Crossword Puzzle Solver CO222-2022 Project

Puzzle Compare Report

GROUP 16

E/19/309 RAMBUKWELLA H.M.W.K.G.
E/19/446 WIJERATHNA I.M.K.D.I.

TABLE OF CONTENTS

1.Project Description

1.1. Project Overview

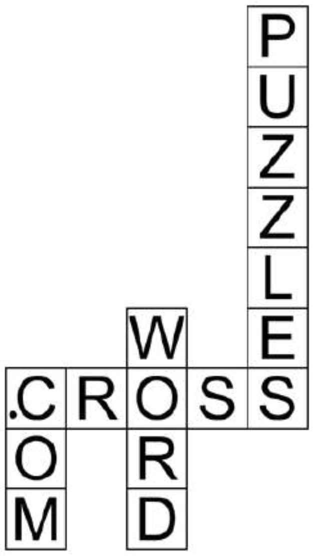
This project is intended to create a Crossword Puzzle Solver using C programming laguage which correctly solves a given crossword puzzle if the inputs are provided correctly. In phase-I, it was instructed to write the C program using static allocation of arrays and, the phase-II, it was instructed to write the C program using dynamic memory allocation. Eventually, the project expects to prepare a report comparing the two programs based on the execution time and memory space usage differences.

Specified Input Format :

- The grid
  - Which consists of '#'s and '*'s only
  - '#'s to indicate Vacant Cells
  - '*'s to indicate Blocked Cells
- Followed by an empty line
- Follwed by a list of words
  - One word per line
  - Both capital and simple letters allowed
  - Every other symbol except the 'A'-'Z' and 'a'-'z' ASCII range is not allowed
- Follwed by an empty line(To indicate the finish of giving words)

Specified Output Format :

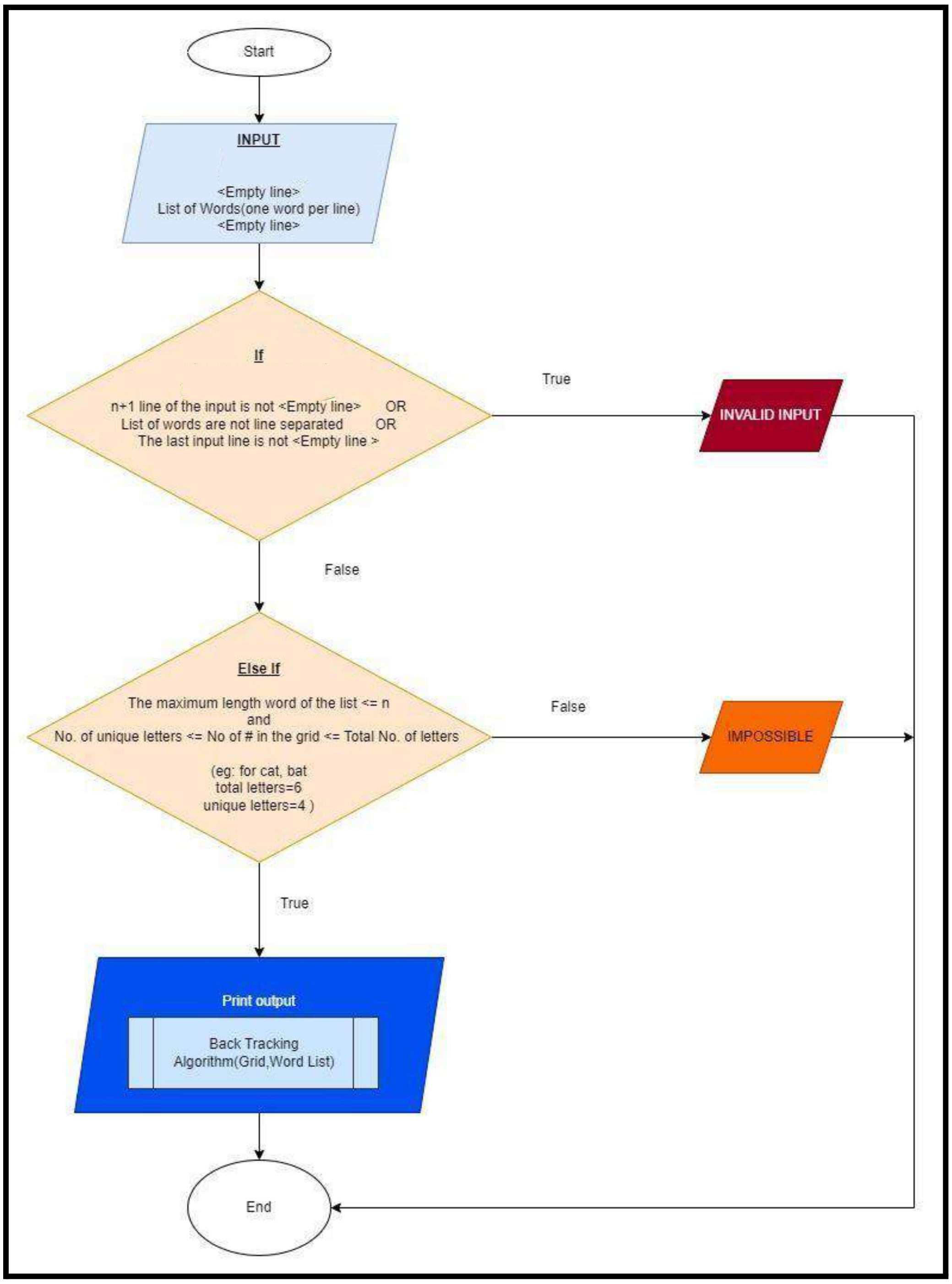
- The Completed Puzzle OR
- "IMPOSSIBLE" Message OR
- "INVALID INPUT" Message



