Hexゲームを解く

鈴木 豪† 乾 伸雄† 小谷善行†

Hex ゲームは、ルールは単純であるが可能手の数が多く、コンピュータにプレーさせようとすると困難を伴う。本稿では盤のサイズが 4 × 4 と 5 × 5 の場合について、様々な探索アルゴリズムを使ってゲームを解き、それらアルゴリズムを組み合わせることによりより高速に解を得られるという結果を示す。また、処理は軽いが精度が低い評価関数と処理は重いが精度が高い評価関数を探索する深さにより使い分けることによりより高速に解を得た。最終的な実験の結果、Pentium4-1.4GHz のマシンを使い 4 × 4 のサイズを 0.17 秒、5 × 5 のサイズを 414 秒で解いた。

Solving Hex

Tsuyoshi SUZUKI,† Nobuo INUI† and Yoshiyuki KOTANI†

The game of Hex is played on a diamond-shaped board by two players, Black and White. Solving Hex is difficult because of its high branching factor. In this paper, we solve the game of Hex played on 4 × 4 and 5 × 5 board using various search algorithms. And we did the speed-up of the search by combining the algorithms. As the result, we solved 4 × 4 and 5 × 5 in 0.17 and 414 seconds respectively.

1. はじめに

Hex は、ルールは単純であるが可能手の数が多く、コンピュータにプレーさせようとすると困難を伴うゲームである。本稿では、盤のサイズが 4 × 4 と 5 × 5 の場合を様々なアルゴリズムを使って解き、それらアルゴリズムを組み合わせることによりより高速に解を得られるという結果を示す。探索アルゴリズムは、始めに単純な $\alpha\beta$、ルートのみの反復深化 $\alpha\beta$、多重反復深化 $\alpha\beta$、PVS、ルートのみの反復深化 PVS、多重反復深化 PVS を使った。そのあとで、これらのアルゴリズムを探索する部分木の深さにより変えたアルゴリズムを採用した。また、評価関数ははじめに駒の周りの升の数と、升の価値という単純で軽いものを利用したが、後により正確な評価関数を使い、それらも探索する深さにより変化させた。最終的な実験の結果、Pentium4-1.4GHz のマシンを使い 4 × 4 のサイズを 0.17 秒、5 × 5 のサイズを 414 秒で解いた。

2. Hexゲーム

Hex は自分の駒の色と同じ対辺を自分の駒で接続することを目的とするゲームである。駒どうしが互いに隣接しているとき、それらは接続しているという。ゲーム盤の形は菱形をしており、各升の形は 6 角形をしている。大きさは、10 × 10、11 × 11、14 × 14 が一般的である。4 × 4 のゲーム盤を**図 1** に示す。文献[10] によると Hex のルールは次のようになる。

（1） 先手のプレーヤーは好きな升に黒駒を 1 個置く。
（2） 後手のプレーヤーは黒駒の位置を見て、黒方を取るか白方を取るか決める。白駒を取ったときは、後手は白駒を好きな升に打ってゲームを開始する。黒駒を取ったときには、手番は先手に移り先手が白駒を好きな升に打ってゲームを開始する。（スワップルール）
（3） 以下、交互に自分の駒を盤上に打つ。隣接する升に同色の駒があるときには、この二つの駒は連結しているとみなす。
（4） こうして升においた自分の駒同士を連結させて、自分の色の対辺を自分の駒で連結させれば勝ちになる。

Hex はサイズが 11 × 11 の盤では平均合法手が 100 になる。Hex ははじめ Danish poet と Piet Hein により研究され (1942)、Polygon とう名前で有名になった。Paker Brothers は Hex とう名で商用のものを発売した (1952)[1]。Hex は引き分けにはならないことが知

† 東京農工大学大学院工学研究科
Tokyo University of Agriculture and Technology

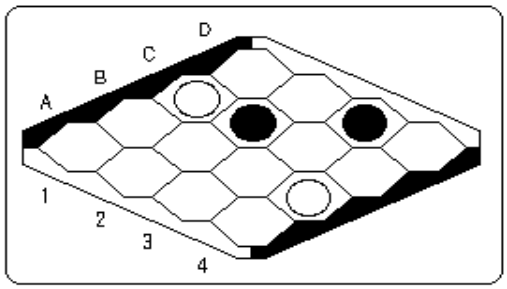

図 1 Hex 盤

られている。これは全ての升が埋められていたなら黒または白の勝ち列が存在しなければならないことによる。David Gale はこの結果が Brouwer の不動点定理と同値であることを示した (1979)[7]。後に John Nash はスワップルールを使わない場合に strategy stealing の議論を使って先手勝ちを証明した (1982)[8]。現在 7 × 7 よりも大きなサイズの盤の必勝法は知られていない。S. Reish は Hex の必勝法が PSPACE 完全であることを証明した (1981)[9]。以下、本稿では、スワップルールは考慮していない。

