CE437: Αλγόριθμοι CAD I

Homework 4

Tcl shell's Implementation

By Vasileiou Christos, 1983

Files' Structure

- customTCL.c: Includes main implemenation.
    - int main(int argc, char *argv[])
- Instructions.h: Includes Tcl instructions in a string array.
    - static char *instructions[]
- functions_1st.c: Includes 1st homework's functions.
- functions_2nd.c: Includes 2nd homework's functions.
    - void *commandsCreation();
    - Tcl_ObjCmdProc *cube_intersect_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[])
    - Tcl_ObjCmdProc *supercube_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *distance_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *cube_cover_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *my_sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - int checkIfValid (char *checked, int size);
- Makefile: Linking and Compilation.

Files' Structure

- functions_3rd.c: Includes 3rd homework's functions.
  - void *do_read_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
  - void *initIntArray (int x, int y, int array [x][y], int val);
  - void *initInt (int x, int n[x], int val);
  - int searchNodes (int x, int n[x], int n2); // return 1 if n2 is not in nD. //
  - void *printGraph (int x, int graph [x][x]);
  - void *do_write_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
  - void *do_draw_graph (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
  - nodesDist_t find_Shortest_Explored_Node (int x, nodesDist_t *d);
  - int maximum (int a, int b);
  - int graphIsNotEmpty (nodesDist_t *n); // return 0 if n is full. //
  - void sortGraph (int x, nodesDist_t *n);
  - void *do_graph_critical_path (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
  - int minimum (int a, int b);
  - nodesDist_t *back_trace (nodesDist_t *Q, int arcWeight[size][size], int longest_path, int Rslack, int *previous, int maxDistanceNode, int *slack, nodesDist_t *criticalPath);

Files' Structure

- functions_4th.c: Includes 4th homework's functions.
  - char *coFactor (int size, char *arg1, char *arg2, char *arg3);
  - void initial (int size, char *array, char val);
  - void AND (int size, char *arg1, char *arg2, char *arg3);
  - void OR (int size, char *arg1, char *arg2, char *arg3);
  - int EQUAL (int size, char *arg1, char *arg2);
  - void DESTROY_CUBE (int size, char *array);
  - int EXIST (int size, char *arg1);
  - void printCube (int size, char *arg1);
  - void *algebraic_division (int size, int cubesDividendLength, char **cubesDividend, int cubesDivisorLength, char **cubesDivisor, int *FinalQuoLength, char **finalQuotient, int *RemLength, char **remainder);
  - void *do_alg_division (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
  - int find_SUPV (int size, char *arg1, int supportiveVarLength, char **supportiveVar);
  - int CUBES (int cubFunLen, int size, char **cubFun, char *supV, int pos, char ***cubListSUP, char **coKernel);
  - void redundant_kernels (int kernelsLevel, int CUBE_LENGTH, int cubesFunctionLength, char **cubesFunction, int *FinalQuoLength, char ***finalQuotient);
  - void *do_r_kernels (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);

Files for compilation and running

- compile_N_run:
  is a script for making compile and run customTCL using valgrind.
- tests.txt:
  contains examples of:
  1. alg_division
  2. r_kernels

cofactor and initial functions.

```
char *coFactor ( int size, char *arg1, char *arg2, char *arg3 )
  {
    int j, validity;
    char invertedArg2 [size+1];
    for ( j = 0; j < size; j++ )
      {
        // 2nd argument is inverted. //
        if ( strncmp ( &arg2 [j], "0", 1) == 0 )
          {
            strncpy ( &invertedArg2 [j], "1", 1 );
          }
» » » » else
          {
» » » » » strncpy ( &invertedArg2 [j], "0", 1 );
          }
        // arg1 OR inverted arg2. //
        if ( (strncmp ( &arg1 [j], "1", 1) == 0) || (strncmp ( &invertedArg2 [j], "1", 1) == 0) )
          {
            strncpy ( &arg3 [j], "1", 1 );
          }
» » » » else
          {
» » » » » strncpy ( &arg3 [j], "0", 1 );
          }
      }
    »strncpy ( &arg3 [j], "\0", 1 );
» » validity = checkIfValid ( arg3, size );
» » if ( validity > 0 )
» » » {
» » » » printf ("Intersect %s is not valid\n", arg3 );
» » » » return NULL;
» » » }
» » return arg3;
  }
```

```
void initial ( int size, char *array, char val )
  {
    int j;
    for ( j = 0; j < size; j++ )
      {
        array [j] = val;
      }
    array [j] = '\0';
  }
```

