Program Promenade

連載

正方形の破壊

田中 哲朗（東京大学情報基盤センター）
ktanaka@tanaka.ecc.u-tokyo.ac.jp

ACM プログラミングコンテストでは使用できるプログラミング言語が制限されている．開催年度や予選の開催場所によって変動はあるが，ここ数年は C，C++，Java，Pascal などが指定されることが多いようだ．参加者からは，「Perl だったらもっと早く書けるのに」とか「ML が使いたい」といった声もあがっているが，使用する言語によって問題の難易度が変化することを避けるためには，この制限も仕方がないだろう．

しかし，C，C++，Java，Pascal の 4 言語だけでも難易度を揃えるのは難しそうである．オブジェクト思考言語か手続き型言語かといった言語のコアの部分の区別は，動くプログラムを限られた時間で作成するというプログラミングコンテストでは差が出にくいと思われるが，標準ライブラリの規模の差は大きい．C++ や Java の標準ライブラリは，無駄に肥大化したかのように見えて，案外アルゴリズムの記述に役立つ要素を含んでいる．個人的には，ある程度以上のレベルに達した参加者ならば，C よりも C++ や Java の方がプログラミングコンテストでは有利ではないかと思っている．

そこで，今回は過去 3 回で使われた C ではなく C++ と STL（Standard Template Library）を利用して，なるべく手を抜いてプログラムを記述することを試みることにする．使用する STL の要素に関しては必要に応じて簡単な説明を加えるので，STL の知識がなくても概略は理解できるだろう．

■問題：正方形の破壊

今回取り上げる問題は，2001 年度アジア地区予選 Taejon（韓国）大会の問題 H「Square Destroyers」である．

図 -1 の左側のように長さの等しいマッチ棒が格子状に置かれている．図 -1 には辺の長さ 1 の正方形が 4 個，辺の長さ 2 の正方形が 1 個，辺の長さ 3 の正方形が 1 個あるが，マッチ棒を取り除くと正方形が破壊されていく．最少いくつのマッチ棒を取り除けば正方形を全部破壊できるかというのがこの問題である．

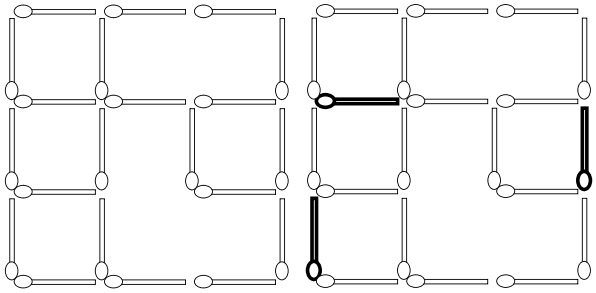

図 -1　入力の例

図 -1 では右側の太線で書かれた 3 本のマッチ棒を取り除くと正方形が全部破壊されるが，2 本で全部破壊することはできないので答えは 3 となる．3 本で正方形を全部破壊する組合せは他にもあるが，問題ではどのマッチ棒を取り除くかを尋ねていないので，3 とだけ答えればよい．

コンテストの問題では，格子の大きさが最大 5 × 5 の範囲に収まると制限している．したがって，使われるマッチ棒の最大数は 2 × 5 × 6 = 60 本，出現する正方形の最大数は $5^2 + 4^2 + 3^2 + 2^2 + 1^2 = 55$ 個となる．

この問題は計算量的には NP 困難のクラスに属する問題であり[☆1]，動的計画法などのプログラミングコンテストの「定石」的なアルゴリズムは適用できない．まずは動くプログラムを作って，少しずつ改良を試みるという方針で進める．