3. Hex プレイングプログラム

本章では実験に使用した Hex プレイングプログラムの探索アルゴリズムと、その評価関数について述べる。

3.1 探索アルゴリズム

探索アルゴリズムは単純な $\alpha\beta$、ルートでのみ反復する $\alpha\beta$、多重反復深化の $\alpha\beta$ および PVS を使い、その速度を比較した。

(1) $\alpha\beta$ 探索

$\alpha\beta$ 探索[4] は、minimax 探索と同じ値を得るにもかかわらず、不要な枝の探索を行わないアルゴリズムである。この探索効率は探索する子ノードの順番に依存する。子ノードの並びが理想的な場合その探索ノード数は、深さが d、分岐数が b のゲーム木に対しては $b^{d/2}$ 程度であることが知られている[4]。$\alpha\beta$ 探索を**図 2** に示す。ここで $eval(pos)$ は局面 pos を評価する評価関数、$makemove(pos)$ は局面 pos における可能手の生成関数である。以下に示すアルゴリズムでは、反復深化を行わないものは、$makemove$ の中で各指し手を評価関数を呼んだだけの仮評価でソートしているものと仮定する。

(2) 反復深化

反復深化[6] は探索をはじめから深く行うのではなく、浅いところから順次深くしていく。浅い深さで探索得られた最善手を使って次の深さの探索を開始できるので、指し手の順序付けの意味から $\alpha\beta$ 探索の効率が向上に貢献する。反復深化はルートノードのみで行う場合 (これを RID と呼ぶ:**図 3**) や、各ノードで再帰的に行う (これを MID と呼ぶ:**図 4**) などがある。また、実験では深くしていく深さは delta=1 と delta=2 の 2 種類の実験を行っている。

```
fab(pos,alpha,beta,depth)
{
  if( depth<=0 )return( eval(pos) );
  count=makemove(pos);
  for( cnt=0 ; cnt<count ; cnt++ ){
    rtn = -fab(pos[cnt],-beta,-alpha,depth-1);
    if( rtn  > alpha ) alpha=rtn ;
    if( alpha>=beta  ) return( alpha );
  }
  return( alpha );
}
```

図 2 $\alpha\beta$ 探索

```
rid(pos,alpha,beta,depth)
{
  for( search=0 ; search<depth ; search+=delta ){
    for( cnt=0 ; cnt<count ; cnt++ ){
      -fab(pos[cnt],-beta,-alpha,search);
    }
    if( search<(depth-1) )sortmove();
  }
  return( alpha );
}
```

図 3 ルートのみでの反復深化

```
mid(pos,alpha,beta,depth)
{
  for( search=0 ; search<depth ; search+=delta ){
    for( cnt=0 ; cnt<count ; cnt++ ){
      -mid(pos[cnt],-beta,-alpha,search);
    }
    if( search<(depth-1) )sortmove();
  }
  return( alpha );
}
```

図 4 多重反復深化

(3) PVS 探索

PVS 探索は Campbell と Marsland により提案された[5]。PVS 探索は pvs と fab という二つの関数からなる。これは最初の子ノードの探索

```
pvs(pos,alpha,beta,depth)
{
  if( depth<=0 )return( eval(pos) );
  count=makemove(pos);
  rtn = -pvs(pos[0],-beta,-alpha,depth-1);
  if( rtn  > alpha ) alpha=rtn ;
  if( alpha>=beta  ) return( alpha );
  for( cnt=1 ; cnt<count ; cnt++ ){
    rtn = -fab(pos[cnt],-alpha-1,-alpha,depth-1);
    if( rtn  > alpha ){
      if( (alpha+1)!=beta && depth<2 )
        rtn = -fab(pos[cnt],-beta,-rtn,depth-1);
      alpha=rtn;
    }
    if( alpha>=beta ) return( alpha );
  }
  return( alpha );
}
```