AND, OR and EQUAL functions.

```
void AND ( int size, char *arg1, char *arg2, char *arg3 )
  {
    int j;
    for ( j = 0; j < size; j++ )
      {
        if ( (arg1 [j] == '1') && (arg2 [j] == '1') )
          {
            arg3 [j] = '1';
          }
        else
          {
            arg3 [j] = '0';
          }
      }
    arg3 [j] = '\0';
  }
```

```
void OR ( int size, char *arg1, char *arg2, char *arg3 )
  {
    int j;
    for ( j = 0; j < size; j++ )
      {
        if ( (arg1 [j] == '1') || (arg2 [j] == '1') )
          {
            arg3 [j] = '1';
          }
        else
          {
            arg3 [j] = '0';
          }
      }
    arg3 [j] = '\0';
  }
```

```
int EQUAL ( int size, char *arg1, char *arg2 )
  {
    int j;
    for ( j = 0; j < size; j++ )
      {
        if ( arg1 [j] != arg2 [j] )
          {
            return 0;
          }
      }
    return 1;
  }
```

Destroy_cube, exist and printCube functions.

```
void DESTROY_CUBE ( int size, char *array )
  {
    initial ( size, array, '\0' );
  }

int EXIST ( int size, char *arg1 )
  {
    int j;
    for ( j = 0; j < size; j++ )
      {
        if ( arg1 [j] != '\0' )
          {
            return 1;
          }
      }
    return 0;
  }
```

```
void printCube (int size, char *arg1 )
  {
    int j;
    char ch = 'a';
    for ( j = 0; j < size; j=j+2 )
      {
        if ( (arg1 [j] == '0') && (arg1 [j+1] == '1') )
          {
            printf (GREEN"%c", ch);
          }
        else if ( (arg1 [j] == '1') && (arg1 [j+1] == '0') )
          {
            printf (GREEN"%c'", ch);
          }
        ch++;
      }
    printf(WHITE"");
  }
```

Algebraic_division function.

```
void *algebraic_division ( int size, int cubesDividendLength, char **cubesDividend,
                           int cubesDivisorLength, char **cubesDivisor, int *FinalQuoLength,
                           char **finalQuotient, int *RemLength, char **remainder )
{
  /* for each cube (i) of divisor */
  int i, j, validity, D, valid, k = 0, l, p, cntFinalQ, lengthDividend = 0, lengthQuotient = 0;
  bool full [cubesDivisorLength][cubesDividendLength];
  char ***quotient = (char***) malloc ( cubesDivisorLength * sizeof (char**) );
  // calculation partial quotients. //
  for ( i = 0; i < cubesDivisorLength; i++ )
    {
      quotient [i] = (char**) malloc ( cubesDividendLength * sizeof(char*) );
      lengthQuotient = 0;
      valid = 0;
      /* for each cube (j) of dividend */
      for ( j = 0; j < cubesDividendLength; j++ )
        {
          full [i][j] = false;
          /* check if there is dividend's cube (d) that contains (a) */
          /* if j does not contain i. */
          D = cube_cover ( size, cubesDivisor [i], cubesDividend [j] );
          quotient [i][j] = (char*) malloc ( (size+1) * sizeof (char) );
          if ( D == 0 )
            {
              continue;
            }
          /*************************************************************
           * Creation of an array with quotient of each divisor's cube *
           * in order to find the final quotient.                      *
           *************************************************************/
          quotient [i][j] = coFactor ( size, cubesDividend [j], cubesDivisor [i], quotient [i][j] );
          valid = 1;
          ++lengthQuotient;
          full [i][j] = true;
        }
    }
  // calculation final quotient. //
  cntFinalQ = 0;
  if ( r_kernels_enable == 0 )
    {
      printf ("quotients: ");
    }
```

