Contents

1 Basic Test Results

```
Running...
Opening tar file
sudukutree.h
sudukusolver.c
makefile
genericdfs.c
sudukutree.c
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Importing files
OK
Compilation check...
Compiling...
gcc -std=c99 -Wextra -Wall -Wvla -c genericdfs.c
ar rcs genericdfs.a genericdfs.o
gcc -std=c99 -Wextra -Wall -Wvla  -g -lm genericdfs.c sudukutree.c sudukusolver.c -o sudukusolver
OK
Running test...
./sudukusolver /cs/course/2014/labc/public/ex3/basic.in |diff -B /cs/course/2014/labc/public/ex3/basic.out -
OK
Compilation seems OK! Check if you got warnings!

========================
= Checking coding style =
========================
sudukutree.c(166, 6):  deep_blocks {Do not make too deep block(6) ({). It makes not readable code}
 ** Total Violated Rules      : 1
 ** Total Errors Occurs       : 1
 ** Total Violated Files Count: 1
```

2 genericdfs.c

```c
/*
 * genericdfs.c
 *
 *  Created on: Dec 7, 2014
 *      Author: mutazmanaa
 */

#include <stdio.h>
#include <stdlib.h>
#include "genericdfs.h"

/**
 * @brief getBest This function returns the node with the highest value in the
 * tree, using
 * DFS algorithm.
 * @param head The head of the tree.
 * @param getChildren A function that gets a node and a pointer to an array of
 *  nodes.
 * The function allocates memory for an array of all the children of the node,
 *  populate it,
 * and returns it using the second parameter. The returned value is the number
 *  of children.
 * @param getVal A function that gets a node and returns its value, as int.
 * @param freeNode A function that frees a node from memory. This function will
 *  be called for each node returned by getChildren.
 * @param copy A function that does a deep copy of a node.
 * @param best The highest possible value for a node. When the function
 * encounters a node with that
 * value, it stops looking and returns it. If the best value can't be determined
 * , pass UINT_MAX (defined in limits.h) for that parameter.
 * @return The node with the highest value in the tree. In case of an error, or
 * when all the nodes
 * in the tree have a value of zero, the returned node is NULL. If some nodes
 * share the best value,the function returns the first one it encounters.
 */
pNode getBest(pNode head, getNodeChildrenFunc getChildren,
              getNodeValFunc getVal, freeNodeFunc freeNode, copyNodeFunc copy,
              unsigned int best)
{

            pNode* childrens = NULL;
            pNode bestChoice = copy(head);
            if(getVal(bestChoice) == best)
            {
                free(childrens);
                return bestChoice;
            }

            int numberOfChildrens;
            numberOfChildrens = getChildren(bestChoice, &childrens);
            int i;
            for(i = 0; i < numberOfChildrens; i++)
            {
                if(getVal(bestChoice) == best)
                {
                      break;

                }
                 pNode maxChoice = getBest(childrens[i], getChildren, getVal,
```

-1/-5 General bad style remark (code='general_bad_style') Missing spaces after keywords

0/-20 (code='general_comment')) you are not looking for the highest value as instructed in the exercise, but finds a node whose value equals to "best"

```
                        freeNode, copy, best);

                if(getVal(maxChoice) > getVal(bestChoice))
                {
                    freeNode(bestChoice);
                    bestChoice = copy(maxChoice);
                }


                freeNode(maxChoice);
            }

            for(i = 0 ; i < numberOfChildrens; i++)
            {
                freeNode(childrens[i]);
            }


            free(childrens);

            return bestChoice;


}
```

3 makefile

```makefile
CC = gcc
CFlag = -std=c99 -Wextra -Wall -Wvla
OBJECTS = genericdfs.c sudukutree.c sudukusolver.c
CleanOBJECTS = sudukusolver

.PHONY: all clean genericDFS

all: sudukuSolver genericdfs.a

sudukuSolver: $(OBJECTS) genericdfs.a
    $(CC) $(CFlag)  -g -lm $(OBJECTS) -o sudukusolver

genericdfs.a: genericDFS
    ar rcs genericdfs.a genericdfs.o

genericDFS: genericdfs.c
    $(CC) $(CFlag) -c genericdfs.c

tar:
    tar -cvf ex3.tar makefile sudukusolver.c sudukutree.c sudukutree.h\
                              genericdfs.c


clean:
    rm -rf $(CleanOBJECTS) *.o *.a
```