図 5　PVS 探索

```
rpvs(pos,alpha,beta,depth)
{
  for( search=0 ; search<depth ; search+=delta ){
    -pvs(pos[0],-beta,-alpha,search);
    for( cnt=1 ; cnt<count ; cnt++ ){
      -fab(pos[cnt],-alpha-1,-alpha,search);
      if( 再探索 )-fab(pos[cnt],-beta,-rtn,search);
    }
  }
  return( alpha );
}
```

図 6　ルートのみでの反復深化 PVS 探索

を行った後、その子ノードの評価値を使ってそれ以降の子ノードを幅が 1 のウインドウで $\alpha\beta$ 探索を行う。探索が失敗した場合は、そのときの情報を使って再探索を行う。PVS 探索はチェスや将棋において実践的に $\alpha\beta$ より高速であることが知られている。PVS 探索を**図 5** に示す。

PVS にも反復深化を組み合わせることができ、ルートでのみ反復深化する rpvs と多重反復深化を行う mpvs を**図 6**、**図 7** のように定義する。

PVS と反復深化との組合せは最初の子の探索やそれ以降の子の探索、最探索にどのアルゴリズムを使うかなど様々なものが考えられる。

3.2 評価関数

評価関数の評価要素は、駒の近傍の升の数と、升の価値の 2 種類を使った。評価関数はこれらの組合せで構成される。

(1)　駒の近傍の升の数

駒の近傍の升の数の少ないものも評価関数として利用されている[1)]。そこで駒を置いたときの周りの升の価値を-1 と定義する (**図 8**)。

```
mpvs(pos,alpha,beta,depth)
{
  for( search=0 ; search<depth ; search+=delta ){
    -mpvs(pos[0],-beta,-alpha,search);
    for( cnt=1 ; cnt<count ; cnt++ ){
      -mid(pos[cnt],-alpha-1,-alpha,search);
      if( 再探索 )-mid(pos[cnt],-beta,-rtn,search);
    }
  }
  return( alpha );
}
```

図 7　多重反復深化 PVS 探索

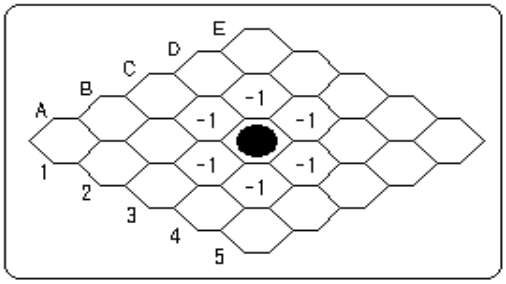

図 8　駒の近傍の升の数による評価

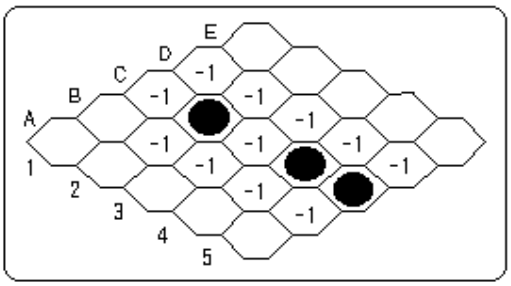

図 9　近傍が重なった場合の評価

複数の駒の近傍は重複は数えずに単に近傍というだけで升の価値を-1 にする。**図 9** の例では 3C,4B,4D,5B,5D はそれぞれ二つの駒の近傍になっているがその価値は-1 である。駒が置かれている升および、近傍でない升の価値は 0 と定義する。

駒が置いてある近傍の升の価値による局面の評価は、盤上の升のこれらの評価による評価値の総和となる。

(2)　升の価値

近傍の升だけでは隅に置く価値が高くなる。例えば、図 9 の 1A の価値は高いとは考え難い。そこで、これを補正するために升の価値を導

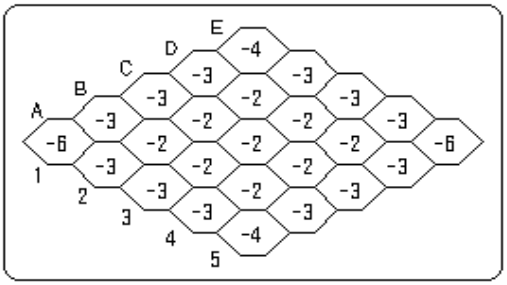

図 10　升の価値の 5 × 5 の場合の升の価値

入した。左右の頂点にある升の価値は-6，上下の頂点にある升の価値は-4，頂点以外の辺にある価値は-3，それ以外の升の価値は-2 である。**図 10** に盤のサイズが 5 × 5 の場合の例を示す。局面の評価は、駒が置いてある升の価値の総和となる。

