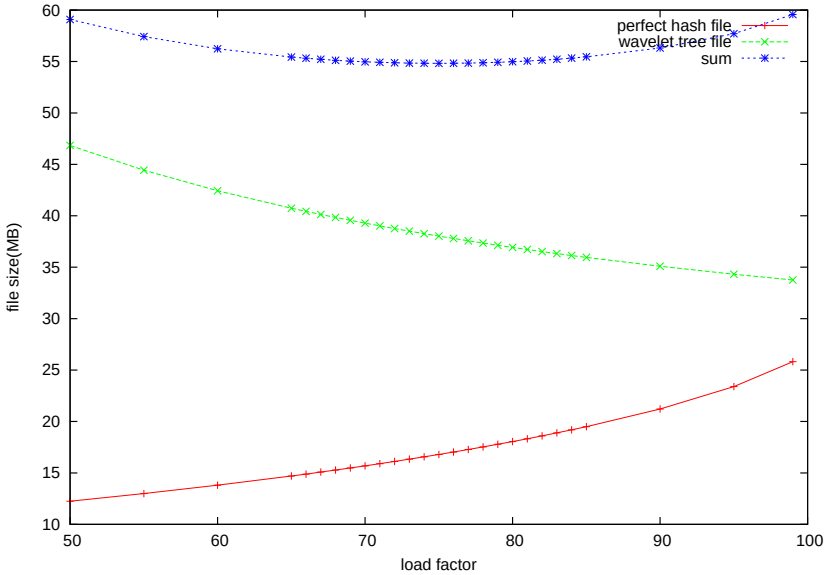

図 4 load factor ごとのファイルサイズ (l(s))

一方，$wl(s)$ の代わりに $l(s)$ を求めるようにする場合のファイルサイズを求めると，MPH に対応する SDSL のウェーブレット木を保存したファイルの大きさは 33.6MB となった．CHD, BDZ でそれぞれ合計は 60.5MB, 68.0MB となる．BDZ で $M = 1.23N$ とした PH ではウェーブレット木のファイル容量は 36.7MB なので合計は 61.2MB となる．

CHD+PH では load factor の値によって，ファイルの大きさが異なる．load factor ごとのそれぞれのファイルのサイズの変化を図 4 に示す．load factor が 75%の時が最小で 54.8MB となっている．

6. おわりに

大規模データに対する完全ハッシュ関数と簡潔データ構造を利用した wavelet tree 等のデータ構造に関する研究の進歩は目覚ましく，オープンソースのライブラリを用いるだけで，かなりの記憶容量の削減を実現できる．本稿ではそれに加えて，局面のノードを一段展開することを許して，負け局面の手数だけを記憶するようにして，更にデータベースのサイズを減らす方法を提案した．

これにより,「どうぶつしょうぎ」の完全プレイのデータベースを 54.8MB の記憶容量で実現できた．ライブラリのライセンスなどの問題はあるものの，実際にゲーム機やスマートフォンでも実現可能なサイズと言えるだろう．

なお，簡潔データ構造は「どうぶつしょうぎ」よりも状態空間が大きく後退解析中のメモリ容量が問題にあるゲームの完全解析でも利用可能で，今後ますます重要になってくると思われる．

参 考 文 献

1) M. L. Fredman and J. Komlos: On the size of separating systems and families of perfect hashing functions, SIAM Journal on Algebraic and Discrete Methods, Vol. 5, pp. 61 – 68(1984).
2) F. C. Botelho, R. Pagh, N. Ziviani: Simple and space-efficient minimal perfect hash functions, In Proceedings of the 10th International Workshop on Algorithms

and Data Structures (WADs'07), Springer LNCS, vol. 4619, pp. 139 – 150 (2007).
3) D. Belazzougui, F. C. Botelho and M. Dietzfelbinger: Hash, Displace and Compress, In Proceedings of the 17th European Symposium on Algorithms (ESA'09). Springer LNCS, vol. 5757, pp. 682 – 693(2009).
4) D. C. Reis and F. C. Botelho: cmph , http://sourceforge.net/projects/cmph/, Online; accessed 06-Mar-2012.
5) D. A. Huffman: A method for the construction of minimum redundancy codes, Proc. IRE, Vol. 40, pp. 1098 – 1101(1951).
6) Witten, Ian H. and Neal, Radford M. and Cleary, John G.: Arithmetic coding for data compression, Communications of the ACM(CACM), Vol. 30, No. 6, pp. 520 – 540(1987).
7) 定兼邦彦: 大規模データ処理のための簡潔データ構造, 情報処理学会誌, Vol. 48, No. 8, pp.899 – 902(2007).
8) Simon Gog: SDSL - Succinct Data Structure Library, http://www.uni-ulm.de/in/theo/research/sdsl.html, Online; accessed 06-Mar-2012.
9) John W. Romein and Henri E. Bal: Awari Is Solved, Journal of the ICGA, Vol. 25, pp. 162–165(2002).