4 sudukusolver.c

```c
/*
 * sudukusolver.c
 *
 *  Created on: Dec 7, 2014
 *      Author: mutazmanaa
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sudukutree.h"
#define MIN_SIZE 1
#define MAX_SIZE 100



/**
 * function that dealing with opening file with match error message
 * @param argc number of parameters in shell call to the function
 * @param argv paramaters in  shell call to the function
 * @return fp the file opened
 */

FILE* openFile(int argc, char* argv[])
{

    if(argc != 2)
    {
        printf("please supply a file!\n");
        printf("usage: sudukusolver <filename>");
        exit(1);
    }
    FILE* fp = fopen(argv[1], "r");
    if(fp == NULL)
    {
        printf("%s: no such file!\n", argv[1]);
        exit(1);

    }

    return fp;


}


/**
 *function get the suduku information from the file
 *@param file a file in suduku format
 *@param node the node of the suduku
 *@param argv use to get the file name
 *returnnode of the suduku
 */
Node* dataRead(FILE* file, char** argv)
{

    int i, j;
    int rowCounter = 0;
    Node* node = (Node*) malloc(sizeof(Node));
```

```
    if(fscanf(file, "%d", &node->size) != 1 || (node->size < MIN_SIZE || node->size
       > MAX_SIZE) || (int)(sqrt(node->size) * sqrt(node->size)) != node->size)
    {
        printf("%s: not a valid suduku file!\n", argv[1]);
        exit(0);

    }

    node->matrix = (int**)malloc(node->size * sizeof(int*));
    int index;
    for (index = 0; index < node->size ; ++index)
    {

        node->matrix[index] = (int*)malloc(node->size * sizeof(in

    }

    for(i = 0; i < node->size; i++)
    {
        int colomCounter = 0;
        for(j = 0; j < node->size; j++)
        {

            if(fscanf(file, "%d", &node->matrix[i][j]) != 1 ||
               node->matrix[i][j]
               < (MIN_SIZE - 1)  || node->matrix[i][j] > node->size)
            {

                printf("%s: not a valid suduku file!\n", argv[1]);
                exit(0);
            }
            colomCounter++;

        }

        if(colomCounter != node->size)
        {

            printf("%s: not a valid suduku file!\n", argv[1]);
            exit(0);
        }

        rowCounter++;
    }


    if(rowCounter != node->size)
    {

        printf("%s: not a valid suduku file!\n", argv[1]);
        exit(0);
    }
    node->value = 0;
    for(i = 0; i < node->size; i++)
    {
        for(j = 0; j < node->size; j++)
        {
            if(node->matrix[i][j] != 0)
            {
                node->value++;
            }
        }
    }
    fclose(file);
    return node;


}
```

-2/-2 Your code accesses pointers without verifying first that the value of the pointer is not null. (code='missing_check_if_null')

0/0 Missing documentation (code='missing_documentation')

```c


void printTable(int** matrix, int size)
{
    int i, j;
    printf("%d\n", size);
    for(i = 0; i < size; i++)
    {
        for(j = 0; j < size - 1; j++)
        {
            printf("%d ", matrix[i][j]);
        }
        printf("%d", matrix[i][j]);
        printf("\n");
    }

}



int main(int argc, char* argv[])
{

    FILE* file = openFile(argc, argv);
    Node* sudukuStruct = NULL;
    sudukuStruct = dataRead(file, argv);
    Node* best = NULL;
    unsigned int bestBoard =
            (unsigned int)sudukuStruct->size * sudukuStruct->size;

    if(!(validSuduku(sudukuStruct)))
    {
        printf("%s: not a valid suduku file!\n", argv[1]);
        exit(0);
    }

    if(getValue(sudukuStruct) == bestBoard)
    {
        printTable(sudukuStruct->matrix, sudukuStruct->size);
        freeNodeP(best);
        freeNodeP(sudukuStruct);
        return 0;
    }

    best = getBest(sudukuStruct, getChildrenCount, getValue, freeNodeP,
            copyNode, bestBoard);
    if(getValue(best) == bestBoard)
    {
        printTable(best->matrix, best->size);
        freeNodeP(best);
        freeNodeP(sudukuStruct);

    }
    else
    {
        freeNodeP(sudukuStruct);
        freeNodeP(best);
        printf("no soluation!\n");
    }


    return 0;
}
```