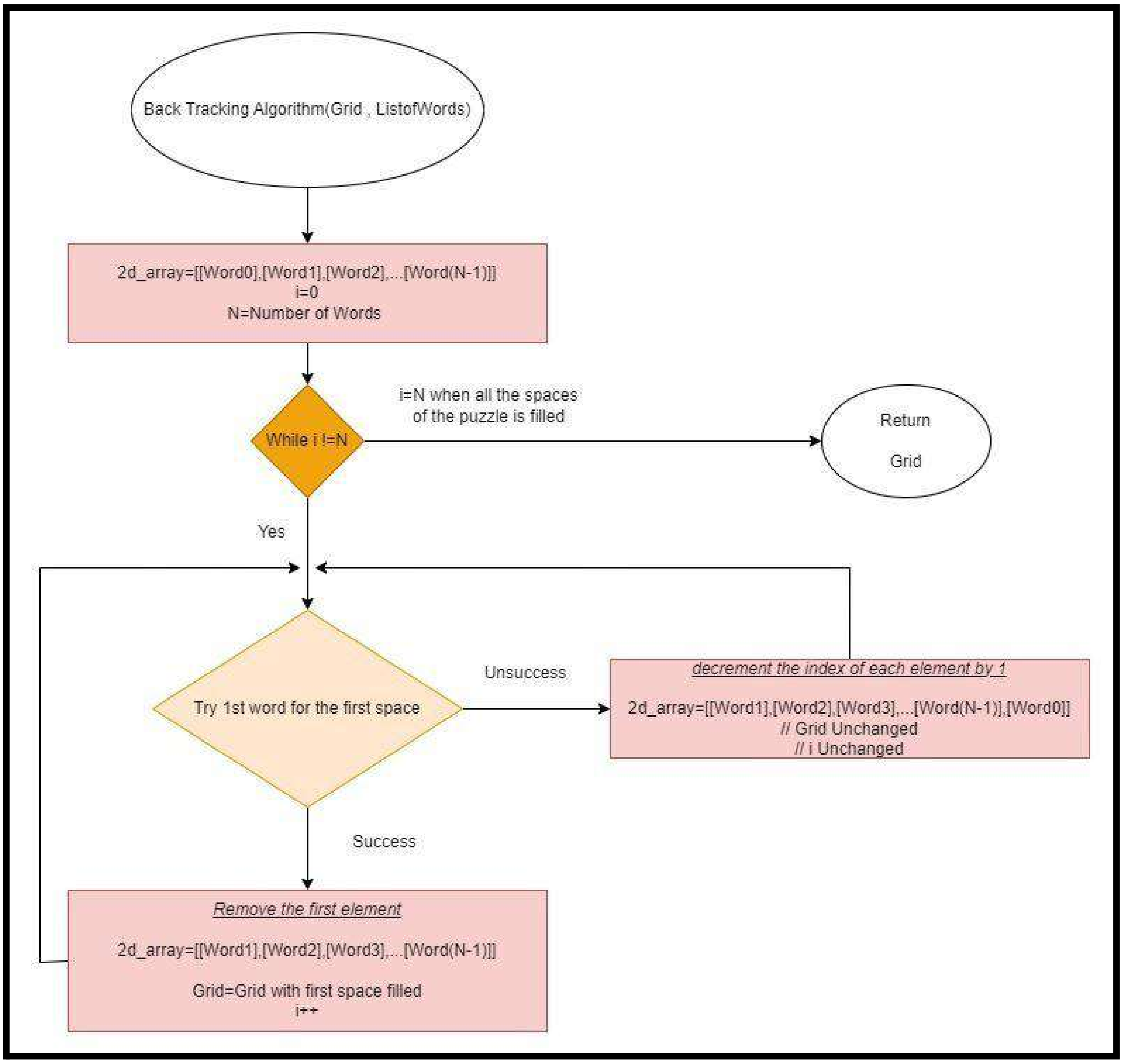
1.2. Project Preperation

Initially, we went through some online resources which are mentioned in the References to get an idea about the most optimum algorithm to be used for solving a crossword puzzle and to clarify the basics of static and dynamic memory allocation in C. Then, we came up with the 'Backtracking' algorithm which seemed to be the most optimum and easier to implement algorithm for our program. The following figures are the rough flowcharts we sketched before get into implementing the code.

The main function()


Start
INPUT
<Empty line>
List of Words(one word per line)
<Empty line>
If
n+1 line of the input is not <Empty line> OR
List of words are not line separated OR
The last input line is not <Empty line >
True
INVALID INPUT
False
Else If
The maximum length word of the list <= n
and
No. of unique letters <= No of # in the grid <= Total No. of letters
(eg: for cat, bat
total letters=6
unique letters=4)
False
IMPOSSIBLE
True
Print output
Back Tracking
Algorithm(Grid,Word List)
End

The puzzle filling function() //Using Backtracking Algorithm



Back Tracking Algorithm(Grid , ListofWords)
2d_array=[[Word0],[Word1],[Word2],...[Word(N-1)]]
i=0
N=Number of Words
While i !=N
i=N when all the spaces of the puzzle is filled
Return Grid
Yes
Try 1st word for the first space
Unsuccess
decrement the index of each element by 1
2d_array=[[Word1],[Word2],[Word3],...[Word(N-1)],[Word0]]
// Grid Unchanged
// i Unchanged
Success
Remove the first element
2d_array=[[Word1],[Word2],[Word3],...[Word(N-1)]]
Grid=Grid with first space filled
i++

When we implemented these flowcharts into our C programs we broke down both the main function and puzzle filling function into a few more small functions. Further more, we added some additional functions to optimize the code.

