International Journal of Scientific Engineering and Science
Volume 6, Issue 1, pp. 121-126, 2022. *ISSN (Online): 2456-7361*

A Word Search Puzzle Construction Algorithm

Lefteris Moussiades[1]
[1]Department of Computer Science, IHU, Kavala, Eastern Macedonia & Thrace, +30

Abstract— *This paper presents a novel word-search puzzle construction algorithm. The proposed algorithm guarantees that it will place all the words of a set of words in the game board with the sole condition that there is at least one correct placement. Furthermore, it dramatically improves the construction average time compared to an exhaustive search algorithm. The algorithm may be used in various educational applications. Finally, it helps retain student interest in repetitive training with the same set of words by placing them in random positions.*

I. INTRODUCTION

Recently, game-based learning and relevant research have become increasingly popular. A valuable overview is given by [1], which ascertains a substantial potential for learning with games and simulations. Computer games for learning computer memory concepts in secondary education is more effective in promoting students' knowledge of computer memory concepts and more motivational than the non-gaming approach [2]. Also, [3] shows that students in secondary education who played a mobile history game gained significantly more knowledge than those who received regular project-based instruction on the historical topic of the Middle Ages. Furthermore, a collaborative game-based learning approach is shown by [4] to improve students' learning performance in science courses. Moreover, an empirical study on engagement, flow, and immersion in game-based learning shows that meeting with the game positively affects learning outcomes [5]. Robots are also widely used and interact with students while helping them learn various cognitive subjects [6, 7, and 8].

In this context, we have proposed a robot capable of helping foreign language learners to acquire the vocabulary they have to learn as part of their current study in the foreign language [9]. Among other functions, VT automatically adapts a set of suitable games to the content of the required vocabulary and suggests them to students according to their individual needs. Word search puzzles help learn vocabulary [10]. Therefore, one of the games that the proposed robot suggests to students is a word search puzzle.

The most critical challenge in developing a word search puzzle application is the puzzle construction itself. A construction algorithm should place the words of a given set randomly on the board so that no word exceeds the board limits and all word intersections occur on common letters. Note that the requirement for random word distribution is essential to ensure that words are hidden in different positions in repeating executions, so the student's interest is attracted to all executions with the same vocabulary. A straightforward solution to the abovementioned problem is to calculate all possible placements of the words in the board, form a list of the valid ones, and finally choose an arrangement randomly from the set of valid placements. However, the space of possible solutions is vast, resulting in a time-consuming algorithm that is practically unacceptable. In this paper, we propose a word-search puzzle construction algorithm, which we call WoSeCon (Word Search Construction), and it has the following characteristics:

- Generates a puzzle distributing the words randomly on the board
- If there is a solution, the algorithm guarantees that it will be found.
- It reduces the average time complexity by returning the first (randomly generated) valid placement that it finds; therefore, it does not need to calculate all possible placements.

The rest of the paper is structured as follows: Chapter 2 discusses existing word-search puzzle construction algorithms. In section 3, we discuss the construction complexity. Next, in section 4, we introduce concepts essential for the description of the construction algorithm. In part 5, we present the construction algorithm in detail, and in section 6, we discuss its performance. Finally, in section 7, conclusions and further development are discussed.

II. EXISTING CONSTRUCTION ALGORITHMS

The word search puzzle and the crossword have several similarities. The construction of a crossword may be more complicated than the construction of a word search puzzle as the former requires all adjacent characters to belong to a word, while the latter does not. However, it is easier to find literature related to crossword construction [11, 12, 13, and 14] than to word search puzzle construction.

A patent paper [15] presents an interactive word search puzzle and a construction algorithm. However, this algorithm does not produce random puzzles but instead aims to create the same puzzle after a certain time if the player has not performed a specific action required by the game. Moreover, the construction algorithm does not guarantee that if a solution exists, it will find it with time efficiency, or it will find it at all. More precisely, the algorithm selects the first word randomly from a list of words and locates it in the puzzle grid. Any subsequent word is selected in order and is placed if it overlaps with a previously located one. Words that do not overlap are skipped. After the list of words has reached the end, a second pass tries to locate the previously skipped words. If the second pass completes and there are still words that have not been located, the algorithm clears the puzzle grid and starts from the beginning by selecting a word randomly from the list of words. This algorithm identifies some "random" placements that meet the overlap conditions in the puzzle board. However, it does not systematically investigate the available "random" arrangements, so it does not guarantee finding a solution, if possible. Besides, since "random" placements are not investigated systematically, it is expected that indifferent attempts to place the words, placements that have already been tested and have been failed, will be repeated. In our view, this algorithm is sufficient for doing puzzles when the board space is enough for the number of words to place. In denser puzzles, it will cause delay problems such that often, they can be considered equivalent to failure to find a solution, although a solution exists.

