A CHALLENGE – FIND ALL SOLUTIONS

Larry Atwood
Computer Science
Minot State University
Minot, ND 58707
unwound@min.midco.net

Abstract

A checkerboard puzzle existed during the 30's-50 in many forms and with many different names. Names like: Amus 'N' Ande Checkerboard Puzzle, the Great Nut House Puzzle and the Famous Banzee Island Checkerboard Puzzle. The puzzles usually contain 12 oddly shaped pieces, sometimes 14 pieces with the goal of putting them together to form a checkerboard. The total ways of arranging 12 pieces each having 4 rotations, is $12! \cdot 4^{12}$ and this is equal to 8036313307545600 or 8.0^^{15} possible solutions. A computer trying a million arrangements per second would take 254.8 years and a computer trying a billion arrangements per second would take .25 years to check all possibilities. The 22 correct solutions for the checkerboard made up of 12 pieces will be found in 45 seconds or less running on a 1.33 MHz laptop. A detail solution, method and program are presented in the main section.

Introduction

A checkerboard puzzle existed during the 30's-50 in many forms and with many different names. Names like: Wrobbell's Checkerboard Puzzle, Amus 'N' Ande Checkerboard Puzzle, Bug House Puzzle, Great Nut House Puzzle and Famous Banzee Island Checkerboard Puzzle. The puzzles usually contain 12 oddly shaped pieces, sometimes 14 pieces with the goal of rearranging them to form a checkerboard. The shape of the 12 pieces is shown in Figure 1. The Banzee Island Puzzle gives the following history: "Chief Zebas realizing the natives of this little island off, South Africa feared mental torture more than physical punishment established the law that whenever a crime was committed, the period of confinement would depend upon how long it took the prisoner to put a checkerboard together that had been cut into twelve pieces. Occassionally[*sic*], by sheer luck, a prisoner would have the board put together within a few hours – often days or weeks passed before the pieces where fitted together – and many natives had even gone crazy from the torture without the solution."

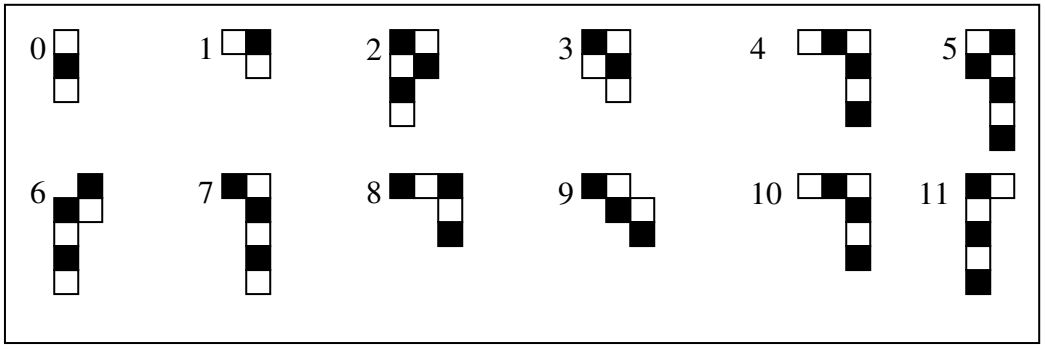

Figure1: The twelve pieces of the puzzle

Puzzle Analysis

The magnitude of the checkerboard puzzle can be realized by an analysis as follows: the checkerboard will be solved by placing a piece in the upper left corner. Then the board will be analyzed by checking the same row for the next empty location and if no empty space is found, the rows directly beneath will be searched. The searching of the rows will always be from left to right. After the second piece is placed, the process will be repeated until all pieces are placed. Because one of the twelve pieces must be placed in the upper left corner, there are 12 choices for this location. After this piece is placed, it follows that there will be 11 choices for the next piece, 10 for the next and finally there will be only one choice for the last piece. Thus 12! is the total for placing the twelve pieces onto the board. All of the pieces except one have 4 different rotations. The rotations for piece 10 are shown in Figure 2. When rotations are added, the total number of possible placements of the 12 pieces with rotations is $12! \cdot 4^{12}$. This evaluates to 8036313307545600 or 8.0^^{15} possible arrangements. A computer trying a million arrangements per second would take 254.8 years and a computer trying a billion arrangements per second would take .25 years.