2. Comparison between Static-Puzzle vs Dynamic-Puzzle

2.1. Static-Puzzle

The static-puzzle consisted of the following globally declared static arrays(2D).

```c
char mtrxPuzzle[N][N] = {0}; //static 2D array to store spaces of the grid
char mtrxWords[N][N] = {0};  //static 2D array to store words of the word list
int wLenMtrx[N][5] = {0};    //static 2D array to store word lengths of each word
```

The above N is globally defined as following.

```c
//Defining the Const. value N for static array sizes for each words in the word list and each spaces in the grid
#define N 50
```

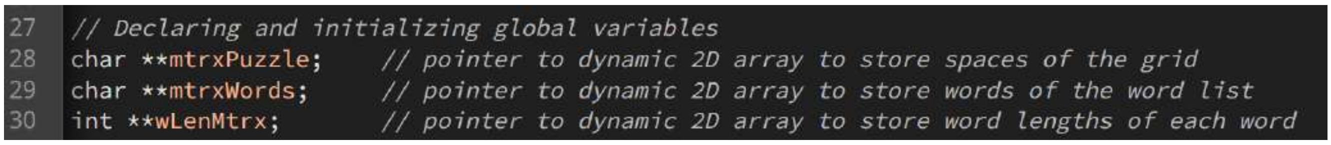
We used the following list of function prototypes in order to implement and handle the puzzle filling algorithms.

```c
//Declaring all the functions used in the program
void getMatrix(char mtrx[N][N]);
int checkPuzzle(char mtrx[N][N]);
int checkWord(char mtrx[N][N]);
void printmtrx(char mtrx[N][N]);
//void printFilemtrx(char mtrx[N][N]);
void copyMatrix(char mtrx1[N][N], char mtrx2[N][N]);
int hText(int i, int j, char mtrx[N][N]);
int vText(int i, int j, char mtrx[N][N]);
void findWordsPuzzle(char mtrx[N][N]);
int fillvert(int x, int y, int size, char mtrx[N][N], char mtrx2[N]);
int fillhor(int x, int y, int size, char mtrx[N][N], char mtrx2[N]);
int fillpuzzle(int pos, char mtrxP[N][N], char mtrxW[N][N], int mtrxWLen[N][5]);
```

These functions are used together to get the input and store the input grid and word list into two separate 2D static arrays, and then find every possible horizontal and vertical spaces in the grid. After finding each horizontal and vertical spaces, then the coordinates of each words' first character is stored in an array. Lengths of each of those empty spaces are also stored in an array. Thereafter, lengths of each words are stored in a array. Then, the backtracking algorithm is implemented to fill all the words into the grid and output the filled puzzle if possible.

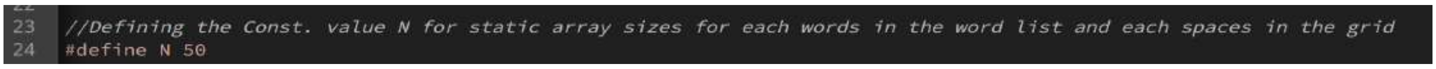
2.2. Dynamic-Puzzle

The dynamic-puzzle consisted of the following globally declared static arrays(2D).



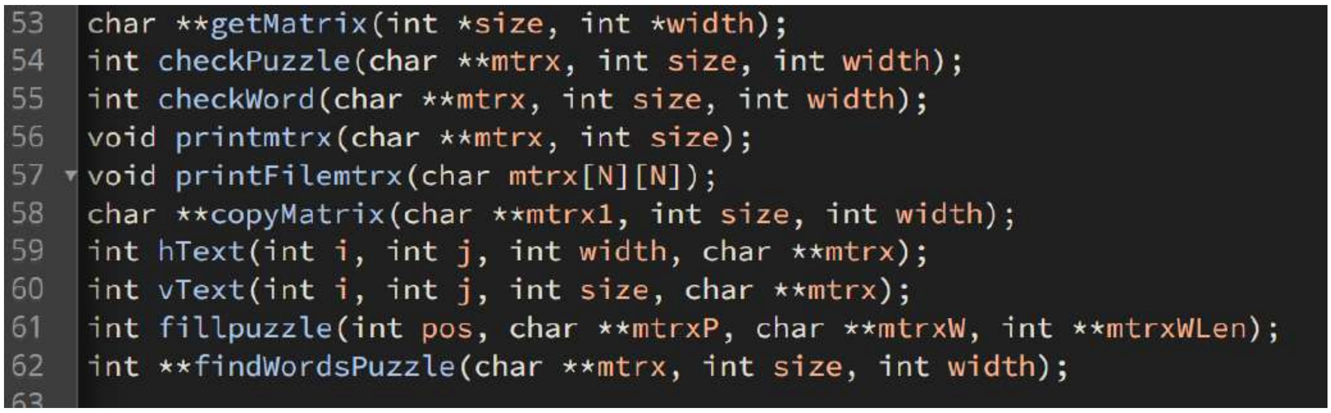
```c
// Declaring and initializing global variables
char **mtrxPuzzle;     // pointer to dynamic 2D array to store spaces of the grid
char **mtrxWords;      // pointer to dynamic 2D array to store words of the word list
int **wLenMtrx;        // pointer to dynamic 2D array to store word lengths of each word
```

The above N is globally defined as following.



```c
//Defining the Const. value N for static array sizes for each words in the word list and each spaces in the grid
#define N 50
```

We used the following list of functions ; in order to implement and handle the puzzle filling algorithms.



```c
char **getMatrix(int *size, int *width);
int checkPuzzle(char **mtrx, int size, int width);
int checkWord(char **mtrx, int size, int width);
void printmtrx(char **mtrx, int size);
void printFilemtrx(char mtrx[N][N]);
char **copyMatrix(char **mtrx1, int size, int width);
int hText(int i, int j, int width, char **mtrx);
int vText(int i, int j, int size, char **mtrx);
int fillpuzzle(int pos, char **mtrxP, char **mtrxW, int **mtrxWLen);
int **findWordsPuzzle(char **mtrx, int size, int width);
```

