Rubik's Cube Solver and Optimal Solution Finder

Richard Seale

Project Description

This project is comprised of two parts: solving the 3x3x3 Rubik's Cube and finding the greatest number of moves required to solve the 3x3x3 Rubik's Cube optimally. This is an intriguing project since the maximum number of actions required to solve any Rubik's Cube has only been narrowed between 20 and 22 moves. That is, the maximum solution is known to be no higher than 22 moves, yet no solution requiring more than 20 moves has been found[1].

In solving the Rubik's Cube, acceptable solutions are available in two forms. In the first mode, which is enabled by default, the solution must have exactly the specified number of actions, without allowing the Rubik's Cube to fall into an already visited state. In the second mode, enabled by including the fact "testOptimal", all possible solutions of the given Rubik's Cube puzzle that require fewer than or equal to the specified number of actions are found. In this way, a generalized Rubik's Cube generator can be used to find all puzzle states that can be solved in n-many moves using the first solution form, and then each of those states may be checked using the second solution form, to see if it the solution is optimal.

The two-mode solver is necessary for the second, more ambitious, part of the project: discovering the maximum number of moves required to solve any puzzle optimally. Using the default mode with a random puzzle generator, it is possible to find all initial puzzle states that are solvable using exactly 21 or 22 moves. From these initial puzzle states, it is then possible to use the "testOptimal" mode to check if the puzzle can be solved using fewer moves. If any puzzle cannot be solved using fewer moves than it took to create the initial state, then we know that the original solution is optimal.

Project Design

The project is designed to consider the location of the moving tiles rather than the position of the colored faces. This design approach reduces the search size from considering 48 moving tiles to only 20 moving blocks. Additionally, this allows limitations to be placed on the tiles, since a corner block can never be anything but a corner block. These limitations are also possible when using tiles, but the implementation is more complex, rather than less.

The design is based on the theory that the Rubik's Cube will be at a solved state when all blocks are placed into their solved positions, the puzzle will be solved. However, the theory makes the assumption that it is impossible for any block to be rotated independently of all other blocks. In other words, it assumes that if a block has been rotated into a non-solved state, then another block must be out of place. This has not, however, been shown to be correct.

The design has one additional, less obvious, flaw. If the assumption made above is correct, then the solved Rubik's Cube would, by necessity, be an oriented solution of the puzzle. That is, if the arrows were drawn on the colored tiles of the Cube, and the Cube was subsequently solved by the solver, the arrows would end pointing in the same direction as they began (relative to the Cube). This is not a problem, of course, but solving an oriented puzzle is generally more complex than solving a non-oriented puzzle, so the numbers of moves required to solve the puzzle may or may not be greater than the number required to solve the puzzle non-oriented.

The solver is designed to pick an action for each time interval in the range of 0 to n-1, for a total of n actions. Actions are specified by designating a "face" – 0 to 5 – and a "move"– clockwise, counterclockwise, or 180 degree twist. "Combination" actions are also available for faces 0, 1, and 2 to represent rotating a middle section as one action. Certain actions are also

restricted to multiple actions from taking the place of a single action (i.e. three clockwise twists = one counter clockwise twist). No pair of sequential actions can take place on the same face . This includes both affected faces of combination actions. Likewise, no action should take place on a face if the only action to take place since the face's last action occurred on the opposite face – moving face 0 clockwise, then face 5 clockwise, then face 0 clockwise again is the same as moving face 0 by 180 degrees and then face 5 clockwise. Finally, no action should place the Cube in a state that has already been visited, since that would clearly not be an optimal solution. Although these restrictions do not guarantee optimality, they do eliminate a large amount of redundancy that would otherwise occur.

In the default mode, the solver ensures that every time interval is assigned some action, so that the ending solution is comprised of exactly n moves. In the testOptimal mode, the solver allows time intervals to have null actions, but only if the puzzle has reached a solved state by that time interval. This allows an easy check to determine whether the n-action solution is optimal or not.