Paper [16] is also a patent paper that includes constructing a 3-dimensional word search puzzle and a way of using it. The puzzle is built from a predefined set of phrases, including words, sentences, numbers, and thoughts expressed as rebuses. The construction of the three-dimensional word-search puzzle is based on creating two-dimensional sheets for each surface of a three-dimensional figure. However, although it is reported that the predefined phrases are placed on two-dimensional sheets, no additional information on the placement algorithm is given. Also, two-dimensional sheets do not define a strict area, as happens with the puzzle grid in our case. On the contrary, each two-dimensional sheet overlaps the surface of a three-dimensional object in such a way that a continuous space is created between the sheets covering adjacent surfaces. More generally, the placement of predefined phrases is ensured only by their predefined small number to the available space.

Another approach simply leaves out words that cannot be located with the first try [17].

Besides, Terzopoulos in [18] places some words firstly horizontally and the remaining ones vertically. Within a row or a column of the puzzle, only one word can be placed. It tries a limited number of times, specified by a parameter, to place a word randomly. If a suitable placement for the word would not be found, then the word is skipped, and the construction process continues with the next word

The better construction algorithm, to our knowledge, tries to place words in the puzzle grid, and when a word cannot be placed, it backtracks to the previously placed word and tries to relocate it [19]. However, when it backtracks, it chooses a random location from the space of available positions without taking into account locations that, although they are available, they have already been tried while trying to place the current word after backtracking. Thus, the space of available placements is not investigated systematically. As a result, the algorithm examines placements that have already been considered and found unsuitable.

Interestingly, two websites [20, 21] offer a web-based service for word search puzzle construction without, however, giving any further information about the construction algorithm.

III. THE CONSTRUCTION COMPLEXITY

Let us start by clarifying the construction problem. Given a set of words and a two-dimensional puzzle-grid, the problem is to find a random but valid placement of the words in the grid. A valid arrangement is a placement of the words in the grid where no word exceeds the grid limits, and all word intersections occur on common letters.

An obvious solution to this problem is calculating all possible placements, finding the valid ones, and selecting one randomly. Assume that the set of words has size *k*, and the grid size is *r x c*, where *r* represents the number of rows, and *c* represents the number of columns. Furthermore, assume that *d* represents the number of all possible directions in which words can be placed in the grid, e.g., vertical or horizontal. Considering that no word consists of only one letter, it becomes obvious that no word can start on the grid limits, i.e., in a grid entry on row r or in a grid entry on column c. Therefore, there are $n = (r-1) \times (c-1) \times d$ entries where a word could be started in the grid. The *k*-combinations of *n* are the different arrangements of *k*-positions of the total *n*-positions and are given by $n! / (n-k)! \times k!$. Furthermore, considering that in each of these *k*-tuples, the *k* words can be placed in *k!* different arrangements, we end up that the total solution space is $n! / (n-k)!$. This is a huge number. For example, for $k = 15$, $r = c = 20$, and $d = 2$, the possible placements are 6,52E+42. Such a solution is computationally prohibited. Therefore, instead of calculating all possible placements, our approach tries to find the first valid but random placement, and it returns it as soon as it finds it. Thus, the average construction time is reduced dramatically, and the problem becomes computationally efficient.

IV. CONCEPTS AND NOTATIONS

A Directed-Location is a triad consisting of a row identification number, a column identification number, and a direction identification number, which identifies the location as vertically or horizontally oriented. The placement of a word in the game board is represented by a Directed-Location, which identifies the row and column of the word's first letter and the word direction.

A Random-Locator keeps a list of available Directed-Locations and can select randomly one of them. Note that a Random-Locator shuffles the list of Directed-Locations immediately after the list creation. Therefore, the random selection is achieved by selecting sequentially the Directed-Locations kept in the list of the Random-Locator. Furthermore, a Random-Locator supports the following operations:

- add(Directed-Location). It adds a Directed-Location to the list of available Directed-Locations.
- remove(Directed-Location). It removes its argument from the list of Directed-Locations and returns it to the caller.
- get(Integer). It returns the Directed-Location at the index position represented by its integer argument. Note, when accessing Directed-Locations sequentially using the get function, we get the available locations in the game board in random order as a Random-Locator shuffles the list of Directed-Locations at its construction.
- minus(list of Directed-Locations). It returns a Random-Locator that considers as available Directed-Locations those that are available at the time the operation is executed, minus the Directed-Locations contained in the list that is given as argument.