Algebraic_division function.

```
for ( l = 0; ( (l < cubesDivisorLength) && (cubesDivisorLength > 1) ); l=l+2 )
  {
    for ( k = 0; k < cubesDividendLength; k++ )
      {
        for ( p = 0; p < cubesDividendLength; p++ )
          {
            /*****************************************************
             * if one of quotient [l][k], quotient [l+1][p] hasn't *
             * been calculated, go on to the next quotients.       *
             *****************************************************/
            if ( (full [l][k] != true) || (full [l+1][p] != true) )
              {
                continue;
              }
            /*****************************************************
             * Check if there is intersection between quotients. *
             *****************************************************/
            if ( EQUAL ( size, quotient [l][k], quotient [l+1][p] ) == 0 )
              {
                continue;
              }
            finalQuotient [cntFinalQ] = (char*) malloc ( (size+1) * sizeof (char) );
            strcpy ( finalQuotient [cntFinalQ], quotient [l][k] );
            validity = checkIfValid ( finalQuotient [cntFinalQ], size );
            if ( validity > 0)
              {
                printf ( RED"Divisor's quotient %s is not valid"WHITE"\n", finalQuotient [cntFinalQ] );
                return NULL;
              }
            printCube ( size, finalQuotient [cntFinalQ] );
            if ( r_kernels_enable == 0 && p < cubesDividendLength - 1 )
              {
                printf (", ");
              }
            else
              {
                printf ("\n");
              }
            ++cntFinalQ;
          }
      }
  }
```

```
if ( cubesDivisorLength == 1 )
  {
    finalQuotient [cntFinalQ] = (char*) malloc ( (size+1) * sizeof (char) );
    strcpy ( finalQuotient [cntFinalQ], quotient [0][0] );
    validity = checkIfValid ( finalQuotient [cntFinalQ], size );
    if ( validity > 0)
      {
        printf ( RED"Divisor's quotient %s is not valid"WHITE"\n", finalQuotient [cntFinalQ] );
        return NULL;
      }
    if ( r_kernels_enable == 0 )
      {
        printf ("%d quotient's cube: "GREEN"%s"WHITE"\n", cntFinalQ+1, finalQuotient [cntFinalQ] );
      }
    printCube ( size, finalQuotient [cntFinalQ] );
    if ( r_kernels_enable == 0 )
      {
        printf ("\n");
      }
    ++cntFinalQ;
  }
if ( r_kernels_enable == 0 )
  {
    printf ("\n");
  }
```

Algebraic_division.

```
// calculation remainder. //
char **quotient_divisor = (char**) malloc ( (cntFinalQ*cubesDividendLength) * sizeof(char*) );
int cnt = 0;
bool cubeExistInDividend = true, cubeExistInRemainder = false;
for ( i = 0; i < cubesDivisorLength; i++ )
  {
    for ( j = 0; j < cntFinalQ; j++ )
      {
        quotient_divisor [cnt] = (char*) malloc ( (size+1) * sizeof (char) );
        initial ( size, quotient_divisor [cnt], '1' );
        AND ( size, cubesDivisor [i], finalQuotient [j], quotient_divisor [cnt] );
        for ( k = 0; k < cubesDividendLength; k++ )
          {
            // Checking if P*Q cube is in dividend. //
            if (EQUAL ( size, cubesDividend [k], quotient_divisor [cnt] ) == 1)
              {
                DESTROY_CUBE ( size, cubesDividend [k] );
                break;
              }
          }
        cubeExistInDividend = true;
        cubeExistInRemainder = true;
        ++cnt;
      }
  }

int cntRem = 0;
```

```
  if ( r_kernels_enable == 0 )
    {
      printf ("remainders: ");
    }
  for ( k = 0; k < cubesDividendLength; k++ )
    {
      if ( EXIST ( size, cubesDividend [k] ) == 1 )
        {
          remainder [cntRem] = (char*) malloc ( (size+1) * sizeof (char) );
          initial ( size, remainder [cntRem], '\0' );
          strcpy ( remainder [cntRem], cubesDividend [k] );
          printCube ( size, remainder [cntRem] );
          if ( r_kernels_enable == 0 && k < cubesDividendLength - 1 )
            {
              printf (", ");
            }
          else
            {
              printf ("\n");
            }
          ++cntRem;
        }
    }

  if ( r_kernels_enable == 0 )
    {
      printf ("\n");
      *FinalQuoLength = cntFinalQ;
      *RemLength = cntRem;
    }
  else
    {
      (*FinalQuoLength)++;
      (*RemLength)++;
    }
  free (quotient_divisor);
  free (quotient);

  return NULL;
}
```

