Word Search Puzzle using Multi-Linked Lists

Shivam Hareshkumar Patel, A. Jackulin Mahariba

Abstract*: Word Search Puzzle is a game in which a player is supposed to find words in the given grid of characters, by unveiling the clues stated for those words. This paper presents the creation of the grid of characters in the Word Search Puzzle using multi-linked list data structure. The primary motivation to create this game using multi-linked list data structure was the efficiency of a linked list to handle data - insertion and deletion of an element to the list. The model which was created using a multi-linked list as described in this paper will now be referred to as Model A, for better understanding. The construction of Model A is described as a 5-step process with the help of algorithms. The time complexities to create the grid for Model A and to access data from it were analyzed and found to be much better than the other existing models in the market.

Index Terms*: Multi-Linked List, Word Search Puzzle, Data Structure, Algorithm.

I. INTRODUCTION

A. Data Structure

The data structure used in the creation of Model A is a linked list. The linked list is a linear collection of data elements. The data elements are referred to as nodes of the linked list. Each node has two parts. One part is called the data field, which stores data related to that node and the other part called the pointer field, contains a reference to the other nodes of the list. Depending on the type of linked list, the nodes can have one or more data and pointer fields in them. Two such types of linked lists are explained below:

Singly Linked List: The simplest type of linked list is the singly linked list. In a singly linked list, every node has only one data field for storing data and only one pointer field for storing the reference to another node of the list.

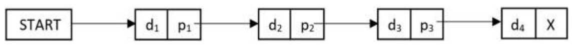

Fig. 1: Sample Singly Linked List

Every linked list contains a pointer variable START that stores the address of the first node of the list. If the START = NULL, then the list is empty, i.e. it does not contain any nodes. The last node of the list does not point to any other node, so its pointer is assigned the value NULL and is represented as 'X'. The d_1, d_2, d_3, d_4 are the data fields and the p_1, p_2, p_3 are the pointers.

The general representation of a node of a singly linked list is as follows:

```
struct node
{
int data;
struct node *next;
};
```

The 'data' variable stores the data for the node and the pointer variable '*next' is used as a pointer to another node of a similar type (self-referential data type). The key advantage of a linked list is that it efficiently allows to insert and delete nodes in the list. Another type of linked list is the multi-linked list which was used for the creation of Model A.

Multi-Linked List: In a multi-linked list, unlike the singly linked list, each node can have more than one data field and more than one pointer.

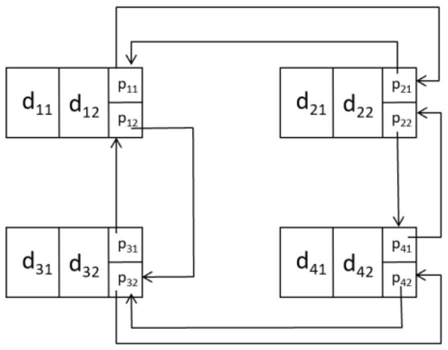

Fig. 2: Sample Multi-Linked List

In the above multi-linked list, each node has two data fields and two pointer fields. The d_{11}, d_{12}, d_{21}, d_{22}, d_{31}, d_{32}, d_{41}, d_{42} are the data fields and the p_{11}, p_{12}, p_{21}, p_{22}, p_{31}, p_{32}, p_{41}, p_{42} are the pointer fields.

The representation of a node of the above multi-linked list is as follows:

```
struct node
{
int data1, data2;
struct node *next1,
*next2;
};
```

Here, 'data1' and 'data2' variables store data of the node, and the '*next1' and '*next2' variables, are pointers of the node, that point to other nodes of similar type.

Advantages of linked lists: Insertion and deletion of elements (nodes) in a linked list are easy and efficient tasks than as compared to that in an array. This is because in an array we shift the data in its memory locations to add or remove elements, but in a linked list, we can just update the address stored by the pointers of the nodes. Linked lists offer efficient memory utilization. In an array, if some memory is allocated, it cannot be de-allocated from the array whereas, in a linked list, allocation and de-allocation of memory can be done at any time. Also, linked lists are dynamic data structures, i.e. they can grow or shrink efficiently during the execution of the program.

Revised Manuscript Received on April 25, 2019.