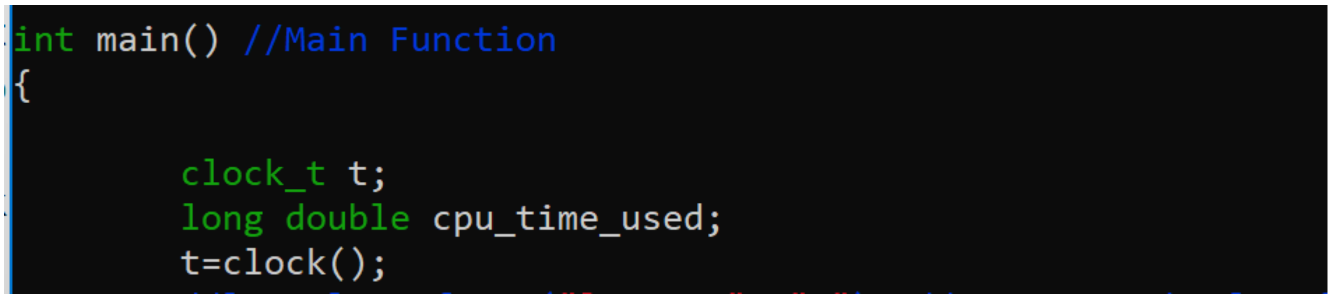
These functions are used together to get the input and store the input grid and word list into two separate dynamic arrays. Other functionalities are the same as the static-puzzle.

<u>More specific descriptions about the above functions of both of the programs are mentioned in the .c files which were submitted.</u>

2.3. Comparison of the Execution Time

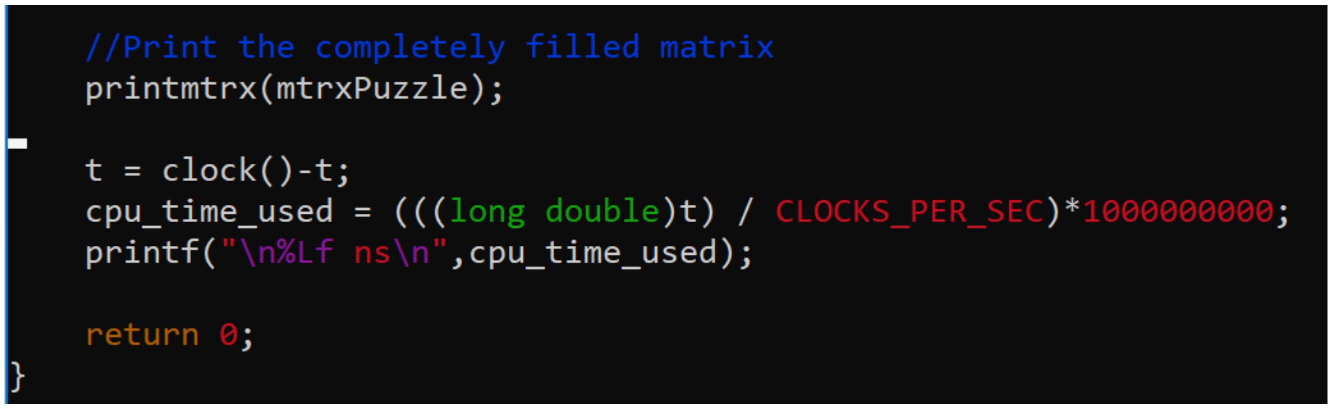
We have compared the execution time of the two programs using clock() function in <time.h> library.

//Calling the clock() function just after starting the main function to get the starting cpu time



```
int main() //Main Function
{

        clock_t t;
        long double cpu_time_used;
        t=clock();
```

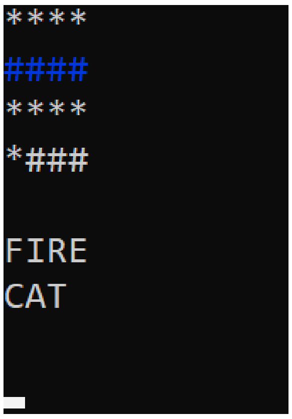
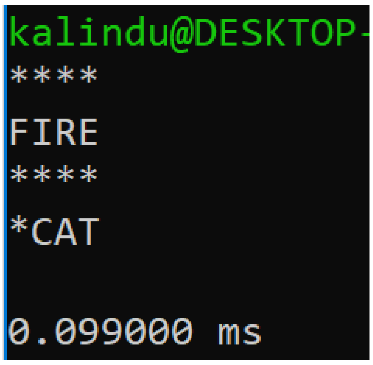
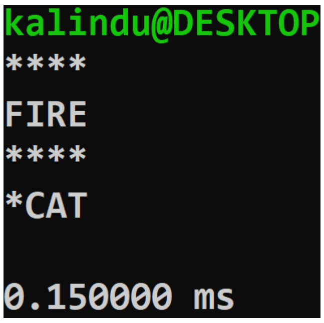
//Get the ending cpu time just before ending the main function