CUBES function.

```
int CUBES ( int cubFunLen, int size, char **cubFun, char *supV, int pos, char ***cubListSUP, char **coKernel )
  {
    int i, j, k, cntCube = 0;
    char intersection [size+1], supercube [size+1];

    cubListSUP [pos] = (char**) malloc ( sizeof (char*) );
    j = ( (supV[0] - 'a') * 2);
    for ( i = 0; i < cubFunLen; i++ )
      {
        if ( supV[1] == '\'' )
          {
            // converting the variable to position in which it has to be matched. //
            if ( cubFun [i][j] == '1' && cubFun [i][j+1] == '0' )
              {
                cubListSUP [pos] = (char**) realloc ( cubListSUP [pos], (cntCube+1) * sizeof (char*) );
                cubListSUP [pos][cntCube] = (char*) malloc ( (size+1) * sizeof (char) );
                initial ( size, cubListSUP [pos][cntCube], '\0' );
                strcpy ( cubListSUP [pos][cntCube], cubFun [i] );
                ++cntCube;
              }
          }
        else
          {
            if ( cubFun [i][j] == '0' && cubFun [i][j+1] == '1' )
              {
                cubListSUP [pos] = (char**) realloc ( cubListSUP [pos], (cntCube+1) * sizeof (char*) );
                cubListSUP [pos][cntCube] = (char*) malloc ( (size+1) * sizeof (char) );
                initial ( size, cubListSUP [pos][cntCube], '\0' );
                strcpy ( cubListSUP [pos][cntCube], cubFun [i] );
                ++cntCube;
              }
          }
      }
```

```
initial ( size, intersection, '1' );
for ( k = 0; k < cntCube; k++ )
  {
    for ( i = 0; i < size; i=i+2 )
      {
        if ( cubListSUP [pos][k][i] == '1' && cubListSUP [pos][k]
          {
            intersection [i] = '0';
            intersection [i+1] = '0';
          }
      }
    AND ( size, intersection, cubListSUP [pos][k], intersection )
  }
coKernel [pos] = (char*) malloc ( (size+1) *sizeof (char) );
initial ( size, coKernel [pos], '\0' );
strcpy ( coKernel [pos], intersection );
for ( j = 0; j < size; j=j+2)
  {
    if ( coKernel [pos][j] == '0' && coKernel [pos][j+1] == '0' )
      {
        coKernel [pos][j] = '1';
        coKernel [pos][j+1] = '1';
      }
  }
return cntCube;
```