Shivam Hareshkumar Patel, UG Student, Department of Computer Science and Engineering, SRM Institute of Science and Technology, Kattankulathur, Chennai, Tamil Nadu, India. E-mail: shivam.patel1606@gmail.com

A. Jackulin Mahariba, Assistant Professor (Senior Grade), Department of Computer Science and Engineering, SRM Institute of Science and Technology, Kattankulathur, Chennai, Tamil Nadu, India. E-mail: jackulin.a@ktr.srmuniv.ac.in

Disadvantages of linked lists: Since the nodes of a linked list do not have an index assigned to them like in arrays, it is difficult to locate nodes in some linked list than in an array. In a linked list, if the nth node was to be accessed, it is necessary to traverse through all the (n-1) nodes that appear before it. Thus, the time required to access a node is too large. Further, node or element traversal in a linked list is difficult, unlike in an array where we can easily traverse it using the indexes. For traversing through the linked list, a lot of alterations in the addresses stored in the pointers are required, which consumes a lot of time.

II. LITERATURE REVIEW

The work done by V. R. Kanagavalli et al. [1] proposes the effective utilization of data structures in the area of information retrieval. Đorđe Stojisavljević et al. [2] depicts the implementation of multi-linked lists in C++ by elaborating on the object-oriented concepts of multi-linked lists. Li-Bing Wu et al. [3] suggests an improved data structure called ‘tacked list’ to enhance the advanced resource reservation mechanism in high-performance networks and distributed systems. Đorđe Stojisavljević et al. [4] proposes a way to improve on the traditional methods of accessing data using multi-linked lists. Vossoughi Hossein et al. [7] proposed the results of word search puzzle game in vocabulary development of 60 female students of a language institute in Semnan. Falley. P [9] detects the differences and similarities in the three categories formed after grouping the different data structures. The categories proposed in [9] were Storage Structures (linked lists, arrays and hash tables), Process-Oriented Data Structures (queues, priority queues, stacks and iterators) and, Descriptive Data Structures (binary trees, sets, collections, etc.). Jeremy Lee Graybill [11] has researched an integration of word scramble, trivia puzzle, word-link and word-search puzzle games. Charles William Ditter [12] proposed a model for a three-dimensional word-search puzzle game. Burston Jack [13] suggested the essential role of possessing a good vocabulary in learning a foreign or a second language and how crossword puzzles help in the same. Parlante N. [14] presented the basic and essential techniques to construct a linked list and in [15], he proposes and explains 18 practice problems related to linked lists.

Morin [5] thoroughly described the working of various data structures on strings. Reema Thareja [6] has detailed the concepts regarding the use of various data structures. It provided a major help to us in formatting the algorithm that was developed for our model. Tanenbaum A. et al. [8] have described the use of the numerous data structures using C and C++. Kenneth A. Reek [10] has clearly explained the use to pointers in C.

The model proposed in [16] uses integers to represent the English alphabets. It has a fixed number of words that can be added to the puzzle grid and also it has assigned a fixed length for each of those words, which implies that the number of words and their length cannot be assigned in runtime. The crossword generator proposed in [17] uses C# and Visual Studio 2010 for its development. For each word to be placed in some position in the grid, it checks if it is a valid position followed by checking whether it can be placed at that position. Finally, when both the above conditions return true, it puts the word at that position. The application as described in [17] was meant only for demonstration purposes which creates a matrix of 13 X 17. It tries to place the list of given words randomly while seeking for the optimal result for up to one minute, which is not optimizing. The word search generator/solver described in [18], proposes four types of grids of alphabets. This model is constructed using arrays which has a function defined to check whether a word can be inserted in the grid or not at the denoted position and if it returns true, it inserts the word in to the grid using a different function. The model proposed in [19] can form grids of the word search puzzle, of greater sizes efficiently as compared to the other models described above. So, we have kept this model [19], as a reference model to compare its efficiency with that of Model A. The model described in [19] will now be referred as Model B, for better understanding.

III. PROPOSED WORK

A. Approach

The grid that we created for Model A, was made using multi-linked list data structure, and each node of that model had one data field and three pointer fields. A sample node of our model is shown in Fig. 3. In the figure, d_1 is the data field of the node and p_{11}, p_{12}, p_{13} are the pointer fields of the node. The pointer fields either point to similar nodes of the model or they store a NULL value, if they do not point to any other nodes. Each data field stores an alphabetical letter as required by the model (puzzle). The general representation of a node of our model is as follows:

```
struct node
{
char DATA;
struct node *RIGHT; struct node *BOTTOM; struct node
*CROSS;
};
```

The ‘DATA’ variable stores the data of the node, i.e. and alphabetical letter and ‘RIGHT’, ‘BOTTOM’ and ‘CROSS’ are the pointer variables of the node. The pointer variables if pointing to some node, store the address of similar nodes, else if they do not point to any node then they store a NULL value. Comparing with Fig. 3, d_1 is the DATA field, p_{11} represents the RIGHT pointer, p_{12} represents the CROSS pointer and p_{13} represents the BOTTOM pointer of the node. A sample 4 x 4 grid of our model is shown in Fig. 4. It shows a connection of 16 single nodes, each node of the form as described in Fig. 3. A few nodes have pointer fields storing NULL value because they do not point to any other nodes.