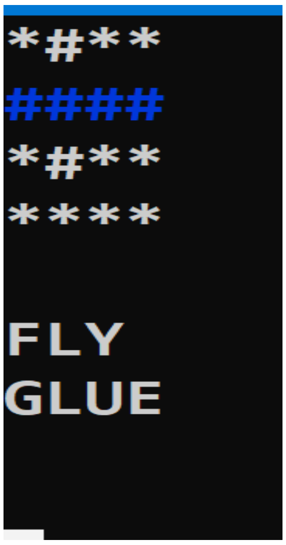
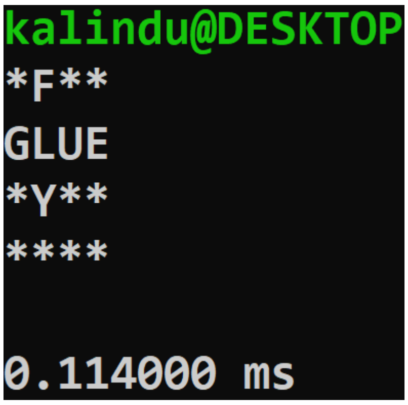
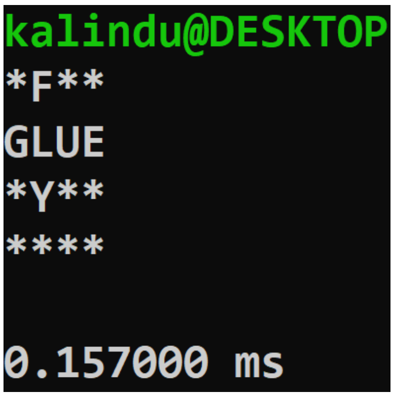


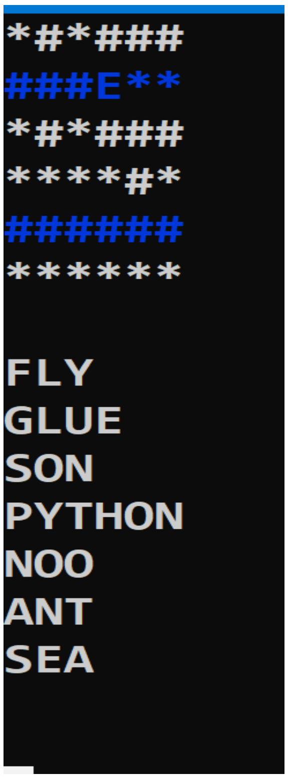
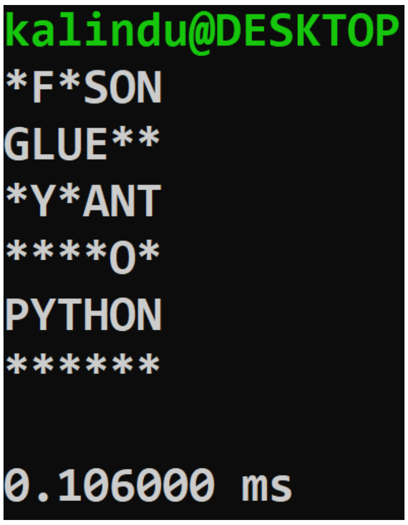
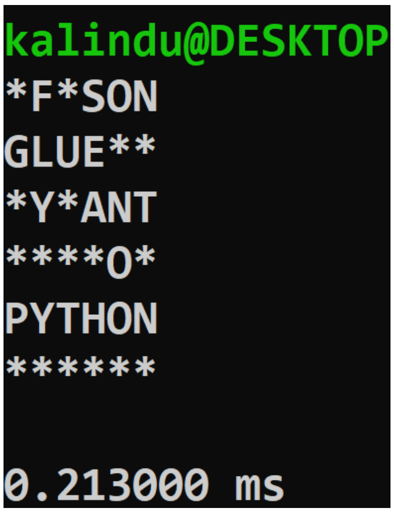
```
    //Print the completely filled matrix
    printmtrx(mtrxPuzzle);

    t = clock()-t;
    cpu_time_used = (((long double)t) / CLOCKS_PER_SEC)*1000000000;
    printf("\n%Lf ns\n",cpu_time_used);

    return 0;
}
```

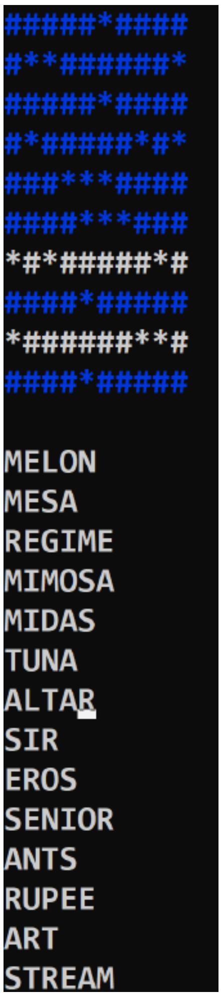
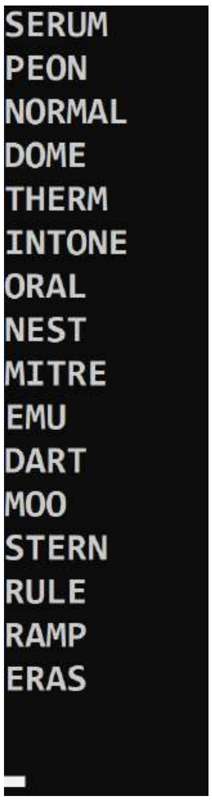
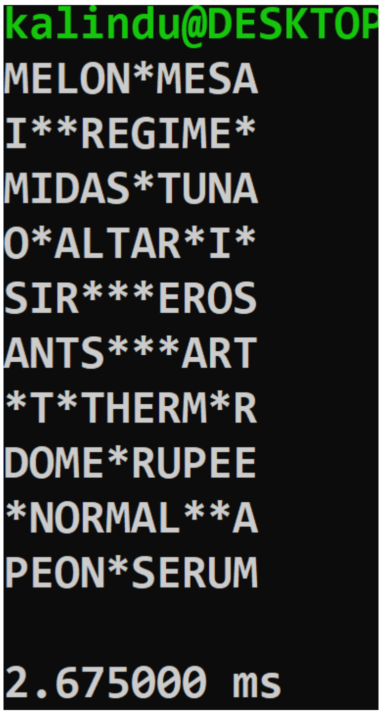
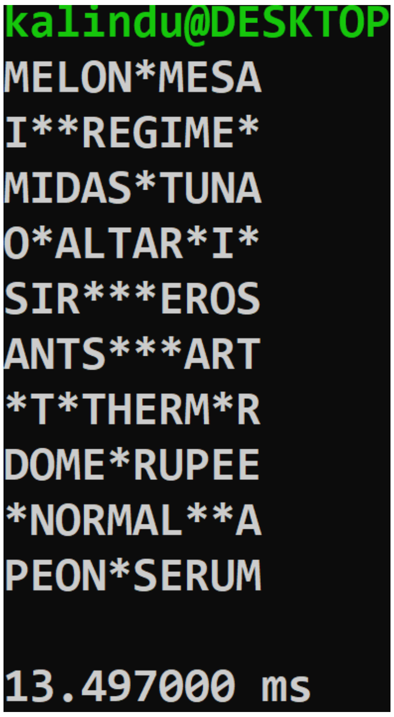
<u>Following table consists a variety of test cases we tried in order to compare the execution times.</u>