Do_alg_division.

```
if ( objc != 4 )
  {
    fprintf (stderr, "Wrong arguments' number!\n");
    return NULL;
  }
int CUBE_LENGTH = atoi ( Tcl_GetString ( objv[1] ) ) * 2;
char dividend [LINE_MAX], divisor [LINE_MAX];
char coArg [CUBE_LENGTH+1];
int lengthDividend = 0, lengthDivisor = 0, i = 0, j = 0;
int cubesDividendLength, cubesDivisorLength, sizeCube, validity;
char *start, *end;
char **cubesDividend, **cubesDivisor;

lengthDividend = strlen ( Tcl_GetString ( objv[2] ) );
lengthDivisor = strlen ( Tcl_GetString ( objv[3] ) );
strncpy ( dividend, Tcl_GetString ( objv[2] ), lengthDividend );
strncpy ( &dividend [lengthDividend], "\0", 1 );
strncpy ( divisor, Tcl_GetString ( objv[3] ), lengthDivisor );
strncpy ( &divisor [lengthDivisor], "\0", 1 );

// finds each dividend's cube and stores them into 2D-array cubesDividend. //
cubesDividend = (char**) malloc ( sizeof (char*) );
start = dividend;
cubesDividendLength = 0;
```

```
for ( end = dividend; end-dividend < lengthDividend; end++ )
  {
    if ( isblank (*end) != 0 )
      {
        cubesDividend [i] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
        strncpy ( cubesDividend [i], start, CUBE_LENGTH );
        strncpy ( &cubesDividend [i][CUBE_LENGTH], "\0", 1 );
        validity = checkIfValid ( cubesDividend [i], CUBE_LENGTH );
        if ( validity > 0)
          {
            printf ( "Dividend's cubes %s is not valid\n", cubesDividend [i] );
            return NULL;
          }
        sizeCube = strlen ( cubesDividend [i] );  // cube's length //
        if ( sizeCube%2 != 0)
          {
            fprintf( stderr, "Cubes must have "RED"even"WHITE" size\n" );
            return NULL;
          }
        start = end+1;
        ++cubesDividendLength;
        cubesDividend = (char**) realloc ( cubesDividend, (cubesDividendLength+1) * sizeof (char*) );
        ++i;
      }
  }
cubesDividend [i] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
strncpy ( cubesDividend [i], start, CUBE_LENGTH );
strncpy ( &cubesDividend [i][CUBE_LENGTH], "\0", 1 );
validity = checkIfValid ( cubesDividend [i], CUBE_LENGTH );
if ( validity > 0)
  {
    printf ( "Dividend's cubes %s is not valid\n", cubesDividend [i] );
    return NULL;
  }
sizeCube = strlen ( cubesDividend [i] );  // cube's length //
if ( sizeCube%2 != 0)
  {
    fprintf( stderr, "Cubes must have "RED"even"WHITE" size\n" );
    return NULL;
  }
start = end+1;
++cubesDividendLength;
++i;
```