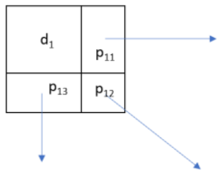

Fig. 3: Single Node of Model A

In the next sub-section under proposed work, we have described the algorithms for the construction of our model.

The algorithms have helped us achieve better efficiency than other models while constructing the grid and even better efficiency while accessing the data stored in the nodes.

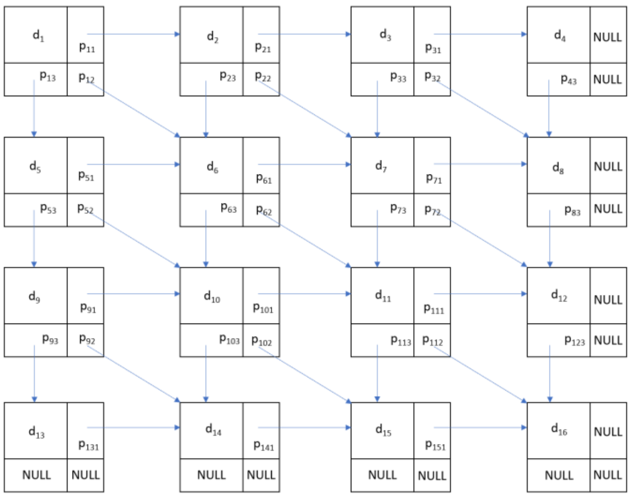

Fig. 4: Sample Structure of Model A

B. Algorithm for Construction of Model A

In this sub-section we have explained the algorithms that were used for the construction of the N x N crossword grid of Model A (where N is the size of one side of the grid, which is called as ‘PSIZE’ in the algorithms discussed below). The entire algorithm for construction is divided into five major steps (functions) which are as follows:

i. makePuzzle()
ii. setBottomLinks()
iii. setMainDiagonalLinks()
iv. setUpperTriangularLinks()
v. setLowerTriangularLinks()

Each of the above steps are explained in detail below:

i. makePuzzle()

In this step we are creating the basic structure of the model with all the nodes containing its data. Also, all the required RIGHT pointers of the model are set using this algorithm. The algorithm of the setBottomLinks() function is described below and the structure of the model after the completion of this step is shown in Fig. 5 (red arrows represent the links added in this step of the algorithm).

Algorithm

```
STRUCT NODE *makePuzzle()
Step 1: CREATE NODE *TEMP_START, *NEW_NODE
Step 2: SET TEMP_START = START
Step 3: Repeat steps 4 to 14 FOR i FROM 1 TO PSIZE
Step 4:     GET DATA
Step 5:     SET START -> DATA = DATA
Step 6:     SET START -> RIGHT = NULL
Step 7:     SET START -> BOTTOM = NULL
Step 8:     SET START -> CROSS = NULL
Step 9:     Repeat steps 10 to 11 FOR j FROM 2 TO PSIZE
Step 10:        GET DATA
Step 11:        SET START = insert_end(START, DATA);
            [END OF LOOP]
Step 12:    IF i != PSIZE
Step 13:        SET START -> BOTTOM = NEW_NODE
Step 14:        SET START = NEW_NODE
            [END OF IF]
[END OF LOOP]
Step 15: SET START = TEMP_START
Step 16: RETURN START
Step 17: EXIT
```

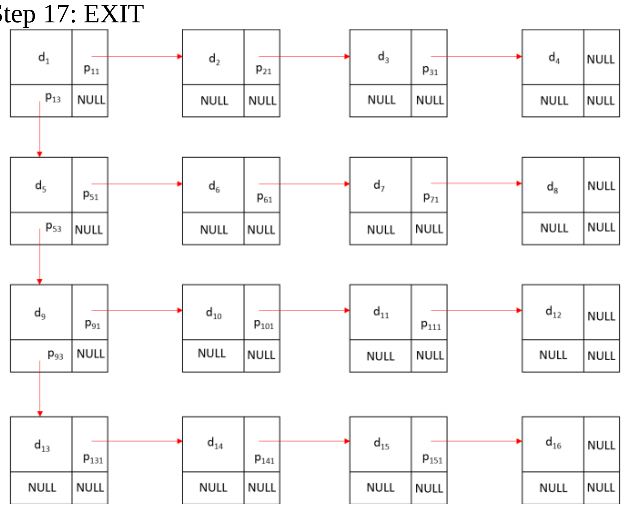

Fig. 5: Model A after execution of makePuzzle()

ii. setBottomLinks()