3.3　トランスポジションテーブル

Hex における探索では同一局面が高い頻度で出現する。そこでトランスポジションテーブルを利用した。局面はハッシュ値として保持され、チェスや将棋で使われている乱数による XOR 方式を採用した。4 × 4 のサイズの場合では、トランスポジションテーブルのエントリー数が 16K でも異なる局面を同一局面と判定することはなかった。また、エントリー数は始め 16K であったが、後に 1M に増やした。しかし、4 × 4 のサイズではエントリー数による大きな差ない。5 × 5 のサイズではエントリー数により時間は大きく変わるが、メモリーの制限から今回は 8M のエントリー数 (実メモリーサイズ 128MB) までしか実験を行っていない。

4. 実 験 結 果

3 章で説明したアルゴリズムと、評価関数で実験を使ってサイズが 4 × 4 と 5 × 5 の盤の解を求めた。以下、反復深化において RID1 と MID1 と表しているものは、探索の深さを増やす数 delta=1 としたものであり、RID2 と MID2 と表しているものは delta=2 としたものである。使用したマシンは Pentium 4-1.4GHz で、1 秒間に約 100 万ノードを探索する。

4.1　4 × 4 の結果

トランスポジションテーブルを使い、評価関数がなし (勝ち負けの判定だけ)、駒の近傍の升、升の価値、駒の近傍の升と升の価値を使ったもので実験を行ったときの探索ノード数を**図 11** に示す。ここで、トランスポジションテーブルのエントリー数は 1M である。

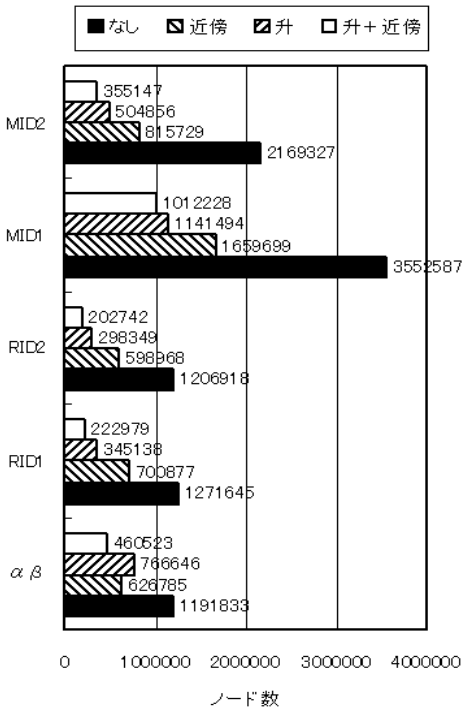

図 11　探索ノード数による比較

アルゴリズムを比較すると RID2 すなわちルートのみの反復深化で深さを $1, 3, 5, \cdots$ と増やすものものが、他のアルゴリズムと比較して最も探索ノード数が少ない結果となった。また、評価関数では駒の近傍と升の価値を使ったものが最も探索ノード数が少ない結果となった。4 × 4 においては多重反復深化は、探索ノード数が多くなる傾向にある。

探索をより高速化するために PVS を実装して実験を行った。その結果を**図 12** に示す。ここで反復深化を行うものは delta=2 のものを使っている。また、評価関数は駒の周りの升と升の価値を使い、トランスポジションテーブルのエントリー数は 1M である。PVS を採用したものが、それを採用しない $\alpha\beta$、ルートのみの反復深化、多重反復深化のいずれにおいても探索ノード数が減少した。特に、RPVS は PVS を採用しなかったもので最もノード数の少なかった RID よりも探索ノード数が 12 %減少し、4 × 4 のサイズにおいて最も高速であった。同様の実験を PVS と並んでよく使われている negascout アルゴリズムを実装して行ったが、結果は PVS の場合とほとんど同じであった。