Do_alg_division.

```
// finds each divisor's cube and stores them into 2D-array cubesDivisor. //
cubesDivisor = (char**) malloc ( sizeof (char*) );
i = 0;
start = divisor;
cubesDivisorLength = 0;
for ( end = divisor; end-divisor < lengthDivisor; end++ )
  {
    if ( isblank (*end) != 0 )
      {
        cubesDivisor [i] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
        strncpy ( cubesDivisor [i], start, CUBE_LENGTH );
        strncpy ( &cubesDivisor [i][CUBE_LENGTH], "\0", 1 );
        validity = checkIfValid ( cubesDivisor [i], CUBE_LENGTH );
        if ( validity > 0)
          {
            printf ( RED"Divisor's cubes %s is not valid"WHITE"\n", cubesDivisor [i] );
            return NULL;
          }
        sizeCube = strlen ( cubesDivisor [i] );  // cube's length //
        if ( sizeCube%2 != 0)
          {
            fprintf( stderr, RED"Cubes must have even size"WHITE"\n" );
            return NULL;
          }
        start = end+1;
        ++cubesDivisorLength;
        cubesDivisor = (char**) realloc ( cubesDivisor, (cubesDivisorLength+1) * sizeof (char*) );
        ++i;
      }
  }
cubesDivisor [i] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
strncpy ( cubesDivisor [i], start, CUBE_LENGTH );
strncpy ( &cubesDivisor [i][CUBE_LENGTH], "\0", 1 );
validity = checkIfValid ( cubesDivisor [i], CUBE_LENGTH );
if ( validity > 0)
  {
    printf ( "Divisor's cubes %s is not valid\n", cubesDivisor [i] );
    return NULL;
  }
sizeCube = strlen ( cubesDivisor [i] );  // cube's length //
if ( sizeCube%2 != 0)
  {
    fprintf( stderr, "Cubes must have "RED"even"WHITE" size\n" );
    return NULL;
  }
start = end;
++cubesDivisorLength;
++i;
```

```
  char ***finalQuotient = (char***) malloc ( sizeof (char**) );
  char ***remainder = (char***) malloc ( sizeof (char**) );;
  int *FinalQuoLength = (int *) malloc ( sizeof (int) );
  int *RemLength = (int *) malloc ( sizeof (int) );

  *finalQuotient = (char**) malloc ( (cubesDividendLength*cubesDivisorLength) * sizeof(char*) );
  *remainder = (char**) malloc ( (cubesDividendLength*cubesDividendLength) * sizeof(char*) );

  algebraic_division ( CUBE_LENGTH, cubesDividendLength, cubesDividend, cubesDivisorLength, cubesDivisor,
                       FinalQuoLength, *finalQuotient, RemLength, *remainder );

  free (finalQuotient);
  free (remainder);
  free (cubesDividend);
  free (cubesDivisor);
  return NULL;
}
```

Find_SUPV.

```
int find_SUPV ( int size, char *arg1, int supportiveVarLength, char **supportiveVar )
  {
    int j, i;
    char ch = 'a';
    bool exist = false;
    for ( j = 0; j < size; j=j+2 )
      {
        for ( i = 0; i < supportiveVarLength; i++ )
          {
            if ( arg1 [j] == '0' && arg1 [j+1] == '1' )
              {
                if ( ( supportiveVar [i][0] == (j/2) + 'a' ) && (supportiveVar [i][1] != '\'') )
                  {
                    exist = true;
                  }
              }
            if ( arg1 [j] == '1' && arg1 [j+1] == '0' )
              {
                if ( ( supportiveVar [i][0] == (j/2) + 'a' ) && ('\'' == supportiveVar [i][1]) )
                  {
                    exist = true;
                  }
              }
          }
        if ( exist == true )
          {
            ++ch;
            exist = false;
            continue;
          }
        if ( arg1 [j] == '0' && arg1 [j+1] == '1' )
          {
            // allocation (1+1) is for character ch ( active value e.g. a ) and '\0'. //
            supportiveVar [supportiveVarLength] = (char*) malloc ( (1+1) * sizeof (char) );
            initial ( 1, supportiveVar [supportiveVarLength], '\0' );
            sprintf ( supportiveVar [supportiveVarLength], "%c", ch );
            ++supportiveVarLength;
          }
        if ( arg1 [j] == '1' && arg1 [j+1] == '0' )
          {
            // allocation (2+1) is for character ch ( non-active value e.g. a' ) and '\0'. //
            supportiveVar [supportiveVarLength] = (char*) malloc ( (2+1) * sizeof (char) );
            initial ( 2, supportiveVar [supportiveVarLength], '\0' );
            sprintf ( supportiveVar [supportiveVarLength], "%c'", ch );
            ++supportiveVarLength;
          }
        ++ch;
      }
    return supportiveVarLength;
  }
```