Solutions are tested as follows. First, a generic Rubik's Cube generator is used as input to the default-mode solver. From this, all initial states which can be solved with n moves are generated and written to file. A C-based parser evaluates each line of the solver's output and recreates the Rubik's Cube's initial state. The parser then runs LParse using the single, recreated initial state file and the testOptimal mode of the solver with a maximum number of moves equal to n-1, and echoes the results to a new file. The parser then evaluates the results to see if a solution was found. If no solution was generated, then the original solution was optimal, and the actions that solved the puzzle (i.e. the reverse of the actions to generate the puzzle from a solved state) are then appended to an optimal results file.

For the sake of uniqueness, and space, the parser only considers solutions whose final action is performed on face 0. This eliminates solutions which are merely symmetric duplications of the solutions considered. That is, each solution ending with an action on face 0 has an equivalent, symmetrically identically, solution for each other face, which are not considered.

Implementation

The project is divided into four primary files, with some additional files being generated on-the-fly by the parser module.

The Rubik.lp file contains LParse commands to generate a random Rubik's Cube arrangement. Each of the 20 movable blocks of the Rubik's Cube is given a number, 1-20, such that the corner blocks are assigned from 1 through 8 and the edge pieces are assigned from 9 through 20. Because it is quite impossible for an edge to become a corner, or vice versa, the assignment commands were implemented to distinguish corner assignments and edge assignments. This prevents many definitely unsolvable solutions from being generated, and reduces the size of the search space when attempting to solve the puzzle. The file also includes, as comments, a diagram of how the blocks are laid out by face in the solved state, with face identifiers pre-appended with 'F'.

The Shift.lp file contains LParse commands which represent the changes made by an action on each face. The changes, or shifts, are split into eshift and cshift, to distinguish between corner and edge changes. This distinction becomes useful when generating solutions because it reduces the search space slightly. That is, comparing corners with corners has 8 * 8 possibilities, and comparing edges with edges has 12 * 12 possibilities, for a total search size of 144 + 64 = 208 possibilities, while comparing blocks with blocks = 20 * 20, which is a total search size of

400 possibilities. The shift commands are laid out in the form xshift(face, initial assignment, new assignment), and are based on a clockwise action being performed on the given face. Counterclockwise actions treat the initial assignment as new, and vice versa. 180-degree actions perform the clockwise action twice, and are a type of combination action.

The Action.lp file is responsible for assigning actions to each time increment, generating the resulting block assignment states, checking for valid action assignments, checking for a repeating Cube state, and checking for a solved state. In default mode, a “do” action is guaranteed to be generated for every time increment, except the last increment, which should hold the final solution. Contrarily, the testOptimal mode is guaranteed to be generated only for time increments prior to arriving at the solved state. The same is true for the “front” command, which defines the face on which the “do” action is to be performed. The file ensures that no location can have multiple assignments, which also guarantees that no block can be assigned to more than one location. The file is setup to generate the state of the Rubik’s Cube at all time intervals, even if a solution is found earlier while in testOptimal mode.

The RubiksCube.c file and its associated UtilFuncs.c are responsible for parsing the generated results of the default Rubik’s Cube solver, as described above. The parser attempts to open the file ResultsX.txt, where X should be replaced by the value of n when generating the results file. The parser will search the file, line by line, to find each solution for which the n-1 action is performed on face 0. The parser then generates a new file, newResults.txt, which is used in conjunction with the action.lp and shift.lp file to determine if the solution found is optimal. If the solution is optimal, then the parser appends the actions taken by the solution to a file named NewResults.txt, which it creates if the file is not found.

Results

For a single action, LParse discovered 27 puzzles that can be solved using exactly one action. All single action solution variants were discovered to be optimal, with six symmetrically unique optimal solutions. The default solver completed generating puzzle possibilities in 1.263 seconds.