Piece 0 which has only 2 rotations will cut these numbers in half. Today's desktop computers cannot try a billion arrangements per second, so will cutting the numbers in half help? These numbers are provided to show the magnitude of the puzzle.

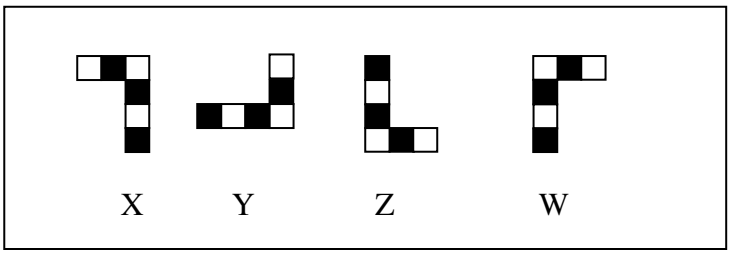

Figure 2: The four rotations of piece #10.

The Challenge

The magnitude of this problem presents a very interesting computer programming assignment in Computer Science. The solution presented uses brute force. See Appendix A for the complete program. The program also uses recursion to process the permutation. Recursion will place a piece, backtracks to remove and rotate the previous piece and tries all the pieces in the permutation until success or failure is found. To be successful with brute force the correct permutation generator [1], data structure, optimization and algorithm have to be chosen. When these correct choices are made, the 22 solutions for the checkerboard are found in slightly less than 45 seconds running on a 1.33 MHz laptop. See Figure 3 for one of the 22 solutions. Because piece 4 and 10 are identical in shape, the number of different solutions can be questioned. Solutions found in Figure 3 and 4 are the same except pieces 4 and 10 have been interchanged. This paper considers Figure 3 and 4 two different solutions. The method presented in this paper was also used to find a single solution during the 4 and 8 MHz. machine age.

4	4	4	4	2	2	2	2
4	7	7	7	7	7	2	2
4	7	3	3	3	6	5	5
9	9	3	3	6	6	5	5
1	9	9	8	6	10	0	5
1	1	9	8	6	10	0	5
11	8	8	8	6	10	0	5
11	11	11	11	11	10	10	10

Figure 3: One of the twenty-two solution.

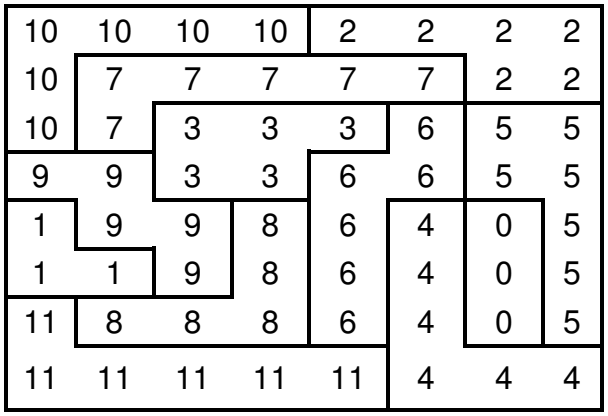

Figure 4: One of the twenty-two solution.

Permutation Generator

The brute force method generates all the permutations of the numbers 0..11. Figure 5 is an example of a generated permutation. The numbers in the permutation represent the pieces in the puzzle and the location in the permutation represents the order in which the pieces will be placed onto the board. At location 0 in Figure 5 is piece 11 and will be the first piece placed onto the checkerboard. At location 1 is piece 5 and will be the second piece placed onto the board. Finally at location 11 is piece 0 and this will be the last piece placed. **Failure of a permutation occurs when a piece cannot be placed and all rotations of the pieces to the left in permutation have been rotated trying to enable the piece to be placed.** Figure 6 is the permutation that would give the solution in Figure 3.

Perm(n)	11	5	9	6	10	7	8	4	3	2	1	0
Index	0	1	2	3	4	5	6	7	8	9	10	11

Figure 5: A permutation of the twelve pieces.

Perm(n)	4	2	7	3	6	5	9	1	8	10	0	11
Index	0	1	2	3	4	5	6	7	8	9	10	11

Figure 6: A permutation of the twelve pieces and solution.