After the basic model was created using the makePuzzle() function, all the vertical links between the nodes were created. We made a setBottomLinks() function, to do the same. Thus, using the setBottomLinks() functions, all the required BOTTOM pointers were made to store the index of the necessary node. And the rest of the unrequired BOTTOM pointers were kept as NULL. The algorithm of the setBottomLinks() function is described below and the structure of the model after the completion of this step is shown in Fig. 6 (red arrows represent the links added in this step of the algorithm and blue arrows represent the links that were already present at the initiation of this step).

Algorithm

```
STRUCT NODE *setBottomLinks()
Step 1: CREATE NODE *RIGHT_WORKER, *BOTTOM_WORKER, *CHECK, BOTTOM_WORKER_INSTANCE

Step 2: SET RIGHT_WORKER = START
Step 3: SET BOTTOM_WORKER = START
Step 4: SET CHECK = START
Step 5: SET BOTTOM_WORKER_INSTANCE = START

Step 6: Repeat steps 7 to 14 WHILE CHECK -> BOTTOM != NULL
Step 7:     SET CHECK = CHECK -> BOTTOM
Step 8:     SET RIGHT_WORKER = BOTTOM_WORKER_INSTANCE -> RIGHT
Step 9:     SET BOTTOM_WORKER = CHECK
Step 10:    SET BOTTOM_WORKER_INSTANCE = BOTTOM_WORKER;
Step 11:    Repeat steps 12 to 14 WHILE BOTTOM_WORKER -> RIGHT != NULL
Step 12:        SET BOTTOM_WORKER = BOTTOM_WORKER -> RIGHT;
Step 13:        SET RIGHT_WORKER -> BOTTOM = BOTTOM_WORKER;
Step 14: IF RIGHT_WORKER -> RIGHT != NULL
            SET RIGHT_WORKER = RIGHT_WORKER -> RIGHT;
        [END OF IF] [END OF LOOP]
    [END OF LOOP]
Step 15: RETURN START;
Step 16: EXIT
```

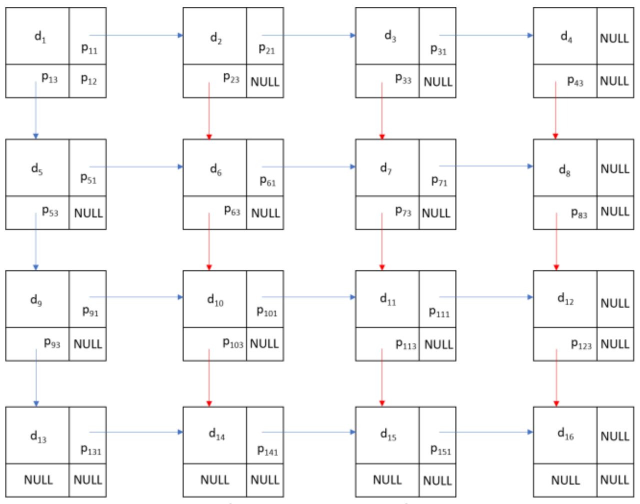

Fig. 6: Model A after execution of setBottomLinks()

iii. set Main Diagonal Links()

We have made this function to assign index value of the appropriate nodes to all the CROSS pointers that aid in creating the main diagonal of the grid structure. The algorithm of the setMainDiagonalLinks() function is described below and the structure of the model after the completion of this step is shown in Fig. 7 (red arrows represent the links added in this step of the algorithm and blue arrows represent the links that were already present at the initiation of this step).