	Test Case	puzzle-static.c	puzzle-dynamic.c

1.	 `****`
	`####`
	`****`
	`*###`
	
	`FIRE`
	`CAT`
		 `kalindu@DESKTOP-`
		`****`
		`FIRE`
		`****`
		`*CAT`
		
		`0.099000 ms`
			 `kalindu@DESKTOP`
			`****`
			`FIRE`
			`****`
			`*CAT`
			
			`0.150000 ms`

2.	 `*#**`
	`####`
	`*#**`
	`****`
	
	`FLY`
	`GLUE`
		 `kalindu@DESKTOP`
		`*F**`
		`GLUE`
		`*Y**`
		`****`
		
		`0.114000 ms`
			 `kalindu@DESKTOP`
			`*F**`
			`GLUE`
			`*Y**`
			`****`
			
			`0.157000 ms`

3.	 `*#*###`
	`###E**`
	`*#*###`
	`****#*`
	`######`
	`******`
	
	`FLY`
	`GLUE`
	`SON`
	`PYTHON`
	`NOO`
	`ANT`
	`SEA`
		 `kalindu@DESKTOP`
		`*F*SON`
		`GLUE**`
		`*Y*ANT`
		`****O*`
		`PYTHON`
		`******`
		
		`0.106000 ms`
			 `kalindu@DESKTOP`
			`*F*SON`
			`GLUE**`
			`*Y*ANT`
			`****O*`
			`PYTHON`
			`******`
			
			`0.213000 ms`

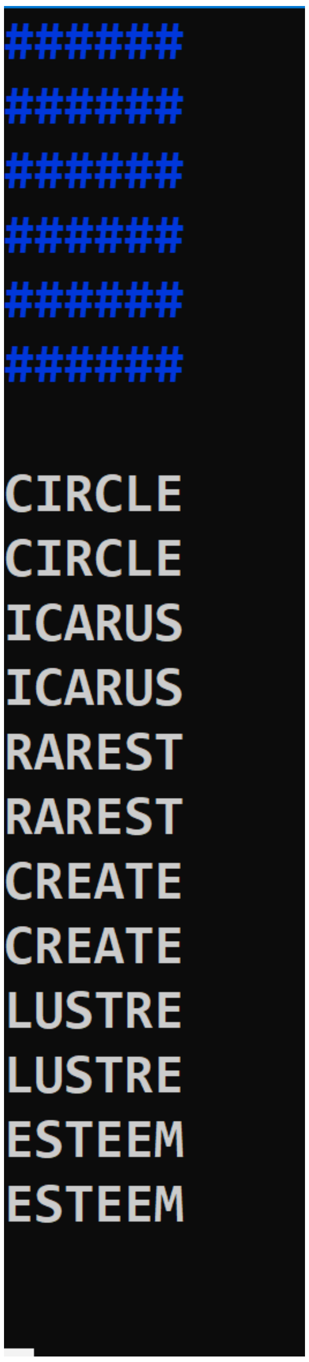
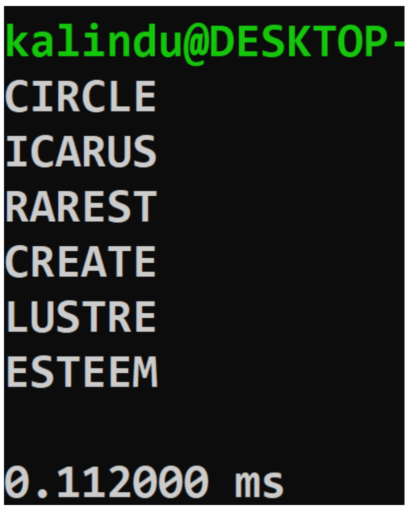
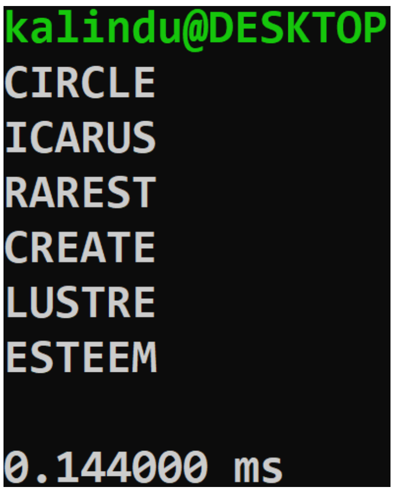
4.	 		

```
#####*####
#**#######*
#####*####
#*######*#*
###***####
####**###
*#*######*#
####*#####
*#######**#
####*#####

MELON
MESA
REGIME
MIMOSA
MIDAS
TUNA
ALTAR
SIR
EROS
SENIOR
ANTS
RUPEE
ART
STREAM
```

```
SERUM
PEON
NORMAL
DOME
THERM
INTONE
ORAL
NEST
MITRE
EMU
DART
MOO
STERN
RULE
RAMP
ERAS
```

```
kalindu@DESKTOP
MELON*MESA
I**REGIME*
MIDAS*TUNA
O*ALTAR*I*
SIR***EROS
ANTS***ART
*T*THERM*R
DOME*RUPEE
*NORMAL**A
PEON*SERUM

2.675000 ms
```

```
kalindu@DESKTOP
MELON*MESA
I**REGIME*
MIDAS*TUNA
O*ALTAR*I*
SIR***EROS
ANTS***ART
*T*THERM*R
DOME*RUPEE
*NORMAL**A
PEON*SERUM

13.497000 ms
```