For two actions, LParse discovered 522 puzzles that can be solved using exactly two actions. All two-action solutions were discovered to be optimal, with 114 symmetrically unique optimal solutions. The default solver completed generating puzzle possibilities in 4.367 seconds.

For three actions, LParse discovered 10044 puzzles that can be solved using exactly three actions. 2180 three-action solutions were discovered to be symmetrically unique optimal solutions. The default solver completed generating puzzle possibilities in 59.873 seconds.

For four actions, LParse discovered 193314 puzzles that can be solved using exactly four actions. 41464 four-action solutions were discovered to be symmetrically unique optimal solutions. The default solver completed generating puzzle possibilities in 1296.851 seconds.

For five actions, LParse discovered 3720582 puzzles that can be solved using exactly five actions. The C based parser ran for more than a week without completing its search for symmetrically unique optimal solutions before the power went out in my apartment. Since the projected runtime was greater than the amount of time remaining before the project deadline, the program was not reattempted. 103569 unique optimal solutions had been discovered prior to termination. The default solver completed generating puzzle possibilities in 27650.459 seconds.

For six actions, LParse discovered 71607360 puzzles that can be solved using exactly six actions. The parser program has not been applied to the six-action initial states. The default solver completed generating puzzle possibilities in 612249.213 seconds.

Tests for more than six actions were not run for this project, since the project deadline did not allow for the amount of time such a test would require. 21 actions was attempted, but had not completed prior to the aforementioned power outage. For 21 actions, 37986228 solutions were found prior to program termination.

References

1. http://en.wikipedia.org/wiki/Rubik's_Cube

Appendix A: Rubik.lp

```
%        **********
%        *07 20 08*
%        *       *
%        *16 F5 18*
%        *       *
%        *05 13 06*
%        **********
%        *05 13 06*
%        *       *
%        *14 F1 15*
%        *       *
%        *01 09 02*
%****************************
%*05 14 01*01 09 02*02 15 06*
%*       *       *       *
%*16 F2 10*10 F0 11*11 F3 18*
%*       *       *       *
%*07 17 03*03 12 04*04 19 08*
%****************************
%        *03 12 04*
%        *       *
%        *17 F4 19*
%        *       *
%        *07 20 08*
%        **********

edge( 9..20 ).
corner( 1..8 ).

#domain edge( E ), corner( C ).

% generate a random, "solvable" rubik's cube
1{ cAssign( C, C0, 0 ) : corner( C0 ) }1.
1{ eAssign( E, E0, 0 ) : edge( E0 ) }1.

hide edge(_).
hide corner(_).
```

Appendix B: shift.lp

```
% face 0 rotation
cshift( 0,  1,  2 ).
cshift( 0,  2,  4 ).
cshift( 0,  4,  3 ).
cshift( 0,  3,  1 ).

eshift( 0,  9, 11 ).
eshift( 0, 11, 12 ).
eshift( 0, 12, 10 ).
eshift( 0, 10,  9 ).

% face 1 rotation
cshift( 1,  5,  6 ).
cshift( 1,  6,  2 ).
cshift( 1,  2,  1 ).
cshift( 1,  1,  5 ).

eshift( 1, 13, 15 ).
eshift( 1, 15,  9 ).
eshift( 1,  9, 14 ).
eshift( 1, 14, 13 ).

% face 2 rotation
cshift( 2,  5,  1 ).
cshift( 2,  1,  3 ).
cshift( 2,  3,  7 ).
cshift( 2,  7,  5 ).

eshift( 2, 14, 10 ).
eshift( 2, 10, 17 ).
eshift( 2, 17, 16 ).
eshift( 2, 16, 14 ).

% face 3 rotation
cshift( 3,  2,  6 ).
cshift( 3,  6,  8 ).
cshift( 3,  8,  4 ).
cshift( 3,  4,  2 ).

eshift( 3, 15, 18 ).
eshift( 3, 18, 19 ).
eshift( 3, 19, 11 ).
eshift( 3, 11, 15 ).

% face 4 rotation
```

```
cshift( 4,  3,  4 ).
cshift( 4,  4,  8 ).
cshift( 4,  8,  7 ).
cshift( 4,  7,  3 ).

eshift( 4, 12, 19 ).
eshift( 4, 19, 20 ).
eshift( 4, 20, 17 ).
eshift( 4, 17, 12 ).

% face 5 rotation
cshift( 5,  7,  8 ).
cshift( 5,  8,  6 ).
cshift( 5,  6,  5 ).
cshift( 5,  5,  7 ).

eshift( 5, 20, 18 ).
eshift( 5, 18, 13 ).
eshift( 5, 13, 16 ).
eshift( 5, 16, 20 ).

hide eshift(_,_,_).
hide cshift(_,_,_).
```