Algorithm

```
STRUCT NODE *setMainDiagonalLinks()
Step 1: CREATE NODE *INSTANCE, *BOTTOM_WORKER, *TO_ADD
Step 2: SET CHECK = 1
Step 3: SET INSTANCE = START
Step 4: SET BOTTOM_WORKER = START
Step 5: SET TO_ADD = START
Step 6: Repeat steps 7 to 15 WHILE CHECK < PSIZE
Step 7:     SET INSTANCE = INSTANCE -> BOTTOM
Step 8:     SET BOTTOM_WORKER = INSTANCE
Step 9:     SET N = 1
Step 10:    Repeat steps 11 to 12 WHILE N <= CHECK
Step 11:        SET BOTTOM_WORKER = BOTTOM_WORKER -> RIGHT
Step 12:        SET N = N + 1
            [END OF LOOP]
Step 13:    SET TO_ADD -> CROSS = BOTTOM_WORKER
Step 14:    SET TO_ADD = BOTTOM_WORKER
Step 15:    SET CHECK = CHECK + 1
        [END OF LOOP]
Step 16: RETURN START
Step 17: EXIT
```

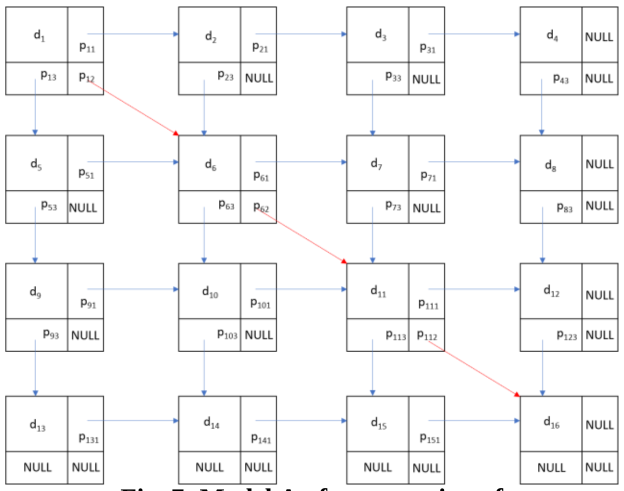

Fig. 7: Model A after execution of setMainDiagonalLinks()

iv. set Upper Triangular Links()

This function was made to set all the CROSS pointers of the upper triangular nodes of the grid excluding the nodes of the main diagonal. Those CROSS pointers were assigned the indices of the appropriate nodes of the grid. The algorithm of the setUpperTriangularLinks() function is described below and the structure of the model after the completion of this step is shown in Fig. 8 (red arrows represent the links added in this step of the algorithm and blue arrows represent the links that were already present at the initiation of this step).

Algorithm

```
STRUCT NODE * setUpperTriangularLinks()
Step 1: CREATE NODE *BOTTOM_WORKER, *TEMP_RIGHT_WORKER, *RIGHT_WORKER
Step 2: SET TEMP_RIGHT_WORKER = START -> RIGHT
Step 3: Repeat steps 4 to 11 WHILE TEMP_RIGHT_WORKER -> RIGHT != NULL
Step 4:     SET RIGHT_WORKER = TEMP_RIGHT_WORKER
Step 5:     SET BOTTOM_WORKER = RIGHT_WORKER -> BOTTOM -> RIGHT
Step 6:     SET RIGHT_WORKER -> CROSS = BOTTOM_WORKER
Step 7:     Repeat steps 8 to 10 WHILE BOTTOM_WORKER -> RIGHT != NULL
Step 8:         SET RIGHT_WORKER = BOTTOM_WORKER
Step 9:         SET BOTTOM_WORKER = BOTTOM_WORKER ->BOTTOM -> RIGHT
Step 10:        SET RIGHT_WORKER -> CROSS = BOTTOM_WORKER
            [END OF LOOP]
Step 11:    SET TEMP_RIGHT_WORKER = TEMP_RIGHT_WORKER -> RIGHT
        [END OF LOOP]
Step 12: RETURN START
Step 13: EXIT
```

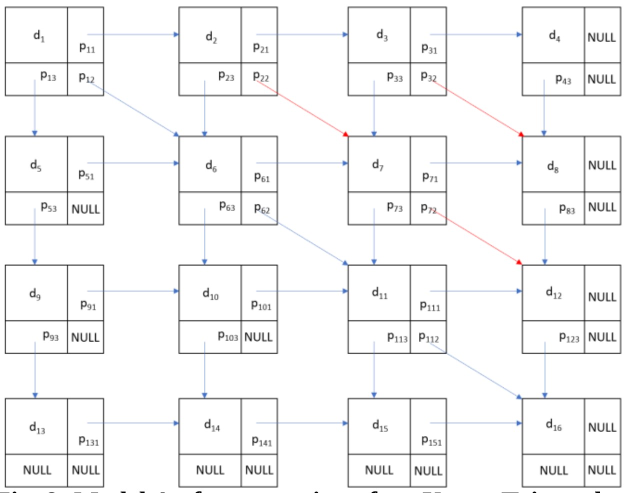

Fig. 8: Model A after execution of set Upper Triangular Links ()