5.	
	```
	#######
	#######
	#######
	#######
	#######
	#######

	CIRCLE
	CIRCLE
	ICARUS
	ICARUS
	RAREST
	RAREST
	CREATE
	CREATE
	LUSTRE
	LUSTRE
	ESTEEM
	ESTEEM
	```
		
		```
		kalindu@DESKTOP-
		CIRCLE
		ICARUS
		RAREST
		CREATE
		LUSTRE
		ESTEEM

		0.112000 ms
		```
			
			```
			kalindu@DESKTOP
			CIRCLE
			ICARUS
			RAREST
			CREATE
			LUSTRE
			ESTEEM

			0.144000 ms
			```

Conclusions:

- Generally for almost all the test cases the execution **took a longer time in dynamic-puzzle.c** compared to static-puzzle.c .
- In Static memory allocation, the memory is allocated at the compile time. However, in Dynamic memory allocation, the memory is allocated at the run time. Therefore, theoretically the Dynamically-memory allocated c program should be(most probably) the one which takes longer time for execution compared to the Statically-memory allocated c program.
- Hence, we can conclude that our c programs static-puzzle.c and dynamic-puzzle.c agree with the theoretical behaviors of the execution times of the above memory allocation types.

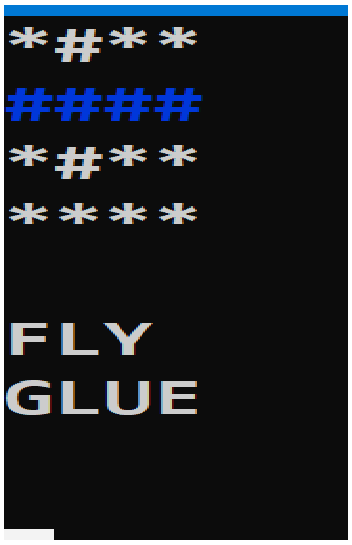
2.5. Comparison of Memory Space Usage

- In staitc-puzzle.c the memory is allocated at the compile time and the variables gets memory allocated permanently(size is fixed) and it is not reusable. This uses stack for managing the memory allocation. Furthermore, we cannot reuse the unused memory. Even Though the execution time is less; the memory usage in this program is comparatively less efficient.

- In dynamic-puzzle.c the memory is allocated at the run time and the memory gets allocated only if the program unit gets active and it is reusable as well as have the ability to free when not required. This uses heap for managing the memory allocation. Also it is possible to change the allocated memory size. The execution time is higher in this memory allocation method. However, in terms of memory usage this is far more efficient.

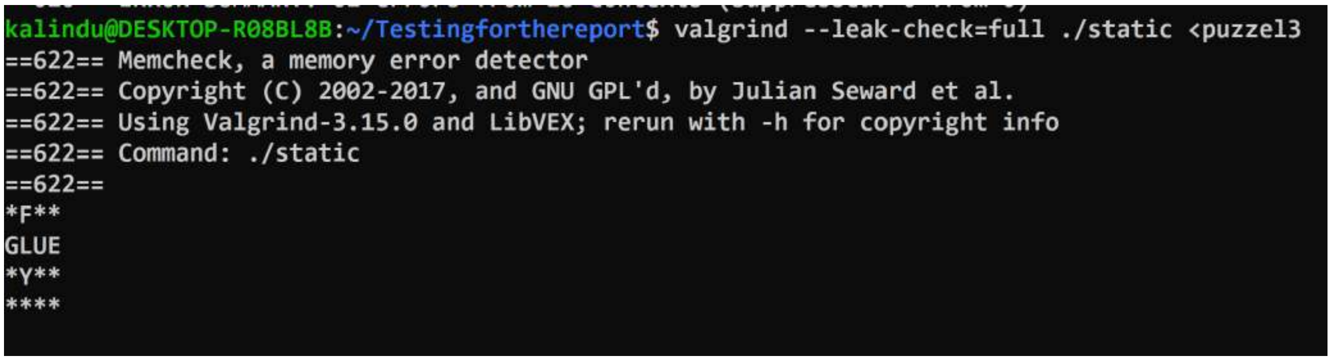
We used “Valgrind” to analyze the memory usage of both the programs using a common testcase.