4.2　5 × 5 の盤

評価関数に”近傍の価値＋升の価値”を使い、トランスポジションテーブルのエントリー数を 8M として各

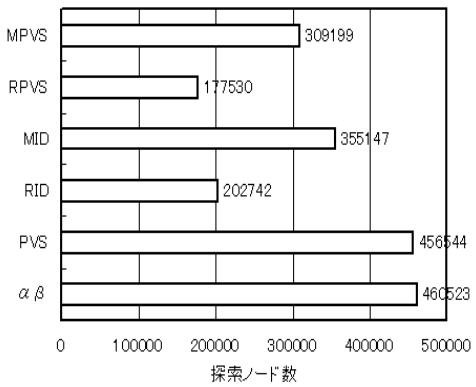

図 12 PVS と探索ノード数の比較

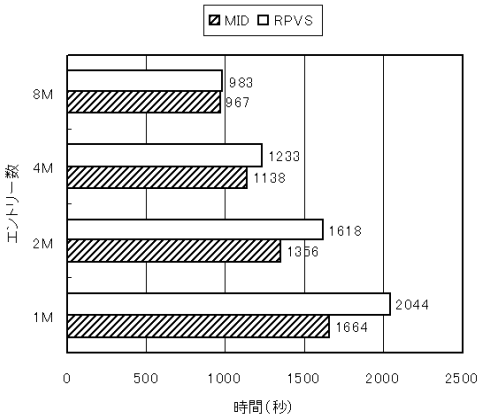

図 14 探索時間

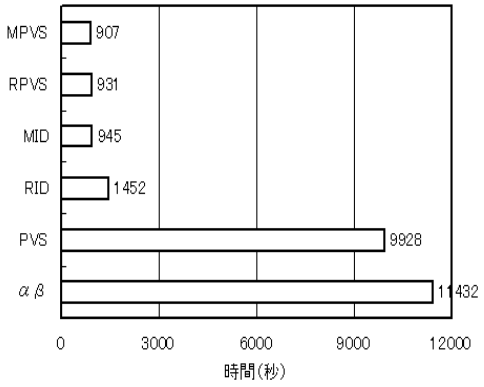

図 13 探索時間

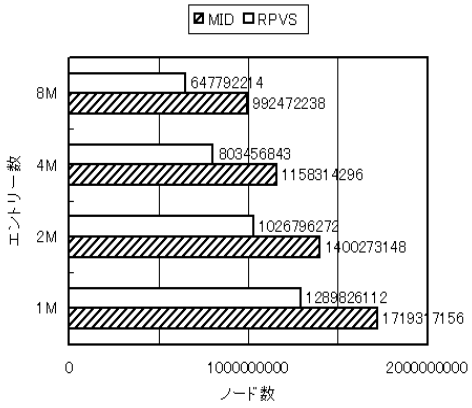

図 15 探索ノード数

アルゴリズムを使って盤のサイズが 5 × 5 の場合を解かせた。ここで反復深化を行うものは delta=2 の $1, 3, 5, \cdots$ 型のものを使っている。

図 13 に解が求まるまでの探索時間を示す。$\alpha\beta$ および PVS では探索時間が 9000 秒を超えたが、それ以外のアルゴリズムでは 1800 秒以下であった。最も速かった MID では 967 秒であり、RPVS も MID に近い 983 秒で解いた。

上記の実験で結果の良かった MID と RPVS を使って、トランスポジションテーブルのエントリー数を変化さて実験を行った。

図 14 に探索時間の結果を示す。どのエントリー数でも、MID の方が RPVS よりも探索時間は少なくなっているが、1M から 8M に増えるにしたがって、その差は小さくなり、8M ではほとんど同じ探索時間となっている。

図 15 に探索ノード数を示す。どのエントリー数でも RPVS の方が MID よりも探索ノード数が少ない結果となった。その減少率は、エントリー数が増えるにつれ大きくなる傾向にあり、1M では 25 %であったが、8M では 35 %となった。

5. 評価関数とアルゴリズムの組合せによる改善

ここでは探索する部分木の大きさにより探索アルゴリズムを変える方法と、探索する深さにより評価関数を変える手法について述べる。以下、実験は全て盤のサイズが 5 × 5 で行なっている。