5 sudukutree.h

```c
/*
 * sudukutree.h
 *
 *  Created on: Dec 7, 2014
 *      Author: mutazmanaa
 */

#ifndef SUDUKUTREE_H_
#define SUDUKUTREE_H_
#include <stdio.h>
#include <stdlib.h>
#include "genericdfs.h"
/********      types and functions types*/

/**
 * Node structur that present a situataion of suduku
 *@param value number of non-zero numbers
 *@param matrix the suduku board
 *@param size the size of the suduku board
 *@param next a pointer to a Node
 */
typedef struct Node
{
    int size;
    int value;
    int** matrix;

}Node;


/**
 * Function check if sudukuBoard is valid
 * @param nod a sudukuboard for validity check
 * @return 1 if valid and zero if not valid
 */

int validSuduku(Node* node);

/**
 * function that get the number of valid values belong to a zero value in suduku
 * board and also update an array of nodes contaaines the children.
 * @Param nodeptr a pointer to Node of suduku
 * @Param childrenArr an array of suduku nodes contains the children
 * (valid values)
 * @Return number of childrens
 */
int getChildrenCount(pNode, pNode** /*for the result*/pNode);

/**
 * function that count the non-zero values in a suduku board saved in the node
 * @Param node a node (contain a suduku board inside)
 * @Return the number of the non-zero values in suduku board
 */
unsigned int getValue(pNode);

/**
 * function get a Node and free all allocations iside and then free the Node
 * @Param node a node to free
 *
```

```c
 */
void freeNodeP(pNode);


/**
 * this function get a Node and copy it
 * @Param node a node to copy
 * @Return  a copied node
 */
pNode copyNode(pNode);

#endif /* SUDUKUTREE_H_ */
```

