

Contents

1	Basic Test Results	2
2	genericdfs.c	3
3	makefile	5
4	sudokusolver.c	6
5	sudukutree.h	9
6	sudukutree.c	11

1 Basic Test Results

```
1 Running...
2 Opening tar file
3 sudukutree.h
4 sudokusolver.c
5 makefile
6 genericdfs.c
7 sudukutree.c
8 OK
9 Tar extracted O.K.
10 Checking files...
11 OK
12 Making sure files are not empty...
13 OK
14 Importing files
15 OK
16 Compilation check...
17 Compiling...
18 gcc -std=c99 -Wextra -Wall -Wvla -c genericdfs.c
19 ar rcs genericdfs.a genericdfs.o
20 gcc -std=c99 -Wextra -Wall -Wvla -g -lm genericdfs.c sudukutree.c sudokusolver.c -o sudokusolver
21 OK
22 Running test...
23 ./sudokusolver /cs/course/2014/lab/public/ex3/basic.in |diff -B /cs/course/2014/lab/public/ex3/basic.out -
24 OK
25 Compilation seems OK! Check if you got warnings!
26
27 =====
28 = Checking coding style =
29 =====
30 sudukutree.c(166, 6): deep_blocks {Do not make too deep block(6) ({}). It makes not readable code}
31 ** Total Violated Rules      : 1
32 ** Total Errors Occurs      : 1
33 ** Total Violated Files Count: 1
```

2 genericdfs.c

```
1  /*
2  * genericdfs.c
3  *
4  * Created on: Dec 7, 2014
5  * Author: mutazmanaa
6  */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include "genericdfs.h"
11
12 /**
13  * @brief getBest This function returns the node with the highest value in the
14  * tree, using
15  * DFS algorithm.
16  * @param head The head of the tree.
17  * @param getChildren A function that gets a node and a pointer to an array of
18  * nodes.
19  * The function allocates memory for an array of all the children of the node,
20  * populate it,
21  * and returns it using the second parameter. The returned value is the number
22  * of children.
23  * @param getVal A function that gets a node and returns its value, as int.
24  * @param freeNode A function that frees a node from memory. This function will
25  * be called for each node returned by getChildren.
26  * @param copy A function that does a deep copy of a node.
27  * @param best The highest possible value for a node. When the function
28  * encounters a node with that
29  * value, it stops looking and returns it. If the best value can't be determined
30  * , pass UINT_MAX (defined in limits.h) for that parameter.
31  * @return The node with the highest value in the tree. In case of an error, or
32  * when all the nodes
33  * in the tree have a value of zero, the returned node is NULL. If some nodes
34  * share the best value, the function returns the first one it encounters.
35  */
36 pNode getBest(pNode head, getNodeChildrenFunc getChildren,
37               getNodeValFunc getVal, freeNodeFunc freeNode, copyNodeFunc copy,
38               unsigned int best)
39 {
40
41     pNode* childrens = NULL;
42     pNode bestChoice = copy(head);
43     if(getVal(bestChoice) == best)
44     {
45         free(childrens);
46         return bestChoice;
47     }
48
49     int numberOfChildrens;
50     numberOfChildrens = getChildren(bestChoice, &childrens);
51     int i;
52     for(i = 0; i < numberOfChildrens; i++)
53     {
54         if(getVal(bestChoice) == best)
55         {
56             break;
57         }
58     }
59     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"

```

60         freeNode, copy, best);
61
62         if(getVal(maxChoice) > getVal(bestChoice))
63         {
64             freeNode(bestChoice);
65             bestChoice = copy(maxChoice);
66         }
67
68         freeNode(maxChoice);
69     }
70
71     for(i = 0 ; i < numberOfChildrens; i++)
72     {
73         freeNode(childrens[i]);
74     }
75
76
77     free(childrens);
78
79     return bestChoice;
80
81
82
83 }
```

3 makefile

```
1 CC = gcc
2 CFlag = -std=c99 -Wextra -Wall -Wvla
3 OBJECTS = genericdfs.c sudukutree.c sudokusolver.c
4 CleanOBJECTS = sudokusolver
5
6 .PHONY: all clean genericDFS
7
8 all: sudukuSolver genericdfs.a
9
10 sudukuSolver: $(OBJECTS) genericdfs.a
11     $(CC) $(CFlag) -g -lm $(OBJECTS) -o sudokusolver
12
13 genericdfs.a: genericDFS
14     ar rcs genericdfs.a genericdfs.o
15
16 genericDFS: genericdfs.c
17     $(CC) $(CFlag) -c genericdfs.c
18
19 tar:
20     tar -cvf ex3.tar makefile sudokusolver.c sudukutree.c sudukutree.h\
21         genericdfs.c
22
23
24 clean:
25     rm -rf $(CleanOBJECTS) *.o *.a
```

4 sudokusolver.c