☆1　このことはそれほど自明ではないので，多項式時間の解法を探し続けて時間切れになってしまった参加チームも多いだろう．

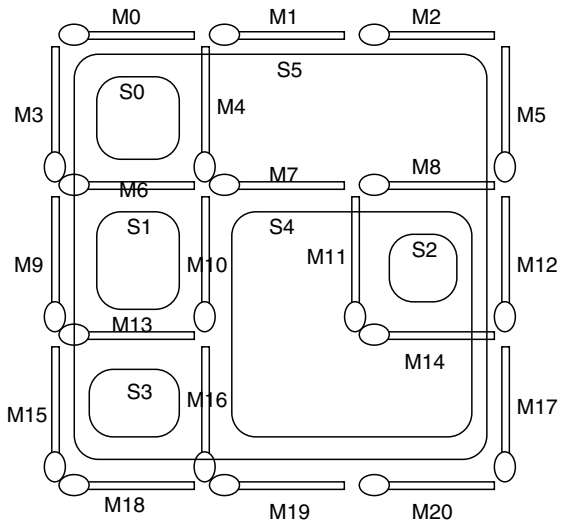

図 -2　マッチ棒と正方形の対応

■プログラムの準備

以降では図 -2 のように，マッチ棒を Mi，正方形を Sj と表記する．この例でマッチ棒 Mi を取り除いた時にどの正方形が取り除かれるかの対応は以下のように書ける．

```
M0, M3 -> {S0, S5}
M1,M2,M5 -> {S5}
M4 -> {S0}
M6 -> {S0, S1}
M7 -> {S4}
M8 -> {S2, S4}
M9 -> {S1, S5}
10 > {1, 4}
M11,M14 -> {S2}
M12 -> {S2, S4, S5}
M13 -> {S1, S3}
M15,M18 -> {S3, S5}
M16 -> {S3, S4}
M17,M19,M20 -> {S4, S5}
```

マッチ棒の位置情報を残しておかなくても，マッチ棒と正方形の対応だけの情報で解を求めることができることは明らかだろう．元のデータからこの表現への変換は容易なので，入力と変換部分のプログラムは省略して，変換後のデータを扱うプログラムを作成することにする☆2．

Mi を取り除いたときに破壊される Sj は，重複がなく順序も定義されないので集合として表現するのが自然である．一般的に STL で集合を表現するのに用いられる標準コンテナは set だが，この問題では正方形の番号である整数 0 ～ 54 の集合を入れればよいので，要素の有無を 1 ビットで表現可能な bitset を使うことにする．

bitset を使うと set よりもメモリ使用量が少なく，基本的な操作を高速に行えるだけでなく，集合の和 (|)，積 (&)，否定 (~) 等を演算子を使って自然に書けるというメリットがある．

Sj の集合は iset という名前で以下のように定義する．55 は表現に必要なビット長を表す．

```
typedef bitset<55> iset;
```

さらに iset を他の標準コンテナの中で用いるため☆3，以下のプログラムのように，全順序関係の演算子「<」を

- 要素数の多い方を大と見なす．
- 要素数が同じなら，若い番号の要素を含んでいる方を大と見なす．

と定義することにする．!(a<b) かつ !(b<a) だと a==b とみなされてしまうので，正しい全順序関係になっていない定義を書くと，後で思わぬバグに悩まされることになる．

```
bool operator<(const iset &a,
               const iset &b) {
  // 要素数の比較
  int cdiff=b.count()-a.count();
  if(cdiff>0) return true;
  else if(cdiff<0) return false;
  // 要素単位での比較
  for(int i=0;i<a.size();i++){
    if (a.test(i) != b.test(i))
      return b.test(i);
  }
  // 等しい時
  return false;
}
```

☆2　実際のコンテストでは，今回省略した部分のプログラムを仕様通りに作成することも重要である．

☆3　set の実装における要素を挿入する場所の決定や，sort で用いられる．

```
bool operator>(const iset &a,
               const iset &b) {
  return b<a;
}
```

上のプログラムで使用したbitsetのメンバの意味を以下に簡単に説明する．

count()　1となっているビットの数
size()　全体のビット長．この場合は55
test(n)　ビット n が1となっていたらtrueを，0ならfalseを返す

■ setを使う

マッチ棒の番号は解を求めるのには不要である．このため，番号の代わりに，それが破壊できる正方形の集合（isetクラスの要素）を用いてマッチ棒を表現することができる．マッチ棒の集合をset<iset>で表すと，重複する要素を含まない形で自然に表現できる．

すでにマッチ棒をいくつか取り除いて，いくつかの正方形を破壊した状態を考える．このとき，残った正方形の集合と，残ったマッチ棒（に対応する破壊できる正方形の集合）の集合を与えて，残った正方形をすべて破壊するに必要なマッチ棒の最小値を求める関数solve0は以下のようにして定義できる．