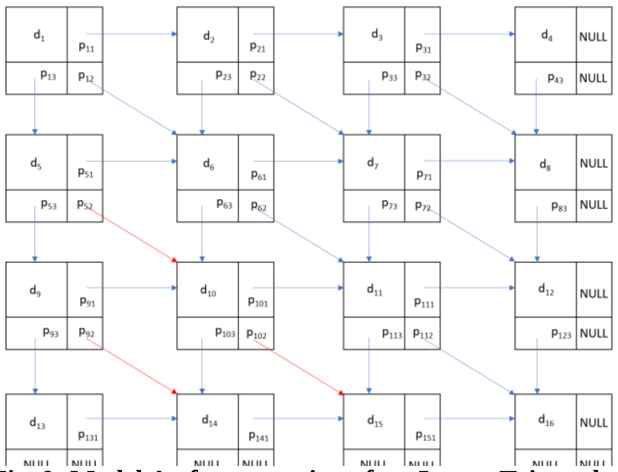

Fig. 9: Model A after execution of set Lower Triangular Links ()

v. setLowerTriangularLinks()

This function was made to set all the CROSS pointers of the lower triangular nodes of the grid excluding the nodes of the main diagonal. Those CROSS pointers were assigned the indices of the appropriate nodes of the grid. The algorithm of the setLowerTriangularLinks() function is described below and the structure of the model after the completion of this step is shown in Fig. 9 (red arrows represent the links added in this step of the algorithm and blue arrows represent the links that were already present at the initiation of this step).

Algorithm

```
STRUCT NODE *setLowerTriangularLinks()
Step 1: CREATE NODE *BOTTOM_WORKER,
*TEMP_BOTTOM_WORKER, *RIGHT_WORKER
   Step 2: SET TEMP_BOTTOM_WORKER = START -> BOTTOM
   Step 3: Repeat steps 4 to 11 WHILE TEMP_BOTTOM_WORKER -> BOTTOM != NULL
   Step 4: SET BOTTOM_WORKER = TEMP_BOTTOM_WORKER
   Step 5: SET RIGHT_WORKER = BOTTOM_WORKER -> BOTTOM -> RIGHT
   Step 6: SET BOTTOM_WORKER -> CROSS = RIGHT_WORKER
   Step 7: Repeat steps 8 to 10 WHILE RIGHT_WORKER -> BOTTOM != NULL
   Step 8: SET BOTTOM_WORKER = RIGHT_WORKER
   Step 9: SET RIGHT_WORKER = RIGHT_WORKER -> BOTTOM -> RIGHT
Step 10: SET BOTTOM_WORKER -> CROSS =
RIGHT_WORKER
[END OF LOOP]
Step 11: SET TEMP_BOTTOM_WORKER =
TEMP_BOTTOM_WORKER -> BOTTOM
         [END OF LOOP] Step 12: RETURN START Step
13: EXIT
```

IV. RESULTS AND DISCUSSION

Our model was compared with the other existing models of similar kind, and it was found that Model B as described in [19] was more efficient than the other existing models in terms of creation of grids of very large sizes and storing and retrieval of data from the grid created. But after comparing Model B with our model (Model A), it was found that Model A has a much better time complexity for the construction of its N x N puzzle grid. Although Model B has used an efficient algorithm to fill the extra spaces of the puzzle grid, but the time taken by it to create the entire grid is very poor compared to Model A, especially when the grids are of large sizes (like 500 x 500).

Since both Model A and Model B were constructed using C++, we had included <chrono> and <ctime> header files for measuring the time taken by both the models for constructing the grid.

To compare both the models, we had used various combinations of grid sizes and the number of answer words added to construct the grid and the various construction times taken by both the models were noted. The answer words here refer to the solutions of the puzzle queries. The data collected was stored in a table, and few of its rows are presented in Table I. The entire table with all the collected data entries is made publicly available and the link for which could be found in the description of Table I.

For comparison, we have created four graphs based on the number of answer words added to the grid while its construction. The four graphs have the following number of answer words added to the grid respectively: 5, 10, 20 and 50. But as expected the behavior of the graph did not significantly change based on the number of answer words added to the grid. It concludes that the total construction time of the grid was independent of the number of answer words added to the grid. The fours graphs are shown in Fig. 10, 11, 12 and 13.