```
1  /*
2   * sudokusolver.c
3   *
4   * Created on: Dec 7, 2014
5   * Author: mutazmanaa
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <math.h>
11 #include "sudukutree.h"
12 #define MIN_SIZE 1
13 #define MAX_SIZE 100
14
15
16
17 /**
18  * function that dealing with opening file with match error message
19  * @param argc number of parameters in shell call to the function
20  * @param argv paramaters in shell call to the function
21  * @return fp the file opened
22  */
23
24 FILE* openFile(int argc, char* argv[])
25 {
26
27     if(argc != 2)
28     {
29         printf("please supply a file!\n");
30         printf("usage: sudokusolver <filename>");
31         exit(1);
32     }
33     FILE* fp = fopen(argv[1], "r");
34     if(fp == NULL)
35     {
36         printf("%s: no such file!\n", argv[1]);
37         exit(1);
38     }
39 }
40
41 return fp;
42
43 }
44
45
46
47 /**
48  *function get the sudoku information from the file
49  *@param file a file in sudoku format
50  *@param node the node of the sudoku
51  *@param argv use to get the file name
52  *return node of the sudoku
53  */
54 Node* dataRead(FILE* file, char** argv)
55 {
56
57     int i, j;
58     int rowCounter = 0;
59     Node* node = (Node*) malloc(sizeof(Node));
```

```

60     if(fscanf(file, "%d", &node->size) != 1 || (node->size < MIN_SIZE || node->size
61         > MAX_SIZE) || (int)(sqrt(node->size) * sqrt(node->size)) != node->size)
62     {
63         printf("%s: not a valid sudoku file!\n", argv[1]);
64         exit(0);
65     }
66 }
67
68 node->matrix = (int**)malloc(node->size * sizeof(int*));
69 int index;
70 for (index = 0; index < node->size ; ++index)
71 {
72     node->matrix[index] = (int*)malloc(node->size * sizeof(int));
73
74 }
75
76 for(i = 0; i < node->size; i++)
77 {
78     int colomCounter = 0;
79     for(j = 0; j < node->size; j++)
80     {
81
82         if(fscanf(file, "%d", &node->matrix[i][j]) != 1 ||
83             node->matrix[i][j]
84             < (MIN_SIZE - 1) || node->matrix[i][j] > node->size)
85         {
86
87             printf("%s: not a valid sudoku file!\n", argv[1]);
88             exit(0);
89         }
90         colomCounter++;
91     }
92
93     if(colomCounter != node->size)
94     {
95         printf("%s: not a valid sudoku file!\n", argv[1]);
96         exit(0);
97     }
98     rowCounter++;
99 }
100
101 if(rowCounter != node->size)
102 {
103     printf("%s: not a valid sudoku file!\n", argv[1]);
104     exit(0);
105 }
106 node->value = 0;
107 for(i = 0; i < node->size; i++)
108 {
109     for(j = 0; j < node->size; j++)
110     {
111         if(node->matrix[i][j] != 0)
112         {
113             node->value++;
114         }
115     }
116 }
117 fclose(file);
118 return node;
119 }
120
121 }
122
123 }

```

-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_documenta
tion')

```
128
129
130 void printTable(int** matrix, int size)
131 {
132     int i, j;
133     printf("%d\n", size);
134     for(i = 0; i < size; i++)
135     {
136         for(j = 0; j < size - 1; j++)
137         {
138             printf("%d ", matrix[i][j]);
139         }
140         printf("%d", matrix[i][j]);
141         printf("\n");
142     }
143 }
144 }
145
146
147
148 int main(int argc, char* argv[])
149 {
150
151     FILE* file = openFile(argc, argv);
152     Node* sudokuStruct = NULL;
153     sudokuStruct = dataRead(file, argv);
154     Node* best = NULL;
155     unsigned int bestBoard =
156         (unsigned int)sudokuStruct->size * sudokuStruct->size;
157
158     if(!(validSudoku(sudokuStruct)))
159     {
160         printf("%s: not a valid sudoku file!\n", argv[1]);
161         exit(0);
162     }
163
164     if(getValue(sudokuStruct) == bestBoard)
165     {
166         printTable(sudokuStruct->matrix, sudokuStruct->size);
167         freeNodeP(best);
168         freeNodeP(sudokuStruct);
169         return 0;
170     }
171
172     best = getBest(sudokuStruct, getChildrenCount, getValue, freeNodeP,
173         copyNode, bestBoard);
174     if(getValue(best) == bestBoard)
175     {
176         printTable(best->matrix, best->size);
177         freeNodeP(best);
178         freeNodeP(sudokuStruct);
179     }
180
181     else
182     {
183         freeNodeP(sudokuStruct);
184         freeNodeP(best);
185         printf("no solution!\n");
186     }
187
188
189     return 0;
190 }
```


5 sudukutree.h