INPUT :

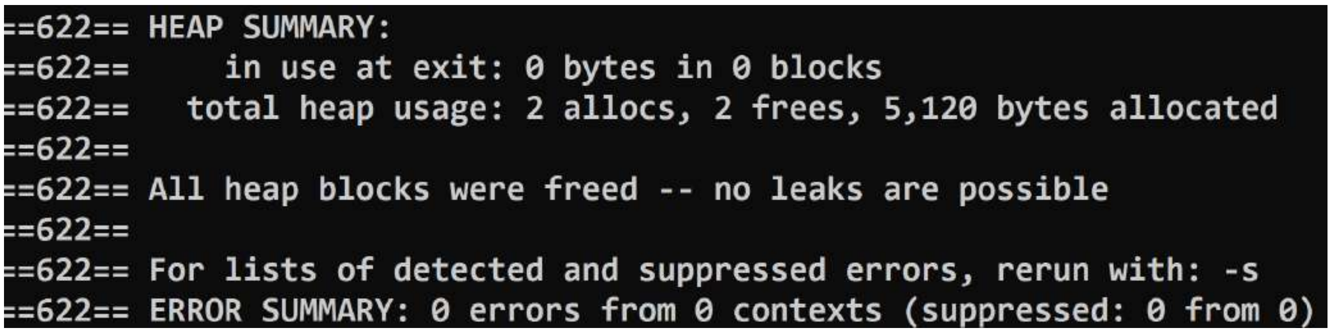


```
*#**
#####
*#**
****


FLY
GLUE
```

Valgrind Analysis - for the static-puzzle.c

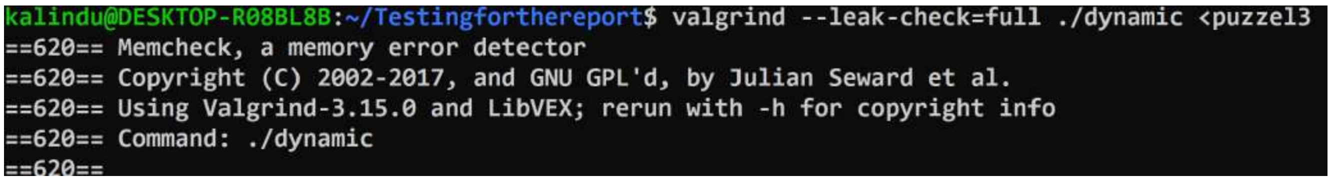


```
kalindu@DESKTOP-R08BL8B:~/Testingforthereport$ valgrind --leak-check=full ./static <puzzel3
==622== Memcheck, a memory error detector
==622== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==622== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==622== Command: ./static
==622==
*F**
GLUE
*Y**
****
```

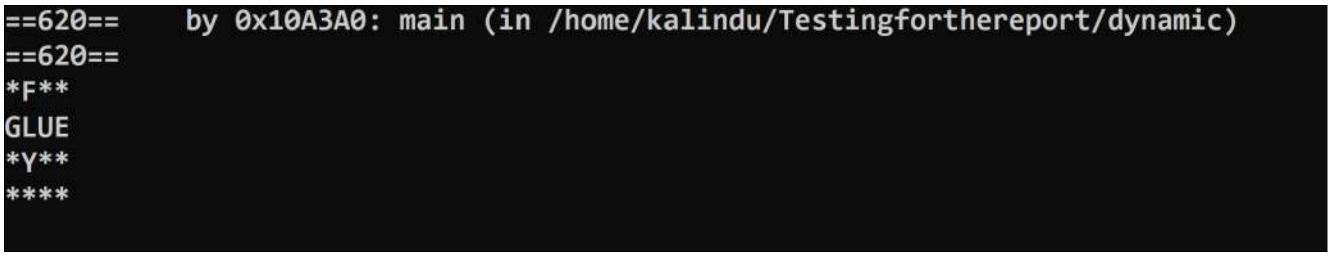


```
==622== HEAP SUMMARY:
==622==     in use at exit: 0 bytes in 0 blocks
==622==   total heap usage: 2 allocs, 2 frees, 5,120 bytes allocated
==622==
==622== All heap blocks were freed -- no leaks are possible
==622==
==622== For lists of detected and suppressed errors, rerun with: -s
==622== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

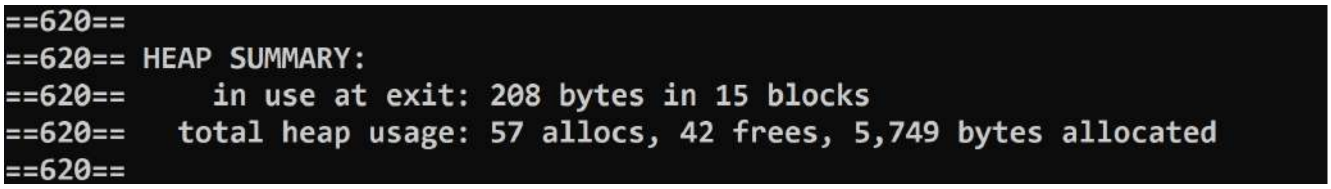
Valgrind Analysis - for the dynamic-puzzle.c



```
kalindu@DESKTOP-R08BL8B:~/Testingforthereport$ valgrind --leak-check=full ./dynamic <puzzel3
==620== Memcheck, a memory error detector
==620== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==620== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==620== Command: ./dynamic
==620==
```



```
==620==    by 0x10A3A0: main (in /home/kalindu/Testingforthereport/dynamic)
==620==
*F**
GLUE
*Y**
****
```



```
==620==
==620== HEAP SUMMARY:
==620==     in use at exit: 208 bytes in 15 blocks
==620==   total heap usage: 57 allocs, 42 frees, 5,749 bytes allocated
==620==
```

7.References

1) HackerRank - Crossword Puzzle - Problem Analysis and Logic Building

18/12//2022

2) https://en.wikipedia.org/wiki/Crossword

06/01/2023

3) https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

09/01/2023

4) Pointers in C / C++ [Full Course]

11/01/2023

5) https://www.youtube.com/playlist?list=PLBlnK6fEyqRjoG6aJ4FvFU1tlXbjLBiOP

11/01/2023