Table I: Comparing the construction times of Model A and Model B based on their grid sizes and the number of answer words added to the grid. (NOTE: The table shows only selected rows of the main table. The entire table could be found at https://goo.gl/gmFYhg)

Sr No	No. of Answer Words	Grid Size	Construction Time for Model A	Construction Time for Model B
1	5	10	1.25E-06	0.000378057
2	5	20	2.97E-06	0.000709776
3	5	30	9.68E-06	0.00123613
4	5	40	1.59E-05	0.00318762
5	5	50	2.60E-05	0.00221284
31	10	10	9.09E-07	0.000278666
32	10	20	2.96E-06	0.00498367
33	10	30	7.75E-06	0.0024282
34	10	40	2.46E-05	0.00242085
35	10	50	3.50E-05	0.0032083
61	20	10	1.16E-06	0.0630093
62	20	20	3.50E-06	0.00171616
63	20	30	1.23E-05	0.00638175
64	20	40	2.08E-05	0.0116467
65	20	50	2.62E-05	0.00873631
91	50	10	2.01E-06	NA
92	50	20	3.79E-06	0.00623795
93	50	30	8.94E-06	0.00534417
94	50	40	2.38E-05	0.0167668
95	50	50	3.01E-05	0.0384575

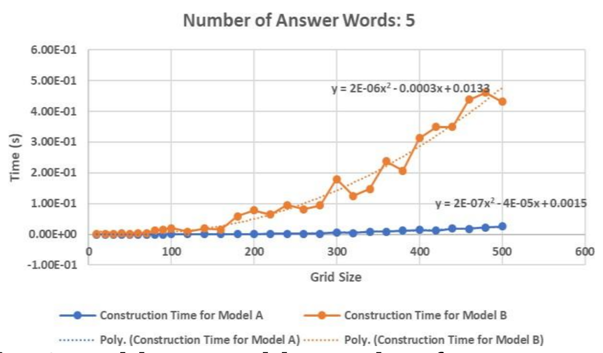

Fig. 10: Model A v/s Model B Number of Answer Words = 5

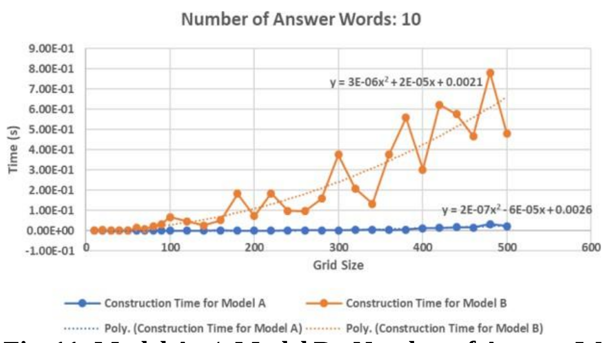

Fig. 11: Model A v/s Model B - Number of Answer Words = 10

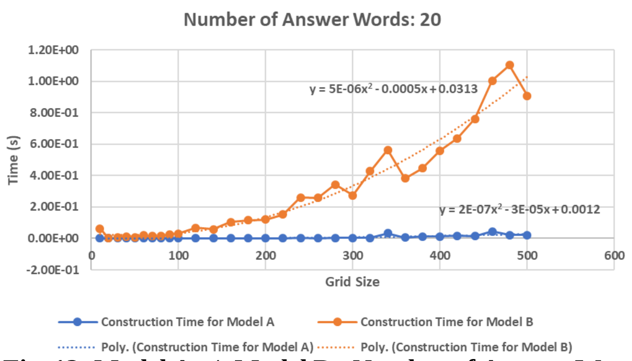

Fig. 12: Model A v/s Model B - Number of Answer Words = 20

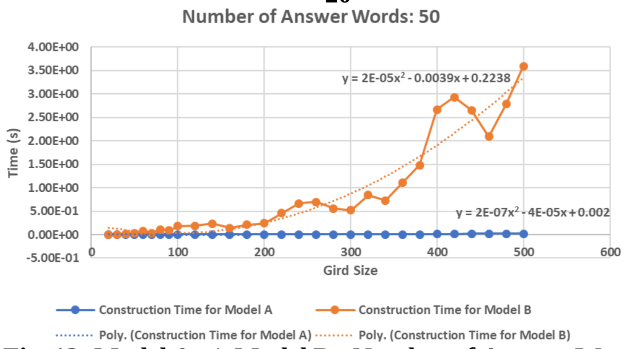

Fig. 13: Model A v/s Model B - Number of Answer Words = 50

V. CONCLUSION AND FUTURE WORK

The performance of any game is directly dependent on the efficiency of the construction of the game. The Word Search Puzzle consists of a rectangular or square grid of letters of various words. We have developed an algorithm to construct that grid efficiently using multi-linked lists data structure. The algorithm developed by us supersedes other models in its time complexity to create puzzle grids of large sizes (500 X 500 or equivalent and above). However, there is one drawback of our model. It does not work efficiently, in filling the extra unnecessary spaces of the grid with random alphabets. A real-time dataset was developed by us to compare the efficiency of our model with the others. In the future, we aim to overcome the limitation of our model, by creating one which efficiently fills the extra spaces of the grid with random alphabets.