```
1  /*
2   * sudukutree.h
3   *
4   * Created on: Dec 7, 2014
5   * Author: mutazmanaa
6   */
7
8  #ifndef SUDUKUTREE_H_
9  #define SUDUKUTREE_H_
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include "genericdfs.h"
13 /*****      types and functions types*/
14
15 /**
16  * Node structur that present a situataion of suduku
17  * @param value number of non-zero numbers
18  * @param matrix the suduku board
19  * @param size the size of the suduku board
20  * @param next a pointer to a Node
21  */
22 typedef struct Node
23 {
24     int size;
25     int value;
26     int** matrix;
27 }Node;
28
29
30
31 /**
32  * Function check if sudukuBoard is valid
33  * @param nod a sudukuboard for validity check
34  * @return 1 if valid and zero if not valid
35  */
36
37 int validSudoku(Node* node);
38
39 /**
40  * function that get the number of valid values belong to a zero value in suduku
41  * board and also update an array of nodes contaaines the children.
42  * @Param nodeptr a pointer to Node of suduku
43  * @Param childrenArr an array of suduku nodes contains the children
44  * (valid values)
45  * @Return number of childrens
46  */
47 int getChildrenCount(pNode, pNode** /*for the result*/pNode);
48
49 /**
50  * function that count the non-zero values in a suduku board saved in the node
51  * @Param node a node (contain a suduku board inside)
52  * @Return the number of the non-zero values in suduku board
53  */
54 unsigned int getValue(pNode);
55
56 /**
57  * function get a Node and free all allocations iside and then free the Node
58  * @Param node a node to free
59  */
```

```

60  */
61  void freeNodeP(pNode);
62
63
64  /**
65   * this function get a Node and copy it
66   * @Param node a node to copy
67   * @Return a copied node
68   */
69  pNode copyNode(pNode);
70
71  #endif /* SUDUKUTREE_H_ */

```

6 sudukutree.c

```
1  /*
2   * sudukutree.c
3   *
4   * Created on: Dec 7, 2014
5   * Author: mutazmanaa
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <math.h>
11
12 #include "sudukutree.h"
13 //gcc -Wall -Wextra -lm genericdfs.c sudukutree.c sudokusolver.c -o sudokusolver
14 //(. /sudokusolver 1.in)
15 //valgrind --leak-check=full --show-possibly-lost=yes --show-reachable=yes
16 //sudokusolver 1.in
17 /**
18  * function check if value found in a row in matrix
19  * @Param matrix the sudoku board
20  * @Param row the row we search in
21  * @Param value the value we search about
22  * @Param size the size of the matrix
23  * @Return 0 if value founded, else return 1
24  *
25  */
26 int checkRow(int** matrix, int row, int col, int value, int size)
27 {
28     int c;
29     if(value != 0)
30     {
31         for(c = 0; c < size; c++)
32         {
33             if(matrix[row][c] == value && c != col)
34             {
35                 return 0;
36             }
37         }
38     }
39     return 1;
40 }
41
42 }
43
44
45 /**
46  * function check if value found in a column in matrix
47  * @Param matrix the sudoku board
48  * @Param col the column we search in
49  * @Param value the value we search about
50  * @Param size the size of the matrix
51  * @Return 0 if value founded, else return 1
52  */
53 int checkCol(int** matrix, int row, int col, int value, int size)
54 {
55     int r;
56     if(value != 0)
57     {
58         for(r = 0; r < size; r++)
59         {
```