Piece Placement and Rotations

The pieces are placed onto the checkerboard so they will fill a selected square and never fill squares to the left of the selected square or above the selected square. This

requirement can be accomplished by choosing an appropriate square in each rotation. Figure 7 shows the four rotations of piece number 10 and arrows pointing to the key squares. Observe that these rotations will fill a chosen square on the board and will never fill above that chosen location. They will also never fill to the left in the same row as the chosen square. This selection and placement process will fill the board in an orderly downward process. Each piece and its rotations can be stored with offsets. Figure 7 also shows the offsets for the rotation of piece 10. The complete list of all pieces and their corresponding offsets can be found in Table 1. The number preceding each group of 4 rotations in Table 1 is the number of individual squares in each piece. This number aids in reading the file of offsets.

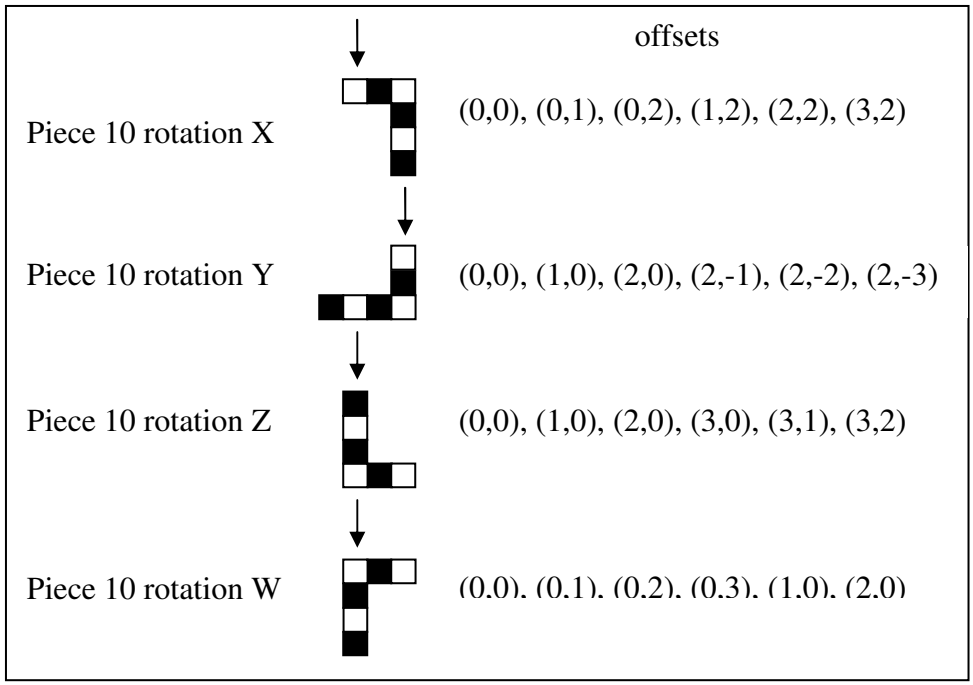

Figure 7: Rotations and offsets

Discovery One

The brute force method can now find 1 of the 22 solutions in approximately 1 hour using a 1.33 MHz. computer. Finding a solution is possible because during the placing of the pieces the permutation fails before all the pieces have to be placed. The number $12! \cdot 4^{12}$ is reduced because only a few of the pieces have to be tried before it fails. The actual time to find the solutions will actually depend on several factors: it depends on the speed of the computer, how the permutation array is initialized, the speed of the permutation generator and other factors. Finding all 22 solutions is still time consuming on a 1.33 MHz. machine.

Discovery Two

As the permutations are generated another observation is discovered that will shorten the process of finding a solution drastically. The permutations are generated by changing the numbers on the left end of the list and working towards the right. Table 2 demonstrates this observation. The numbers at index 11..3 are the same for all 4 permutations. This means that if the algorithm had started trying to place the pieces onto the board by starting with index 11, then 10 and working left and if the process fails at say index 8, no permutation has to be tried until the number in the permutation at 8 changes. This would require the routine that tries the pieces to remember the lowest location of failure and pass this information back to the permutation generator. The generator would then not have to call the solve routine until it generates a permutation with a different number at that failure point. This cuts the time of finding all the solutions to less than 2 minutes on a 1.33 MHz computer. Considerable time is saved because the next empty location does not have to found, the board does not have to be analyzed to see if the piece will fit, the piece doesn't have to be placed and removed many times and other auxiliary routines do not have to be called.