5.1 評価関数の改善

先の実験では精度は低いが処理の軽い評価関数を使った。これは両端が何手で接続できるといった、精度は高いが処理の重い評価関数を使ったものより、探

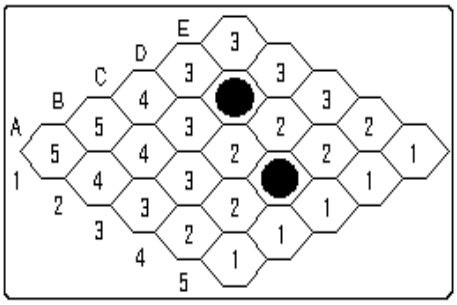

図 16　右辺につながるまでの手数

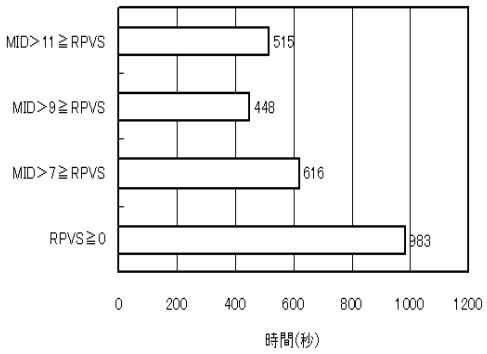

図 17　探索深さによるアルゴリズムの組合せ

索ノード数が増えても結果的には高速に解を得られたためである。ここでは正確だが重い評価関数を分解することにより、精度は落ちるか処理の軽い評価要素を導入した。この評価要素は次のように計算する。

(1)　先手が右辺と接続できるまでのまでにかかる手数を各升について計算する

(2)　左辺に接した升の値を小さいものから二つとり、その和にマイナスをつけたものを評価値とする

例えば、**図 16** には先手から見た場合の各升が右辺に何手で接続できるかを計算した値が入っている。この局面の評価値は、先手左辺に接する升のなかの小さなほうから上位二つの値の和を取ったものにマイナスをつけて-6 となる。

小さな値二つの和にしているのは、一つだと後手に接続を阻止される可能性があるためである。実際、実験を行なった結果も、一つの値を使うよりも二つの値を使うほうが良かった。また、後手についても両端が接続できる手数による評価値を計算でき、これらを考慮した評価関数も定義できるが、実験の結果、探索ノードは減少するものの、全体の探索時間が増加したので、これらは組み込んでいない。

以下、先の駒の近傍の升の価値と、升の価値の和を使った評価関数を E1、それに加えてここで導入した評価要素を使った評価関数を E2 と呼ぶことにする。

5.2　探索アルゴリズムの組合せによる高速化

先の実験では深さ 4 では RPVS が最も探索が速く、深さ 5 では MID が最も探索が速い結果となった。このことから、大きな部分木を探索するときには MID、小さな部分木を探索するときには RPVS を使うことが考えられる。この場合、どの深さで探索アルゴリズムを切り替えるかその深さを決定する必要がある。そこで、アルゴリズムを変える深さを変化させて実験を行なった。その結果を**図 17** に示す。ここで評価関数は駒の周りの升の価値と升の価値を使った E1 を使っている。また、トランスポジションテーブルのエントリー数は 8M である。

図 17 から、RPVS と MID を切り替える深さを 0 〜 9 と大きくしていくと探索時間が次第に減少し、9 を超えると再び探索時間が増加することがわかる。RPVS と MID を切り替える深さを 9 にしたときには、RPVS のみを使った探索よりその探索時間が 54 %減少し、MID のみを使ったものよりも探索時間が、53 %減少した。

5.3　評価関数の組合せによる高速化

探索と同様に深さごとに評価関数を変えることも考えられる。いくつかの予備実験の結果、探索がルートに近い部分では正確だが処理の重い関数 E2 を使い、末端に近い部分では軽い評価関数 E1 を使う場合が良好な結果を示したため、これについてさらに実験を行なった。結果を**図 18** に示す。ここでは探索アルゴリズムを上記の深さ 9 を基準としてそれより浅いところでは RPVS を使い、深さが 9 よりも大きいところでは MID を使っている。また、トランスポジションテーブルのエントリー数は 8M である。

図 18 から深さが 5 よりも浅いところでは評価関数 E2 を使い、それよりも深いところでは E1 を使ったものが最も高速な結果となった。これは MID のみを使ったものよりも最終的に 57 %探索時間が減少した。

6. 考　　察

4 × 4 の場合、探索時間が最も速かったものはルートのみで反復深化を $1, 3, 5, \cdots$ と行う RPVS であった。このとき、$1, 3, 5, \cdots$ 型の反復深化を行う MID は時間が RPVS の約 2 倍となっている。一方、5 × 5 のサイズにおいては、MID が最も探索時間が短かった。これは、4 × 4 の場合は問題が小さく、多重反復深化のメリットが発揮しにくいためと考えられる。小さい