1．マッチ棒を1つ選ぶ．
　見込みのありそうなマッチ棒から選ぶのがよい．
2．選んだマッチ棒を取り除くことに決めた場合の最小値を求める．
　そのマッチ棒を取り除いた後の正方形の集合とマッチ棒の集合を求め，solve0に再帰的に渡す．返った値に1を足したのがこの場合の最小値となる．
3．選んだマッチ棒を取り除かないと全部の正方形を破壊できないときは2の値を返す．
4．3以外のとき，選んだマッチ棒を取り除かないことに決めた場合の最小値を求め，2の場合の最小値と比べ，小さい方を返す．
　自然に書くと，この部分もsolve0への再帰になるが，1に戻るループとして書ける．

このアルゴリズムを，書き下すと以下のようなプログラムとなる．

```
// ssに残りのマッチ棒（に対応するiset）の集合
// uSetに残りの正方形の集合が入る
int solve0(const set<iset>& ss,
           const iset& uSet){
  // 計測用
  nodeCount++;
  // BIGINTは55以上の適当な数
  int minval=BIGINT;
  // 集合を後ろ（要素数の多い方）から試す
  for(set<iset>::const_reverse_iterator
      i =ss.rbegin(); i!=ss.rend(); ){
    const iset& selected= *i++;
    // マッチ棒1つで残りの正方形すべてを
    // 破壊できてしまった場合
    if(selected==uSet) return 1;
    // このマッチ棒を取り除いた後の
    // ss を別の領域に作る
    set<iset> newSs;
    // i以降のすべての要素に関して
    for(set<iset>::const_reverse_iterator
        j=i; j!=ss.rend();j++){
      // *jからselectedの要素を取り除いて
      // 集合newSsに加える．setなので，
      // 重複要素は自動的に削除される
      newSs.insert(*j & ~selected);
    }
    // selectedと，それ以降のマッチ棒
    // のみで破壊した場合の最小値をこれまでの
    // 最小値と比較
    minval=min(minval,
               solve0(newSs, uSet & ~selected)
               +1);
    // それ以降のマッチ棒を全部取り除いても
    // 正方形を全部破壊できない場合は終了
    iset newUset;
    for(set<iset>::const_reverse_iterator
        j=i; j!=ss.rend();j++)
      newUset |= *j;
    if(newUset != uSet) break;
  }
  return minval;
}
```

上のプログラムで使用したsetのメンバの意味を以下に簡単に説明する．

const_reverse_iterator　後ろ（要素数の多い方）から変更なしに繰り返しアクセスするための

反復子の型
rbegin()　reverse_iterator の始点を得る
rend()　reverse_iterator の終点を得る
insert(x)　x が集合に含まれていない時は加える

この関数に，初期の正方形の集合，初期のマッチ棒（に対応する破壊できる正方形の集合）の集合を与えると問題の解が得られる．

コンテストでは，用意された入力データ☆4 に対して制限時間の 3 分以内に解を出力する必要があるが，このプログラムに対して，5 × 5 の完全格子を入力として与えてみると，nodeCount は 452184438 となり，手元のマシン（Athlon MP2000+）では 55 分かかった．

実際のコンテストで用いられた入力セットに対する nodeCount は 127336 で，実行時間の制限内に収まるものと思われるが，これは，たまたますべての入力が 4 × 4 に収まっていたことによる．条件を満たす入力に対して制限時間内で解けないことがあるというのでは，不満が残る．

■包含関係を利用した枝刈り

図 -2 を見直してみると，M0 は S0 と S5 を破壊するが，M4 は S0 しか破壊しないので，M4 を取り除く解がある場合，代わりに M0 を取り除いても解になっているということが分かる．別の表現をすると，M4 の破壊する正方形の集合が，M0 の破壊する正方形の集合に包含されるので，M4 を選択するという枝は刈ることができると言える．

包含関係を利用した枝刈りは，初期配置だけでなく，初期配置から M12 を取り除いて S2, S4, S5 を破壊した後の，

```
M0,M3,M4 -> {S0}
M1,M2,M5,M7,M8,M9,M11,M14,M17,M19,M20 -> {}
M6 -> {S0, S1}
M9,M10 -> {S1}
M13 -> {S1, S3}
M15,M16,M18 -> {S3}
```

という状態でも行える．たとえば M0 の破壊する正方形の集合は M6 の破壊する正方形の集合に包含されるので（初期配置では比較不能だった），M0 は選択肢から除外できる．このような要素をすべて除外すると，残るのは，

```
M6 -> {S0, S1}
M13 -> {S1, S3}
```