3	5	6	6
1001020	00001101121	000101-12-13-14-1	0000110203040
1000102	1000111101-1	1000102031314	0000102030414
1001020	10010112120	1001020303-14-1	000102030404-1
1000102	10001021011	0000111121314	1001011121314
3	6	6	5
0000110	1000102122232	0000111213141	00001111222
1000111	10010202-12-22-3	000101-11-21-31-4	000101-12-12-2
100101-1	0001020303132	1001020304041	00010112122
1001011	1000102031020	1000102030410	10001101-12-1
6	7	5	6
0000110112030	100011011213141	00001021222	1000102122232
1000102031213	0000111101-11-21-	00010202-12-2	10010202-12-22-3
100102-1203-130	000102030404131	00010202122	0001020303132
1000110111213	000010203041011	00001021020	1000102031020

Table 1: Offsets for all 12 pieces.

Perm(n)	11	5	9	6	10	7	8	4	3	2	1	0
Perm(n+1)	5	11	9	6	10	7	8	4	3	2	1	0
Perm(n+2)	9	11	5	6	10	7	8	4	3	2	1	0
Perm(n+3)	11	9	5	6	10	7	8	4	3	2	1	0
Index	0	1	2	3	4	5	6	7	8	9	10	11

Table 2: Four selected permutations.

Conclusions

Finally when the compiler optimization is turned on, the time drops to about 45 seconds to find all 22 solutions. The time can be reduced even more by removing recursion and moving the auxiliary procedures into the solve routine to save the creation of activation records. The time has been reduced to less than 9 seconds on a 1.33 MHz machine by storing the board, pieces, and rewriting the procedures using bit numbers and bit manipulation. The bit manipulation procedures were also moved into the solve procedure to save creation of activation records. **Once the algorithm of solving the checkerboard puzzle is presented, it seems simple and straight forward. Prior to presenting the algorithm it seems like an insurmountable problem to most beginning Computer Science students.**

Reference

[1] Robert L. Kruse (1984), *Data Structure and Program Design*, Prentice Hall, P. 269