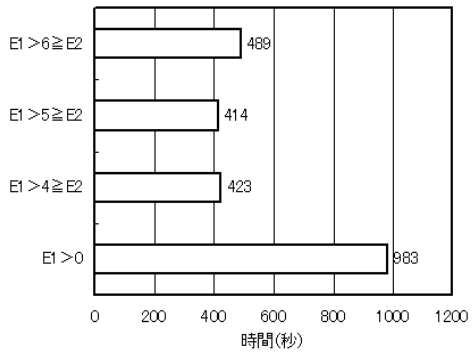

図 18 深さごとに評価関数を変えた場合

問題に対しては単純なアルゴリズムでも良いが、ある程度大きくなると多重反復深化のような多少のコストが増えてもより有望そうな部分木を探索するアルゴリズムのほうが勝ると考えられる。

5 × 5 の場合、トランスポジションテーブルのエントリー数が増えるにしたがって、RPVS の探索時間は最も高速であった MID の探索時間に漸近した。また、エントリー数の増加に伴って、RPVS の MID と比較した探索ノード数の減少率も大きくなる傾向を示した。エントリー数をより増やせば RPVS のほうが MID よりも高速になる可能性はある☆。

各升に駒を置くと何手で両端を接続できるかというより正確な関数や、その簡易計算をした評価関数も用意し実験も行なった。これらを使うと 4 × 4 では探索ノード数が 5 分の一に減少したが、処理が重く、軽い簡単な評価関数を使ったものの方がゲームとを解く上では高速であった。

探索している部分木の大きさにより、探索アルゴリズムを変化させるアルゴリズムは、5 × 5 のサイズにおいて解を求める速度が一つのアルゴリズムを使うよりも約 2 倍に高速化した。また、探索する深さにより評価関数を変えることによっても高速化が行なえた。単純なアルゴリズムと単純な評価関数を使った場合でも、探索する木の性質によりそれらをうまく切り替えて使うことは有効であるといえる。

7. ま と め

盤のサイズが 4 × 4 と 5 × 5 の場合について様々な探索アルゴリズムを使ってゲームを解き、さらにそれらアルゴリズムを探索する部分木の大きさにより使い分けて高速化を行なった。また、評価関数においても精度は高いが処理の重いものと、精度は低いが処理が軽いものを探索する深さにより使い分けてより高速化を行なった。最終的な実験の結果 4 × 4 においては $1, 3, 5, \cdots$ と深さを増やしていくルートのみの反復深化と PVS の組合せが最も探索ノード数が少なく、0.17 秒で先手必勝を解いた。また、5 × 5 では探索する部分木が深さ 9 以下ではルートノードで反復深化する PVS を使い、それより大きな部分木では多重反復深化をする手法を使って 414 秒で解いた。

本稿で用いた探索アルゴリズムを問題により切り替える手法は他の探索問題でも利用できると考えられる。ここではアルゴリズムを切り替える深さを実験により決定しているが、探索中に自動的に決定すること、ノードのタイプなどにより切り替えることなどが今後の課題である。

☆ より大きなメモリーを搭載したマシンでのエントリー数を 16M,32M とした場合には実際にそうなった.

参 考 文 献

1) V. V. Anshelevich. The Game of Hex: An Automatic Theorem Proving Approach to Game Programming. Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 2000, 189-194, AAAI Press, Menlo Park, CA.
2) V. V. Anshelevich. The Game of Hex: The Hierarchical Approach. Combinatorial Game Theory Workshop, MSRI, Berkeley, July 2000.
3) V. V. Anshelevich. Hexy Wins Hex Tournament. The ICGA Journal, 23, 3, 2000, 181-184.
4) D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning. Artificial Intelligence 6(4), 1975, 293-326.
5) M.S. Campbell and T.A. Marsland, Parallel Search of Strongly Ordered Game Tree, Computing Surveys 14(4), 1982, 533-551.
6) T.A. Marsland, A Review of Game-Tree Pruning. ICCA Journal 9(1), 1986, 3-19.
7) Gale, D. The Game of Hex and the Brouwer Fixed-Point Theorem. American Mathematical Monthly 86, 1979, 818-827.
8) Berlekamp, E.R. Conway, J.H. and R.K. Guy, R.K. Winning Ways for Your Mathematical Plays. New York, Academic Press. 1982.
9) Reisch, S. Hex ist PSPACE-vollstadig. Acta Infomatica 15, 1981, 167-191.
10) 松原, 竹内編 ゲームプログラミング, 共立出版, 1997.