Appendix C: action.lp

```
time( 0..n ).

#domain time(T).

face( 0..5 ).

edge( 9..20 ).
corner( 1..8 ).

#domain edge( E;EA ), corner( C;CA ).
#domain face( F ).

move( twist_ccw; twist_cw; twist_180 ).
move( twist_middle_cw; twist_middle_ccw; twist_middle_180 ).

1{ do( A, T ) : move( A ) }1 :- T < n, not finish( T ), testOptimal.
1{ front( F0, T ) : face( F0 ) }1 :- T < n, not finish( T ), testOptimal.

1{ do( A, T ) : move( A ) }1 :- T < n, not testOptimal.
1{ front( F0, T ) : face( F0 ) }1 :- T < n, not testOptimal.

% prevent actions from occuring on the same face twice consecutively, to maintain uniqueness
:- front( F, T ), front( F, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_cw, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_ccw, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_180, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_cw, T ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_ccw, T ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_middle_180, T ).
% prevent actions from occuring on a face if the only action that has occurred is on the opposite
face, since that would be the same as the case above
:- front( F, T ), front( 5 - F, T - 1 ), front( F, T - 2 ).

% only allow middle rotations for faces 0-2 - reduces the search space slightly
:- do( twist_middle_cw, T ), front( F, T ), F > 2.
:- do( twist_middle_ccw, T ), front( F, T ), F > 2.
:- do( twist_middle_180, T ), front( F, T ), F > 2.

% prevent representing a middle rotation as two actions on opposite faces
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_ccw, T ), do( twist_cw, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_cw, T ), do( twist_ccw, T - 1 ).
:- front( F, T ), front( 5 - F, T - 1 ), do( twist_180, T ), do( twist_180, T - 1 ).

% face rotations
```

```
cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_cw, T - 1 ), cshift( F, C0, C ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_cw, T - 1 ), eshift( F, E0, E ).

cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_ccw, T - 1 ), cshift( F, C, C0 ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_ccw, T - 1 ), eshift( F, E, E0 ).

cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0;C1 ), do( twist_180, T - 1 ), cshift( F, C, C1 ), cshift( F, C1, C0 ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0;E1 ), do( twist_180, T - 1 ), eshift( F, E, E1 ), eshift( F, E1, E0 ).

% middle rotations
cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_middle_cw, T - 1 ), cshift( F, C, C0 ).