Redundant_kernels function.

```
void redundant_kernels ( int kernelsLevel, int CUBE_LENGTH, int cubesFunctionLength, char **cubesFunction, int *FinalQuoLength, char ***finalQuotient )
  {
    char **supportiveVar;
    int supportiveVarLength = 0, i, j;
    // find supportive variables of functions. //
    supportiveVar = (char**) malloc ( ( cubesFunctionLength * (CUBE_LENGTH/2) ) * sizeof (char*) );
    printf ("\nnumber of cubes: "GREEN"%d"WHITE"\nCubes: ", cubesFunctionLength );
    for ( i = 0; i < cubesFunctionLength; i++ )
      {
        printCube ( CUBE_LENGTH, cubesFunction [i] );
        if ( i < cubesFunctionLength - 1 )
          {
            printf(", ");
          }
        supportiveVarLength = find_SUPV ( CUBE_LENGTH, cubesFunction [i], supportiveVarLength, supportiveVar );
      }

    printf (" are going to be examined at level: "GREEN"%d"WHITE"\nSupportive Variables: ", kernelsLevel );
    for ( i = 0; i < supportiveVarLength; i++ )
      {
        printf ( GREEN"%s"WHITE" ", supportiveVar [i] );
      }
    printf ("\n");

    // variables' definition for list of supportive cubes and cokernels. //
    char ***cubesListSUP = (char***) malloc ( supportiveVarLength * sizeof (char**) );
    char **coKernel = (char**) malloc ( supportiveVarLength * sizeof (char*) );
    int cntCubesListSUP [supportiveVarLength];

    // variables' definition for algebraic division. //
    if ( kernelsLevel == 0 )
      {
        finalQuotient = (char***) realloc ( finalQuotient, supportiveVarLength * sizeof (char**) );
        FinalQuoLength = (int*) realloc ( FinalQuoLength, supportiveVarLength * sizeof (int) );
      }
    char ***remainder = (char***) malloc ( supportiveVarLength * sizeof (char**) );;
    int *RemLength = (int*) malloc ( supportiveVarLength * sizeof (int) );
```

Redundant_kernels function.

```
for ( i = 0; i < supportiveVarLength; i++ )
  {
    FinalQuoLength [i] = 0;
    RemLength [i] = 0;
    printf ( "Variable: "GREEN"%s"WHITE" at kernel's level "GREEN"%d"WHITE"\n", supportiveVar [i], kernelsLevel );
    cntCubesListSUP [i] = CUBES ( cubesFunctionLength, CUBE_LENGTH, cubesFunction, supportiveVar [i], i, cubesListSUP, coKernel );

    if ( cntCubesListSUP [i] >= 2 )
      {
        printf ( "  Cubes in supportive list are: %d\n", cntCubesListSUP [i] );
        printf ( "  Cubes that coKernel " );
        printCube ( CUBE_LENGTH, coKernel [i] );
        printf ( " is supported: {");
        for ( j = 0; j < cntCubesListSUP [i]; j++ )
          {
            // print variables from bits. //
            printCube ( CUBE_LENGTH, cubesListSUP [i][j] );
            if ( j < cntCubesListSUP [i] - 1 )
              {
                printf (", ");
              }
          }
        printf ("}\n    coK*K = ");
        printCube ( CUBE_LENGTH, coKernel [i] );
        printf ("*(");

        finalQuotient [i] = (char**) malloc ( cntCubesListSUP [i] *sizeof (char*) );
        remainder [i] = (char**) malloc ( cntCubesListSUP [i] *sizeof (char*) );
        for ( j = 0; j < cntCubesListSUP [i]; j++ )
          {
            algebraic_division ( CUBE_LENGTH, 1, &cubesListSUP [i][j], 1, &coKernel [i], &FinalQuoLength [i], &( finalQuotient [i][j] ), &RemLength [i], &( remainder [i][j] ) );
            if ( j < cntCubesListSUP [i] - 1 )
              {
                printf ("+");
              }
          }
        printf (")\n");
```