```

60
61         if(matrix[r][col] == value && r != row)
62         {
63             return 0;
64         }
65     }
66 }
67
68     return 1;
69 }
70
71
72 /**
73  * function check if value found in a small square in matrix
74  * @param matrix the suduku board
75  * @Param row the row index of the value
76  * @Param col the colum index of the value
77  * @Param value the value we search about
78  * @Param size the size of the matrix
79  * @Return 0 if value founded, else return 1
80  */
81
82 int checkSquare(int** matrix, int row, int col, int value, int size)
83 {
84
85     int rowS = (row / (int)sqrt(size)) * sqrt(size);
86     int colS = (col / (int)sqrt(size)) * sqrt(size);
87     int r, c;
88     if(value != 0)
89     {
90         for(r = 0; r < sqrt(size); r++)
91         {
92             for(c = 0; c < sqrt(size); c++)
93             {
94
95                 if(matrix[rowS + r][colS + c] == value && (rowS + r) != row &&
96                    (colS + c) != col)
97                 {
98                     return 0;
99                 }
100
101             }
102         }
103     }
104 }
105     return 1;
106 }
107
108 /**
109  * Function check if sudukuBoard is valid
110  * @param nod a sudukuboard for validity check
111  * @return 1 if valid and zero if not valid
112  */
113
114 int validSudoku(Node* node)
115 {
116     int i, j;
117     for(i = 0; i < node->size; i++)
118     {
119         for(j = 0; j < node->size; j++)
120         {
121             if(checkRow(node->matrix, i, j, node->matrix[i][j], node->size) &&
122                checkCol(node->matrix, i, j, node->matrix[i][j], node->size) &&
123                checkSquare(node->matrix, i, j, node->matrix[i][j], node->size))
124             {
125                 continue;
126             }
127             else

```

```

128         {
129             return 0;
130         }
131     }
132 }
133 return 1;
134 }
135
136 /**
137  * function that get the number of valid values belong to a zero value in sudoku
138  * board and also update an array of nodes contains the children.
139  * @Param nodeptr a pointer to Node of sudoku
140  * @Param childrenArr an array of sudoku nodes contains the children
141  * (valid values)
142  * @Return number of childrens
143  */
144 int getChildrenCount(pNode nodeptr, pNode** childrenArr /*for the result*/)
145 {
146     int i, j, m, rowPos = 0, colPos = 0;
147     int s = 0;
148     int childrenCount = 0;
149     Node* node = (Node*) nodeptr;
150     Node*** childrenArray = (Node***) childrenArr;
151
152     int* validChildren = (int*) malloc ((node->size + 1) * sizeof(int));
153     for(i = 0 ; i < node->size; i++)
154     {
155         for(j = 0; j < node->size; j++)
156         {
157             if(node->matrix[i][j] == 0)
158             {
159                 rowPos = i;
160                 colPos = j;
161                 for(m = 1; m < node->size + 1; m++)
162                 {
163                     if(checkRow(node->matrix, i, j, m, node->size) &&
164                        checkCol(node->matrix, i, j, m, node->size) &&
165                        checkSquare(node->matrix, i, j, m, node->size))
166                     {
167                         validChildren[s++] = m;
168                         childrenCount++;
169                     }
170                 }
171             }
172             if(!(childrenCount))
173             {
174                 free(validChildren);
175                 return 0;
176             }
177
178             i = node->size;
179             break;
180         }
181     }
182 }
183
184 }
185
186
187
188
189 (*childrenArray) = (Node**) malloc (childrenCount * sizeof(Node*));
190 int d;
191 for(d = 0; d < childrenCount; d++)
192 {
193     (*childrenArray)[d] = copyNode(node);
194     (*childrenArray)[d]->value++;
195     (*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')

```

196     }
197
198
199     free(validChildren);
200     return childrenCount;
201 }
202
203
204 /**
205  * function that count the non-zero values in a sudoku board saved in the node
206  * @Param node a node (contain a sudoku board inside)
207  * @Return the number of the non-zero values in sudoku board
208  */
209 unsigned int getValue(pNode node)
210 {
211     Node* valNode = (Node*) node;
212     return (unsigned int)(valNode->value);
213 }
214
215 /**
216  * function get a Node and free all allocations inside and then free the Node
217  * @Param node a node to free
218  *
219  */
220 void freeNodeP(pNode node)
221 {
222     Node* frNode = (Node*)node;
223     int i;
224     for(i = 0; i < frNode->size; i++)
225     {
226         free(frNode->matrix[i]);
227     }
228     free(frNode->matrix);
229     free(frNode);
230 }
231
232
233 /**
234  * this function get a Node and copy it
235  * @Param node a node to copy
236  * @Return a copied node
237  */
238 pNode copyNode(pNode node)
239 {
240     Node* copyNode = (Node*) malloc(sizeof(Node));
241     Node* orgNode = (Node*) node;
242     int i, j;
243     copyNode->matrix = (int**)malloc(orgNode->size * sizeof(int*));
244     int index;
245     for (index = 0; index < orgNode->size; ++index)
246     {
247         copyNode->matrix[index] = (int*)malloc((orgNode->size) * sizeof(int));
248     }
249     copyNode->size = orgNode->size;
250     copyNode->value = orgNode->value;
251
252     for(i = 0; i < orgNode->size; i++)
253     {
254         for(j = 0; j < orgNode->size; j++)
255         {
256             copyNode->matrix[i][j] = orgNode->matrix[i][j];
257         }
258     }
259 }
260
261
262 return copyNode;
263 }

```

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