cAssign( C, CA, T ) :- front( 5 - F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_middle_cw, T - 1 ), cshift( F, C0, C ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_middle_cw, T - 1 ), eshift( F, E, E0 ).
eAssign( E, EA, T ) :- front( 5 - F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_middle_cw, T - 1 ), eshift( F, E0, E ).

cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_middle_ccw, T - 1 ), cshift( F, C0, C ).
cAssign( C, CA, T ) :- front( 5 - F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0 ), do( twist_middle_ccw, T - 1 ), cshift( F, C, C0 ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_middle_ccw, T - 1 ), eshift( F, E0, E ).
eAssign( E, EA, T ) :- front( 5 - F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0 ), do( twist_middle_ccw, T - 1 ), eshift( F, E, E0 ).

cAssign( C, CA, T ) :- front( F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0;C1 ), do( twist_middle_180, T - 1 ), cshift( F, C, C1 ), cshift( F, C1, C0 ).
cAssign( C, CA, T ) :- front( 5 - F, T - 1 ), cAssign( C0, CA, T - 1 ), corner( C0;C1 ), do( twist_middle_180, T - 1 ), cshift( F, C, C1 ), cshift( F, C1, C0 ).
eAssign( E, EA, T ) :- front( F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0;E1 ), do( twist_middle_180, T - 1 ), eshift( F, E, E1 ), eshift( F, E1, E0 ).
eAssign( E, EA, T ) :- front( 5 - F, T - 1 ), eAssign( E0, EA, T - 1 ), edge( E0;E1 ), do( twist_middle_180, T - 1 ), eshift( F, E, E1 ), eshift( F, E1, E0 ).


% check for unique state
```

```
isDifferent( T, T0 ) :- cAssign( C, CA, T ), cAssign( C, CA0, T0 ), time( T0 ), corner( CA0 ), T > T0, CA != CA0.
isDifferent( T, T0 ) :- eAssign( E, EA, T ), eAssign( E, EA0, T0 ), time( T0 ), edge( EA0 ), T > T0, EA != EA0.

:- do( A, T - 1 ), move( A ), time( T0 ), T > T0, not isDifferent( T, T0 ).

:- 2{ cAssign( C, C0, T ) : corner( C0 ) }.
:- { cAssign( C, C0, T ) : corner( C0 ) }0.

:- 2{ eAssign( E, E0, T ) : edge( E0 ) }.
:- { eAssign( E, E0, T ) : edge( E0 ) }0.

{ cAssign( C, C0, T ) } :- cAssign( C, C0, T - 1 ), corner( C0 ).
{ eAssign( E, E0, T ) } :- eAssign( E, E0, T - 1 ), edge( E0 ).

fail( T ) :- not cAssign( C, C, T ).
fail( T ) :- not eAssign( E, E, T ).

finish( T ) :- not fail( T ).

:- not finish( n ).

initAssign( C, C0, n ) :- cAssign( C, C0, 0 ), corner( C0 ).
initAssign( E, E0, n ) :- eAssign( E, E0, 0 ), edge( E0 ).

hide time(_).
hide face(_).
hide move(_).
hide corner(_).
hide edge(_).
hide cAssign(_,_,_).
hide eAssign(_,_,_).
hide isDifferent(_,_).
hide finish(_).
hide fail(_).
show initAssign(_,_).
show do(_,_).
show front(_).
hide testOptimal.
```