A Word-Info keeps three types of information: First is the content, a string representing the word itself. Next is the placement, a Directed-Location that keeps the position of the first letter of this word in the game board and the direction of the content layout. If this word has not been placed, then placement is null. Note that the content, together with the placement, is sufficient to represent the content placement in the game board. Finally, a Word-Info keeps a list of Directed-Locations called tested locations, which we explain later in this chapter.

V. The WoSeCon Algorithm

U Given a list of Word-Info Objects and a game board, the basic idea is to position the words of the list in random locations of the game board, one by one. If, however, one word cannot be placed, then the algorithm steps backwards and repositions a previously positioned word. Therefore, WoSeCon operates in two modes: Backward and forward modes.

Initially, the algorithm operates in the forward mode. If the placement of the current word is successful, the operation mode remains forward. If the placement of the current word fails, then WoSeCon enters the backward mode. In the backward mode, the algorithm tries to reposition a previously positioned word. In the case of success, the operation mode changes again to the forward mode. Therefore, in an extreme case, all but one word may have been placed, and the algorithm may step backwards continuously until the first word in the list would be repositioned. If the first word in the list cannot be placed, then there is no way to place the words, and WoSeCon terminates with failure.

When the algorithm operates in the forward mode, Directed-Locations for the placement of the current word are given from a global Random-Locator, which we call global locator. The global locator keeps a list of unoccupied positions, initially consisting of the total Directed-Locations produced based on the game board. This list is initially shuffled, and the sequential selection gives the locations in random order.

However, when the algorithm operates in the backward mode, the global locator does not serve the purpose of a systematic search of suitable positions. Assume that the algorithm tries to place the word found at index i in the list of words and fails. Then, it backtracks to reposition word at index i-1. However, the word at index i-1 has already been positioned. Therefore, its current position should be marked as unoccupied and given back to the global locator that keeps the list of empty positions. Now, the word at index i-1 should take a position from the list of unoccupied ones minus the position that has been already tested and proved unsuitable as it does not leave space for placement of word i.

Moreover, after successful repositioning of word i-1, it is possible that the placement of word at i will fail again. In this case, the word at index i-1 should take a position from the list of unoccupied ones minus the two positions that have been already tested. The algorithm may backtrack several times from word at i to word at i-1. Therefore, each time when the algorithm backtracks from word i to word i-1, word i-1 should be placed in a position from the space of unoccupied positions minus the positions that have already been tested and have been proved unsuitable. These positions are kept in a list called the tested locations and kept in the corresponding Word-Info object. Note that when the algorithm steps backwards from the word at index i to the word at index i-1, the list of tested locations (of the word i) does not apply, as any of these positions may be appropriate for placement of the word i after the word i-1 has been repositioned. Therefore, the list of tested locations of the word i must be deleted when the algorithm steps backwards to reposition word i-1. In conclusion, when the algorithm operates in the backward mode to reposition word at index i-1, the tested locations of the word i should be deleted, and the tested locations of word i-1 should be updated. WoSeCon is given in algorithm 1.

Algorithm 1: The WoSeCon Algorithm

Input: *list of words* is the list of words to place on the game board, *board* represents the game board
Output: It updates the *board*

```
1   construct() {
2     int current word index = 0;
3     Word-Info current word = list of words.get(current word index);
4     operation mode = FORWARD;
5     while (true) {
6
7         if (locateOne(current word)) {
8         if (current word index == list of words.size()-1) break;
9         current word = list of words.get(++current word index);
10        operation mode = FORWARD;
11    } else {
12          if (current word index == 0) fail();
13          current word.deleteTested();
14          current word = list of words.get(--current word index);
15          operation mode = BACKWARD;
16        }
17    }
18
19    return Board (list of words);
20  }
```

In line 2, the variable current word index is initialized to 0. Variable current word index represents the index position in the list of words of the current word, i.e., the word that the algorithm is currently placing on the game board. In line 3, the current word, a Word-Info object, gets its value from the list of words.

The while-loop from line 5 to line 18 performs the actual placement of the words. In line 7, the algorithm tries to place the current word. We detail explain the placement of a word in the game board later in this section. In case of successful placement of the current word, the algorithm proceeds and checks if the current word that has just been placed, is the last one (line 8). If it is, all words have been placed successfully; therefore, we break the loop. If the current word is not the last one, then the next word in the list becomes the current one, and the operation mode is set to forward. Therefore, the construction will proceed to place the next word. If the placement of the current word fails (line 7), the control is transferred in line 12, where we check if the word that cannot be placed in is the first one in the list of words, in which case, the algorithm terminates with failure. If the first word cannot be placed, then it either does not fit the dimensions of the game board, or no suitable placement provides sufficient space for the placement of the next words. If the current word is not the first word in the list of words, then the algorithm deletes (line 13) the tested locations of the current word as we have already explained that this deletion is necessary. Next, the previous word in the list becomes the current word (line 14), and the operation sets to the backward mode (line 15). Therefore, the algorithm will proceed to reposition the previously positioned word. When the algorithm exits the while-loop, it updates the game board (line 19) and returns it. Recall that the list of words is a list containing Word-Info objects, each keeping the information related to the placement of the word on the board. Thus, the "return Board" function in line 19 only needs the list of words to update the placement of the words in the game board.