6 sudukutree.c

```c
/*
 * sudukutree.c
 *
 *  Created on: Dec 7, 2014
 *      Author: mutazmanaa
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "sudukutree.h"
//gcc -Wall -Wextra -lm genericdfs.c sudukutree.c sudukusolver.c -o sudukusolver
//(./sudukusolver 1.in)
//valgrind --leak-check=full --show-possibly-lost=yes --show-reachable=yes
//sudukusolver 1.in
/**
 * function check if value found in a row in matrix
 * @Param matrix the suduku board
 * @Param row the row we search in
 * @Param value the value we search about
 * @Param size the size of the matrix
 * @Return 0 if value founded, else return 1
 *
 */
int checkRow(int** matrix, int row, int col, int value, int size)
{
    int c;
    if(value != 0)
    {
        for(c = 0; c < size; c++)
        {
            if(matrix[row][c] == value && c != col)
            {
                return 0;
            }
        }
    }

    return 1;

}


/**
 * function check if value found in a column in matrix
 * @param matrix the suduku board
 * @Param col the culumn we search in
 * @Param value the value we search about
 * @Param size the size of the matrix
 * @Return 0 if value founded, else return 1
 */
int checkCol(int** matrix, int row, int col, int value, int size)
{
    int r;
    if(value != 0)
    {
        for(r = 0; r < size; r++)
        {
```

```c

            if(matrix[r][col] == value && r != row)
            {
                return 0;
            }
        }
    }

    return 1;
}


/**
 * function check if value found in a small square  in matrix
 * @param matrix the suduku board
 * @Param row the row index of the value
 * @param col the colum index of the value
 * @Param value the value we search about
 * @Param size the size of the matrix
 * @Return 0 if value founded, else return 1
 */

int checkSquare(int** matrix, int row, int col, int value, int size)
{

    int rowS = (row / (int)sqrt(size)) * sqrt(size);
    int colS = (col / (int)sqrt(size)) * sqrt(size);
    int r, c;
    if(value != 0)
    {
        for(r = 0; r < sqrt(size); r++)
        {
            for(c = 0; c < sqrt(size); c++)
            {

                if(matrix[rowS + r][colS + c] == value && (rowS + r) != row &&
                   (colS + c) != col)
                {
                    return 0;
                }


            }
        }
    }
    return 1;
}

/**
 * Function check if sudukuBoard is valid
 * @param nod a sudukuboard for validity check
 * @return 1 if valid and zero if not valid
 */

int validSuduku(Node* node)
{
    int i, j;
    for(i = 0; i < node->size; i++)
    {
        for(j = 0; j < node->size; j++)
        {
            if(checkRow(node->matrix, i , j, node->matrix[i][j], node->size) &&
               checkCol(node->matrix, i, j, node->matrix[i][j], node->size)  &&
               checkSquare(node->matrix, i, j, node->matrix[i][j], node->size))
            {
                 continue;
            }
            else
```

```
            {
                return 0;
            }
        }
    }
    return 1;
}

/**
 * function that get the number of valid values belong to a zero value in suduku
 * board and also update an array of nodes contaaines the children.
 * @Param nodeptr a pointer to Node of suduku
 * @Param childrenArr an array of suduku nodes contains the children
 * (valid values)
 * @Return number of childrens
 */
int getChildrenCount(pNode nodeptr, pNode** childrenArr /*for the result*/)
{
    int i, j, m, rowPos = 0, colPos = 0;
    int s = 0;
    int childrenCount = 0;
    Node* node = (Node*) nodeptr;
    Node*** childrenArray = (Node***) childrenArr;

    int* validChildren = (int*) malloc ((node->size + 1) * sizeof(int));
    for(i = 0 ; i < node->size; i++ )
    {
        for(j = 0; j < node->size; j++)
        {
            if(node->matrix[i][j] == 0)
            {
                rowPos = i;
                colPos = j;
                for(m = 1; m < node->size + 1; m++)
                {
                    if(checkRow(node->matrix, i, j, m, node->size) &&
                       checkCol(node->matrix, i, j, m, node->size) &&
                       checkSquare(node->matrix, i, j, m, node->size))
                    {
                        validChildren[s++] = m;
                        childrenCount++;

                    }

                }
                if(!(childrenCount))
                {
                    free(validChildren);
                    return 0;
                }

                i = node->size;
                break;
            }

        }

    }



    (*childrenArray) = (Node**) malloc (childrenCount * sizeof(Node*));
    int d;
    for(d = 0; d < childrenCount; d++)
    {
        (*childrenArray)[d] = copyNode(node);
        (*childrenArray)[d]->value++;
        (*childrenArray)[d]->matrix[rowPos][colPos] = validChildren[d];
```

-2/-2 Your code accesses pointers without verifying first that the value of the pointer is not null. (code='missing_check_if_null')

```c
    }


    free(validChildren);
    return childrenCount;

}

/**
 * function that count the non-zero values in a suduku board saved in the node
 * @Param node a node (contain a suduku board inside)
 * @Return the number of the non-zero values in suduku board
 */
unsigned int getValue(pNode node)
{
    Node* valNode = (Node*) node;
    return (unsigned int)(valNode->value);
}

/**
 * function get a Node and free all allocations iside and then free the Node
 * @Param node a node to free
 *
 */
void freeNodeP(pNode node)
{
    Node* frNode = (Node*)node;
    int i;
    for(i = 0; i < frNode->size; i++)
    {
        free(frNode-> matrix[i]);

    }
    free(frNode->matrix);
    free(frNode);
}

/**
 * this function get a Node and copy it
 * @Param node a node to copy
 * @Return  a copied node
 */
pNode copyNode(pNode node)
{
    Node* copyNode = (Node*) malloc(sizeof(Node));
    Node* orgNode = (Node*) node;
    int i, j;
    copyNode->matrix = (int**)malloc(orgNode->size * sizeof(int*));
    int index;
    for (index = 0; index < orgNode->size; ++index)
    {
        copyNode->matrix[index] = (int*)malloc((orgNode->size) * si

    }
    copyNode->size = orgNode->size;
    copyNode->value = orgNode->value;

    for(i = 0; i < orgNode->size; i++)
    {
        for(j = 0; j < orgNode->size; j++)
        {
            copyNode->matrix[i][j] = orgNode->matrix[i][j];
        }
    }


    return copyNode;
}
```

-2/-2 Your code accesses pointers without verifying first that the value of the pointer is not null. (code='missing_check_if_null')