Appendix D: Rubik.c

```
#ifdef UNICODE
#undef UNICODE
#endif

#include <stdio.h>
#include <conio.h>
#include <windows.h>
#include <direct.h>
#include <string.h>
#include "UtilFuncs.h"
#include "Rubik.h"

void main()
{
  int i, len = 0, resultIsValid = 1, knownOptimal = 1, isOptimal = 2;
  FILE* fp = 0;
  char filename[ 5000 ], *line, *tline, *tline2, buffer[ 1000 ];

  SetCurrentDirectory( PATH );

  for( i = 1; i < 6; i++ )
  {
    __int64 flen, fpos;

    sprintf( filename, "results%d.txt", i );
    fp = fopen( filename, "rb+" );

    if( fp )
    {
      FILE* ftemp;

      flen = fgetlen( fp );

      do
      {
        fgetLine( fp, &line, 5000 );

        fgetpos( fp, &fpos );
        //printf( "%d:  % 2.5f%% complete\r", i, fpos * 100.0f / flen );

        if( line )
        {
          if( !memcmp( line, "Answer", 6 ) )
          {
            printf( "%d\t%s\r", i, line );
```

```
  }
  else if( !memcmp( line, "Duration", 8 ) )
  {
    printf( "\n%d\t%s\n", i, line );
    continue;
  }
}

if( !line || memcmp( line, "Stable Model:", 13 ) )
{
  // if nothing was read, or if this line is not a result line, ignore the line
  continue;
}

sprintf( buffer, "front(0,%d)", i-1 );
if( !strstr( line, buffer ) )
{
  // if the first action is performed a face other than face 0, ignore the result
  // this eliminates some symmetrical duplication
  continue;
}

tline = line;

ftemp = fopen( "rubikInput.lp", "wb+" );

while( tline = strstr( tline, "initAssign" ) )
{
  int face, assign;
  tline = strchr( tline, '(' ) + 1;

  tline2 = strchr( tline, ',' );
  *tline2 = 0;
  face = atoi( tline );

  tline = tline2 + 1;

  tline2 = strchr( tline, ',' );
  *tline2 = 0;
  assign = atoi( tline );

  tline = tline2 + 1;

  if( face < 9 )
  {
    sprintf( buffer, "cAssign(%d,%d,0).\n", face, assign );
```

```
      }
      else
      {
        sprintf( buffer, "eAssign(%d,%d,0).\n", face, assign );
      }

      fwrite( buffer, sizeof(char), strlen(buffer), ftemp );
    }

    sprintf( buffer, "testOptimal.\n" );
    fwrite( buffer, sizeof(char), strlen(buffer), ftemp );

    fclose( ftemp );

    sprintf( filename, "results%d.x", i );

    // check all possible solutions that are less than i actions
    sprintf( buffer, "lparse -c n=%d action.lp shift.lp rubikInput.lp |smodels 1 > %s", i - 1,
filename );
    system( buffer );

    ftemp = fopen( filename, "rb" );
    if( !ftemp )
    {
      printf( "\nCould not find results. Press enter to exit.\n" );
      _getche();
      return;
    }

    memset( buffer, 0, sizeof(buffer) );
    fread( buffer, sizeof(char), sizeof(buffer), ftemp );
    fclose( ftemp );

    if( strstr( buffer, "do(twist_" ) )
    {
      // if a solution was found then the result is not optimal
      resultIsValid = 0;
      isOptimal = 0;
    }
    else
    {
      resultIsValid = 1;
    }

    if( resultIsValid )
    {
```

```
        ftemp = fopen( "newResults.txt", "ab" );

        tline = line + 14;
        tline2 = strstr( line, "initAssign" );
        *( tline2 - 1 ) = '\n';
        *tline2 = 0;
        fwrite( tline, sizeof(char), strlen( tline ), ftemp );

        fclose( ftemp );
       }
     } while( line && memcmp( line, "Duration", 8 ) );

     fclose( fp );
     fp = 0;

     test_and_free( (void**)&line );
    }

    if( isOptimal )
    {
     knownOptimal = imax( i, knownOptimal );
     printf( "Known Optimal Solution = %d\t\t\n", knownOptimal );
    }
   }

  getche();
}
```