Appendix A

C++ Program

```
#include <fstream.h>
#include <iomanip.h>

class piece {
public:
     int number;
     int rotations[4][15];
};

class ckbd {
public:
     ckbd();
     void placepiece(int row, int column, int loc, int rot);
     void removepiece(int row, int column, int loc, int rot);
     bool checkpiece(int row, int column, int loc, int rot, int & color);
     void locateposition(int  &  row, int & column);
     void printboard();
     void ReadPieces();
     void generate_permutation(int n);
     void solveit(int & loc, int row, int column, int & color, int & count);
     void printpermute();
     int numberoftimes;
     int list[13];
     int count;
     bool dif;
     int oldlist[13];
     int board[8][8];
     piece pieces[12];
};

ckbd::ckbd() {                          /* initialized the permutation array and saves a copy*/
     for(int i=0;i<=12;i++) {           /* the duplicate array is used to see if the next  permutation */
          list[i] = 12-i;               /* compares for optimization*/
     oldlist[i] = list[i];
     }
   oldlist[12]=1;
     count=12;
   for (int r=0;r<8;r++)
        for (int c=0;c<8;c++)
             board[r][c]=-1;
     numberoftimes = 1;
     dif = false;
```

```
}

void ckbd::placepiece(int row,int column,int loc,int rot){     /* place the piece*/
     for (int i=1; i<2*pieces[list[loc]].number; i=i+2)
          board[row+pieces[list[loc]].rotations[rot][i]][column+pieces[list[loc]].rotations[rot][i+1]]=list[loc];
}

void ckbd::removepiece(int row,int column,int loc,int rot){  /* remove the piece*/
     for (int i=1; i<2*pieces[list[loc]].number; i=i+2)
          board[row+pieces[list[loc]].rotations[rot][i]][column+pieces[list[loc]].rotations[rot][i+1]]=-1;
}

bool ckbd::checkpiece(int row,int column, int loc, int rot, int &color){         /* checks to see if all the */
     if((((row+column)%2==0)&&(pieces[list[loc]].rotations[rot][0]!=0))||    /* squares are available */
        (((row+column)%2==1)&&(pieces[list[loc]].rotations[rot][0]!=1)))
       return false;
     for (int b=1;b<(pieces[list[loc]].number * 2); b= b + 2) {
          int rownumber=pieces[list[loc]].rotations[rot][b];
          int columnnumber=pieces[list[loc]].rotations[rot][b+1];
          if ((rownumber + row < 0) || (rownumber + row > 7)||
             (columnnumber + column < 0) || (columnnumber + column > 7))
            return false;
          if(board[rownumber + row][columnnumber + column]!=-1)
            return false;
     }
     return true;
}



void ckbd::locateposition(int & row,int & column){       /* locates the location for placing the next piece*/
     for (column;column<8;column++)
          if(board[row][column]==-1)
            return;
     for(row;row<8;++row)
          for (column=0;column<8;column++)
               if (board[row][column] == -1)
                    return;
}

void ckbd::printboard(){
     for (int r=0;r<8;r++) {
          for (int c=0;c<8;c++)
               cout << setw(3) << board[r][c];
          cout << endl;
     }
```

```
    cout << endl<<endl;
}

void ckbd::ReadPieces(){
    char num[80];
    char filename[30]="puzzle.dat";
    fstream infile;
    infile.open(filename, ios::in);
    if (!infile)
       cout << "Cannot open file  " << endl;
    else {
        int count=0;
        int size;
        while (!infile.eof()) {
            infile.getline(num,80);
            size = num[0]-48;
            pieces[count].number=size;
            for (int rot=0;rot<=3;rot++) {
                infile.getline(num,80);
                int ct = 0;
                for(int h=0;h<=size*2;h++){
                    if(num[ct]=='-') {
                        ct++;
                        pieces[count].rotations[rot][h]=(num[ct]-48) * -1;
                    }
                    else
                        pieces[count].rotations[rot][h]=(num[ct]-48);
                    ct++;
                }
            }
            count++;
        }
    }
    infile.close();
}

void ckbd::generate_permutation(int n){                    /*Generates the permutations*/
    int c;
    int t;
    c = 1;
    if(n > 2)
        generate_permutation(n-1);
    else {
        dif = false;
        for ( int i = 12; i >= count; i--)
            if ( list [i] != oldlist[i])
```

```
                dif = true;
        if (dif)  {
            for ( i = 0; i <= 12; i++)
                oldlist[i] = list[i];
            int loc =12;
            int color=0;
            count = 12;
            solveit(loc, 0, 0, color, count);
        }
    }
    while (c < n ) {
        if (n%2==0) {
            t = list[n];
            list[n] = list[c];
            list[c] = t;
        }
        else {
            t = list[n];
            list[n] = list[1];
            list[1] = t;
        }
        c = c+1;
        if (n > 2)
            generate_permutation(n-1);
        else {
          dif = false;
          for ( int i = 12; i >= count; i--)
            if ( list [i] != oldlist[i])
                dif = true;
          if (dif) {
            for ( i = 0; i <= 12; i++)
                oldlist[i] = list[i];
            int loc =12;
            int color=0;
            count = 12;
            solveit(loc, 0, 0, color, count);
          }
        }
    }
}

void ckbd::solveit(int &loc, int row, int column, int &color, int &count){   /* recursive procedure to*/
    int times;                                                               /* find solutions*/
    if (loc == 0) {
        printboard();
        cout << numberoftimes << endl;
```

```
            numberoftimes++;
        }
        else {
            locateposition(row,column);
            if(list[loc]==0)
                times=2;
            else
                times=4;
            for(int rot=0;rot<times;rot++) {
                if (checkpiece(row, column, loc, rot, color)) {
                    placepiece(row,column,loc,rot);
                    loc--;
                    if (count > loc)
                        count = loc;
                    solveit(loc,row,column,color,count);
                    loc++;
                    removepiece(row,column,loc,rot);
                }
            }
        }
}

void ckbd::printpermute() {
    for(int j=1;j<=12;j++)
        cout<<list[j]<<"  ";
    cout<<endl;
}

#include <iostream.h>
#include "checker.h"

void main()
{
    ckbd myboard;
    int n=12;
    myboard.ReadPieces();
    myboard.generate_permutation(n);
}
```