Redundant_kernels function.

```
            printf ("      redundant_kernels are called with cubes: " );
            for ( j = 0; j < FinalQuoLength [i]; j++ )
              {
                printCube ( CUBE_LENGTH, finalQuotient [i][j] );
                if ( j < FinalQuoLength [i] - 1 )
                  {
                    printf (", ");
                  }
              }
            printf ("\n");
            ++kernelsLevel;
            redundant_kernels ( kernelsLevel, CUBE_LENGTH, FinalQuoLength [i], finalQuotient [i], FinalQuoLength, &finalQuotient [i] );
            printf ( "End of level "GREEN"%d"WHITE"\n", kernelsLevel );
            --kernelsLevel;
          }
      }
}
```

do_r_kernels function.

```
void *do_r_kernels ( ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[] )
  {
    if ( objc != 3 )
      {
        fprintf (stderr, "Wrong arguments' number!\n");
        return NULL;
      }

    int CUBE_LENGTH = atoi ( Tcl_GetString ( objv[1] ) ) * 2;
    char function [LINE_MAX];
    int lengthFunction = 0, i = 0, j = 0;
    int cubesFunctionLength, sizeCube, validity, kernelsLevel = 0;
    char *start, *end;
    char **cubesFunction;

    lengthFunction = strlen ( Tcl_GetString ( objv[2] ) );
    strncpy ( function, Tcl_GetString ( objv[2] ), lengthFunction );
    strncpy ( &function [lengthFunction], "\0", 1 );

    cubesFunctionLength = lengthFunction / CUBE_LENGTH;
    cubesFunction = (char**) malloc ( sizeof (char*) );
    cubesFunctionLength = 0;
    start = function;
    printf ("F = ");
```

```
for ( end = function; end-function < lengthFunction; end++ )
  {
    if ( isblank (*end) != 0 )
      {
        cubesFunction [cubesFunctionLength] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
        strncpy ( cubesFunction [cubesFunctionLength], start, CUBE_LENGTH );
        strncpy ( &cubesFunction [cubesFunctionLength][CUBE_LENGTH], "\0", 1 );
        printCube ( CUBE_LENGTH, cubesFunction [cubesFunctionLength] );
        printf (" + ");
        validity = checkIfValid ( cubesFunction [cubesFunctionLength], CUBE_LENGTH );
        if ( validity > 0)
          {
            printf ( "Function's cubes %s is not valid\n", cubesFunction [cubesFunctionLength] );
            return NULL;
          }
        sizeCube = strlen ( cubesFunction [cubesFunctionLength] );  // cube's length //
        if ( sizeCube%2 != 0)
          {
            fprintf( stderr, "Cubes must have "RED"even"WHITE" size\n" );
            return NULL;
          }
        start = end+1;
        ++cubesFunctionLength;
        cubesFunction = (char**) realloc ( cubesFunction, (cubesFunctionLength+1) * sizeof (char*) );
        ++i;
      }
  }
```

do_r_kernels function.

```
  cubesFunction [cubesFunctionLength] = (char*) malloc ( (CUBE_LENGTH+1) * sizeof(char) );
  strncpy ( cubesFunction [cubesFunctionLength], start, CUBE_LENGTH );
  strncpy ( &cubesFunction [cubesFunctionLength][CUBE_LENGTH], "\0", 1 );
  printCube ( CUBE_LENGTH, cubesFunction [cubesFunctionLength] );
  printf ("\n");
  validity = checkIfValid ( cubesFunction [cubesFunctionLength], CUBE_LENGTH );
  if ( validity > 0)
    {
      printf ( "Function's cubes %s is not valid\n", cubesFunction [cubesFunctionLength] );
      return NULL;
    }
  sizeCube = strlen ( cubesFunction [cubesFunctionLength] );  // cube's length //
  if ( sizeCube%2 != 0)
    {
      fprintf( stderr, "Cubes must have "RED"even"WHITE" size\n" );
      return NULL;
    }
  start = end+1;
  ++cubesFunctionLength;

  char ***finalQuotient = (char***) malloc ( sizeof (char**) );
  int *FinalQuoLength = (int*) malloc ( sizeof (int) );

  redundant_kernels ( kernelsLevel, CUBE_LENGTH, cubesFunctionLength, cubesFunction, FinalQuoLength, finalQuotient );
  printf ( "End of level "GREEN"%d"WHITE"\n", kernelsLevel );

  free (cubesFunction);
}
```