Appendix E: UtilFuncs.c

```
#include <io.h>
#include <string.h>
#include <stdlib.h>
#include "UtilFuncs.h"

unsigned __int64 fgetlen
(
 FILE* fp
 )
{
  if( !mem_test( fp ) )
  {
    return 0;
  }

  return _filelengthi64( fp->_file );
}

int fgetLine
(
 FILE* fp,
 char** retStr,
 unsigned int maxSize
 )
{
  char *temp, *str, lineIn[ 1025 ] = "";
  unsigned int size = 0, rLen = 0, sLen = 0, offset;
  unsigned __int64 pos, fsize;

  test_and_free( retStr );

  if( mem_test( fp ) )
  {
    fgetpos( fp, &pos );
    fsize = fgetlen( fp );
    fsetpos( fp, &pos );
    if( pos >= fsize )
    {
      return 0;
    }

    do
    {
      temp = lineIn;
      rLen = fread( temp, 1, 1024, fp );
```

```
    // ensure null termination
    temp[ rLen ] = 0;

    // remove preceeding invalid characters
    sLen = 0;
    offset = 0;
    while( sLen < rLen && ( temp[ sLen ] == '\r' || temp[ sLen ] == '\f' || temp[ sLen ] == '\n' ||
temp[ sLen ] == 0 ) )
    {
     temp++;
     rLen--;
     offset++;
    }

    if( rLen )
    {
     // check for line feeds, carriage returns, and newlines
     char* nstr = strchr( temp, '\n' );
     char* lstr = strchr( temp, '\f' );
     char* rstr = strchr( temp, '\r' );

     str = temp + rLen;

     if( nstr )
     {
      str = (char*) imin( (int) nstr, (int) str );
     }

     if( lstr )
     {
      str = (char*) imin( (int) lstr, (int) str );
     }

     if( rstr )
     {
      str = (char*) imin( (int) rstr, (int) str );
     }

     sLen = str - temp;
    }

    if( sLen > 0 )
    {
     array_concat( retStr, &size, temp, imin( sLen, maxSize - size - 1 ), 1 );
    }
```

```
      if( sLen < rLen )
      {
        while( sLen < rLen && ( temp[ sLen ] == '\r' || temp[ sLen ] == '\f' || temp[ sLen ] == '\n' ||
temp[ sLen ] == 0 ) )
        {
          sLen ++;
        }

        pos += sLen + offset;
        fsetpos( fp, &pos );
        rLen = 0;
      }
    }while( rLen && size < maxSize );
  }

  *temp = 0;
  array_append( retStr, &size, temp, 1 );

  return size;
}

void test_and_free
(
 void** temp
 )
{
  if( mem_test( *temp ) )
  {
    free( *temp );
  }
  *temp = 0;
}

int imax
(
 int x,
 int y
 )
{
  return x > y ? x : y;
}

int imin
(
 int x,
```

```
 int y
 )
{
  return x < y ? x : y;
}

int mem_test
(
 void* mem
 )
{
  return mem && (int)mem != 0xcdcdcdcd && (int)mem != 0xfdfdfdfd && (int)mem !=
0xcccccccc && (int)mem != 0xdddddddd;
}

void array_concat
(
 char** array1,
 int* size1,
 char* array2,
 int size2,
 int dataSize
 )
{
  int count = *size1 * dataSize;
  char *temp = 0, *t;
  int n_size = ( *size1 + size2 ) * dataSize;
  n_size += 1024 - n_size % 1024;

  if( !mem_test( *array1 ) )
  {
    *array1 = 0;
  }

  temp = allocate( n_size );
  memcpy( temp, *array1, count );
  memcpy( ( temp + count ), array2, size2 * dataSize );

  t = *array1;
  *array1 = temp;

  *size1 += size2;

  test_and_free( &t );
}
```

```
void array_append
(
 char** array,
 int* size,
 char* obj,
 int dataSize
 )
{
  int count = *size * dataSize;
  char* temp = 0, *t;
  int n_size = ( *size + 1 ) * dataSize;
  n_size += 1024 - n_size % 1024;

  if( !mem_test( *array ) )
  {
    *array = 0;
  }

  temp = allocate( n_size );
  memcpy( temp, *array, count );
  memcpy( ( temp + count ), obj, dataSize );

  t = *array;
  *array = temp;

  (*size)++;

  test_and_free( &t );
}

void* allocate
(
 unsigned int size
 )
{
  void* temp;
  if( size )
  {
    temp = malloc( size );
    if( temp )
    {
      memset( temp, 0, size );
    }
    return temp;
  }
  return 0;
```

}