の 2 つのみになる．

準備のため，包含関係をもとに不要な要素を除外する関数を作成する．前章ではマッチ棒の集合を表現するのに set を用いたが，包含関係をもとに不要な要素を除外することにすると自然に重複要素も除外されるので，ここからは単位操作のコストが安い vector を使うことにする．

```
vector<iset> rmSmall(vector<iset>& from){
  // 演算子 > を使って from をソート
  sort(from.begin(),from.end(),
       greater<iset>());
  // newSv に集合を作っていく
  vector<iset> newSv;
  // すべての from の要素について
  for(vector<iset>::const_iterator
      i=from.begin();i!=from.end();i++){
    // newSv 中に自分を包含する要素があるか
    // チェックする．なお sort されているので，
    // newSv 中に自分に包含される要素はない．
    for(vector<iset>::const_iterator
        j=newSv.begin();j!=newSv.end();j++)
      // *i が *j に包含される場合は挿入しない
      if( (*i & ~*j).count() == 0)
        goto found;
    // newSv の後ろに要素を加える
    newSv.push_back(*i);
  found:;
  }
  return newSv;
}
```

この rmSmall を使うと，solve0 に対応する関数は以下のように書ける．

```
int solve1(const vector<iset>& sv,
           const iset& uSet){
  nodeCount++;
  int minval=BIGINT;
  for(vector<iset>::const_iterator
      i=sv.begin(); i!=sv.end(); ){
    const iset& selected= *i++;
    if(selected==uSet) return 1;
    vector<iset> newSv;
```

☆4 コンテスト中は公開されない．Taejon 大会ではコンテスト終了後に問題とともに公開された．

```
    for(vector<iset>::const_iterator
        j=i; j!=sv.end();j++ )
      newSv.push_back(*j & ~selected);
    minval=min(minval,
               solve1(rmSmall(newSv),
                      uSet & ~selected)+1);
    iset newUset;
    for(vector<iset>::const_iterator
        j=i;j!=sv.end();j++)
      newUset |= *j;
    if(newUset != uSet) break;
  }
  return minval;
}
```

このプログラムは vector の以下のメンバを使っている．

const_iterator　前方に繰り返しアクセス（ただし読み出しのみ）するための反復子の型
begin()　iterator の始点を得る
end()　iterator の終点を得る
push_back(x)　要素 x を最後に加える

また，次のような標準アルゴリズムや関数オブジェクトを使っている．

sort　列をソートする
greater　演算子「>」に対応する関数オブジェクトを作る

包含関係を利用した枝刈りによる計算量削減の効果は大きく，5 × 5 の欠けのない格子に対して適応すると，nodeCount は 7972475 となり，手元のマシンでは 76 秒で終了した．しかし，コンテストで使うマシンではまだ制限時間内に終了するかどうか不安が残る．

■最小値に基づく枝刈り

前章のプログラムの実行途中で，たとえば 6 個のマッチ棒で破壊できることが分かれば，それ以降は 6 個以上で破壊する解はいらない．そこで，関数を呼ぶ側で，それまでの最良解を渡して呼び出し，呼ばれた側は最良解を更新できない場合はそれ以上の探索を打ちきり，直ちに返るようにすれば探索範囲を減らすことができる．

プログラムとしては，以下のように solve1 に少々の変更（変更部分は下線で表現）を施すだけで書ける．

```
int solve2(const vector<iset>& sv,
           const iset& uSet,int minval){
  nodeCount++;
   // uSet が残っている状態では 1 よりも良い解は返せない
  if(minval<=1) return minval;
  for(vector<iset>::const_iterator
      i=sv.begin(); i!=sv.end(); ){
    const iset& selected= *i++;
    if(selected==uSet) return 1;
    vector<iset> newSv;
    for(vector<iset>::const_iterator j=i;
        j!=sv.end();j++ )
      newSv.push_back(*j & ~selected);
    minval=min(minval,solve2(rmSmall(newSv),
                             uSet & ~selected,
                                 minval-1)+1);
    iset newUset;
    for(vector<iset>::const_iterator j=i;
        j!=sv.end();j++)
      newUset |= *j;
    if(newUset != uSet) break;
  }
  return minval;
}
```