REFERENCES

1. V. R. Kanagavalli, G. Maheeja, “A Study on the Usage of Data Structures in Information Retrieval”, National Conference on Innovations in Communication and Computing Technologies, Feb 2016.
2. Đorđe Stojisavljević, Eleonora Brtka, Vladimir Brtka, Ivana Berković, “Multi-Linked Lists – An Object-Oriented Approach”, ICIST 2015 5th International Conference on Information Society and Technology, 2015, pp. 391-396.
3. Li-Bing Wu, Jing Fan, Lei Nie, Bing-Yi Liu, “Tacked Link List - An Improved Linked List for Advance Resource Reservation”, 11th IFIP International Conference on Network and Parallel Computing (NPC), Ilan, Taiwan, Sep 2014, pp. 538-541.
4. Đorđe Stojisavljević, Eleonora Brtka, “Application of Multi-Linked Lists Technique for the Enhancement of Traditional Access to the Data”, Proceedings of the International Conference on Applied Internet and Information Technologies, Zrenjanin, Serbia, 2013, pp. 403-407.

5. Morin, Patrick, "Data Structures for Strings", chapter 7, March 2012.
6. Reema Thareja, "Data Structures in C", Oxford Higher Education, 2011.
7. Vossoughi Hossein, Zargar Marzieh, "Using Word-Search-Puzzle Games for Improving Vocabulary Knowledge of Iranian EFL Learners", Journal of Teaching English as a Foreign Language and Literature of Islamic Azad University of Iran, Winter 2009, Volume 1, Number 1, pp. 79-85.
8. Tanenbaum A., Augenstein M., Langsam Y., "Data Structures Using C and C++", PHI Learning, 2009.
9. Falley. P, "Categories of Data Structures", Journal of Computing Sciences in Colleges - Papers of the Fourteenth Annual CCSC Midwestern Conference and Papers of the Sixteenth Annual CCSC Rocky Mountain Conference, October 2007, Volume 23 Issue 1, pp. 147-153.
10. Kenneth A. Reek, "Pointers in C", Pearson Education, 2007.
11. Jeremy Lee Graybill, "Integrated Word-Search, Word Link, Trivia Puzzle and WordScramble", United States Patent Application Publication, 2007, Pub. No. US 20070267815A1.
12. Charles William Ditter, "Three-Dimensional Word-Search Puzzle and Methods for Making and Playing the Three-Dimensional Word-Search Puzzle", United States Patent Application Publication, 2005, Pub. No. US 20050253335A1.
13. Burston Jack, "Theoretical foundations of crossword puzzle usage in foreign language vocabulary acquisition", 2005.
14. Parlante N., "Linked List Basics", Document #103, Standford CS Education Library, 2001.
15. Parlante N., "Linked List Problems", Document #105, Standford CS Education Library, 2001.
16. Hakan Kjellerstrand, "JaCoP model", Available: http://www.hakank.org/jacop/CrossWord.java, accessed on 16/06/2019.
17. Michael Haephrati, "Creating a Crossword Generator", Available: https://www.codeproject.com/Articles/530853/Creating-a-crossword-generator, 2018, accessed on 16/06/2019.
18. Student at Bangor University, Defa1t, "Simple Word Search Game", Available: https://codereview.stackexchange.com/questions/88733/simple-word-s earch-game, 2015, accessed on 16/06/2019.
19. Steve Baker, "Word_Search-1.1", Available: http://www.gtoal.com/wordgames/wordsearch_baker/word_search-1.1/word_search.cxx, accessed on 16/06/2019.

AUTHORS PROFILE

Shivam Hareshkumar Patel, is currently an undergraduate senior at SRM Institute of Science and Technology, Chennai pursuing his Bachelor of Technology in Computer Science and Engineering. He is interested in Data Science and Analytics, especially its application to the field of Business Analytics.

A Jackulin Mahariba, is currently an Assistant Professor (Senior Grade) in the Department of Computer Science and Engineering at SRM Institute of Science and Technology. Scoring a gold medal, she has pursued her Masters of Technology from Anna University of Technology, Tirunelveli in the field of Remote Sensing. Her research includes Vehicular Networks, Emergency Management and Geographic Information System.