この改良で，nodeCount は 4558896 と若干減るが，実行時間はまだ 67 秒かかってしまっている．

■分枝限定法

前章のプログラムは，それまでの最小値の深さまで再帰を繰り返して初めて，最小値を更新できないと判断していた．しかし，途中で残りの正方形を全部破壊するために必要なマッチ棒の本数の下限を求めることができ，下限値でも最小値を更新できないことが分かれば，さらに枝を刈ることができる．

ここでは，以下の 2 種類の下限を使ってみる．

(1) 残っている大きさ 1 の正方形の数の半分 (の切り上げ)
1 つのマッチ棒で破壊できる大きさ 1 の正方形はたかだか 2 つなので[☆5]，少なくとも大きさ 1 の正方形

[☆5] 研究室の I 君が指摘してくれた．

の数の半分（の切り上げ）の数のマッチ棒を取り除く必要がある．

(2)（残りの）正方形の数の多いマッチ棒から順に，正方形の数を「重複を無視して」足し算して，残りの正方形の数を超える本数．

これを取り入れると以下のようなプログラムになる．なお，グローバル変数

```
iset sq1Set;
```

に，初期配置における大きさ1の正方形の集合を入れておくものとする．

```
int solve4(const vector<iset>& sv,
           const iset& uSet,int minval){
  nodeCount++;
  // 大きさ1の正方形の数による枝刈り
  int estimate0=((uSet&sq1Set).count()+1)/2;
  if(minval<=estimate0) return estimate0;
  for(vector<iset>::const_iterator
      i=sv.begin(); i!=sv.end(); ){
    const iset& selected= *i++;
    if(selected==uSet) return 1;
    vector<iset> newSv;
    for(vector<iset>::const_iterator
        j=i; j!=sv.end();j++ )
      newSv.push_back(*j & ~selected);
    minval=min(minval,solve4(rmSmall(newSv),
                             uSet & ~selected,
                             minval-1)+1);
    iset newUset;
    for(vector<iset>::const_iterator
        j=i;j!=sv.end();j++)
      newUset |= *j;
    if(newUset != uSet) break;
    // 残りは要素数順に並んでいるので，
    // minval-1個分，要素数を足してみる．
    int bCount=0,count=0;
    for(vector<iset>::const_iterator
      j=i;j!=sv.end() && ++count<minval;j++)
     bCount+= (*j).count();
    if(bCount<uSet.count()) break;
  }
  return minval;
}
```

ここまでくると，nodeCountは254946で，実行時間は9秒となり，コンテストのマシンでも，クリア可能な速度を達成できた．

この時点でも最初のプログラムと比較すると十分速いが，まだまだ速度向上の余地はある．ここまでのプログラムでは，なるべく多くの正方形を破壊するマッチ棒から順に選択してきたが，破壊するマッチ棒が少ない正方形から順に選択するというアプローチもある．ある正方形を破壊するマッチ棒が1つしかなければ，まずはその正方形（を破壊するマッチ棒）を選択した方が，後の枝刈りが有効に行われるので，このアプローチは有効であろう．また，より大規模な問題になった場合はIDA*（繰り返し反復 A*）[1] を使う方法も有力である．

NP困難の問題を相手にするときは，計算量のオーダでアルゴリズムの優劣を議論するのが難しい．少しの工夫でデータによっては数十倍も実行時間が変わってくることもあるが，どこまで工夫しても終わりということがないので，奥が深い．

STLを使うとプログラムが簡潔に書けることを見てきたが，まだまだ書きにくい点もある．ここにあげたプログラムでも，意味から考えると`for_each`，`find_if`，`accumulate`などを使うのが自然であるのに，使うとかえって分かりにくくなってしまうので使っていないところがある．Lispや関数型言語なら，lambda記法を使って簡潔に書けるのに，STLでは別の関数やクラスを定義する必要があるような場所である．

この点に不満を持っている人は多いらしく，C++の標準ライブラリ化を目指すプロジェクトの1つであるBoost（`http://www.boost.org/`）にはlambda記法に相当する書き方ができるLambda Libraryが含まれているようだ．今後，このような拡張がC++の標準に取り込まれていくと，ますます，Cでプログラムを書く参加者は不利になっていくかもしれない．

参考文献

1）Korf, R.E.: Depth-first Iterative-deepening: An Optimal Admissible Tree Search, Art. Intell., Vol.27, pp.97-109 (1985).

（平成14年6月7日受付）