Now we explain function locateOne, which is presented as algorithm 2. As its name indicates, it tries to place one word, i.e., the current one, in the game board. It receives the current word as parameter and updates it with the information relevant to its placement in the game board. It also accesses the list of words and the global locator.

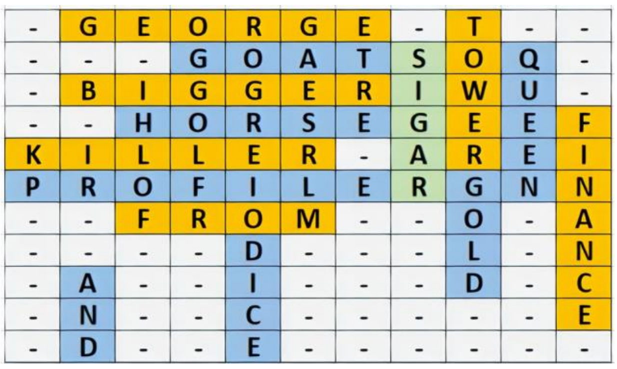
Algorithm 2: Function locateOne

Input: The list of words, the current word and the global locator
Output: true for successful placement of the *current word* and false otherwise. It also updates the *current word* with the information that is relevant to its placement in the game board

```
1  locateOne(current word) {
2
3    Random-Locator local locator;
4    if (operation mode==BACKWARD) {
5        Directed-Location dL = current word.getPlacement();
6        global locator.add(dL);
7        current word.moveLocationToTested();
8        local locator = global locator.minus(current word.getTested());
9    } else {
10       local locator = global locator;
11   }
12
13   int location index = 0;
14   while (location index < local locator.size()) {
15       Directed-Location suitable location = local locator.get(location index);
16       if (validPlacement(list of words, current word, suitable location)) {
17               global locator.remove(suitable location);
18               return true;
19       }
20       location index++;
21   }
22   return false;
23 }
```

In line 3, a Random-Locator named local locator is declared to be used later in the scope of the locateOne. In line 4, the operation mode is checked. If it is backward, in line 5, we assign to the variable dL the Directed-Location where the current word was placed. Next, in line 6, we give back to the space of unoccupied positions the dL. In line 7, function moveLocationToTested performs two tasks: First, it deletes the placement of the current word such that the current word is not considered that has been placed anymore; second, it updates the tested locations of the current word. Finally, in line 8, the local locator is prepared to give a Directed-Location from the space of unoccupied ones minus the tested locations of the current word. If the operation mode is forward (line 4), then the local locator becomes equal to the global locator, which means that all unoccupied locations are available. Next, locateOne searches the space of suitable positions to find one for the placement of the current word. In the while loop (lines 14 to 21), the appropriate Locator, which the local locator holds, returns the suitable positions (line 15) sequentially. The first suitable valid position is removed from the space of unoccupied positions (line 17), and locateOne returns true. Function validPlacement (line 16) checks whether or not the position returned by the local locator is valid. It considers as valid a placement if the content is placed such that it does not exceed the limits of the game board and does not overlap with any other word or it overlaps on a joint letter. Besides, the validPlacement updates the placement information of the current word. Suppose the while loop will terminate without a position to be found, which means that a valid placement for the current word has not been found, although all suitable placements have been tried. In that case, locateOne returns false to indicate failure of current word placement.

C++ code sources of the WoSeCon Algorithm can be found in github [22].



-	G	E	O	R	G	E	-	T	-	-
-	-	-	G	O	A	T	S	O	Q	-
-	B	I	G	G	E	R	I	W	U	-
-	-	H	O	R	S	E	G	E	E	F
K	I	L	L	E	R	-	A	R	E	I
P	R	O	F	I	L	E	R	G	N	N
-	-	F	R	O	M	-	-	O	-	A
-	-	-	-	D	-	-	-	L	-	N
-	A	-	-	I	-	-	-	D	-	C
-	N	-	-	C	-	-	-	-	-	E
-	D	-	-	E	-	-	-	-	-	-

Fig. 1. A dense puzzle constructed in milliseconds

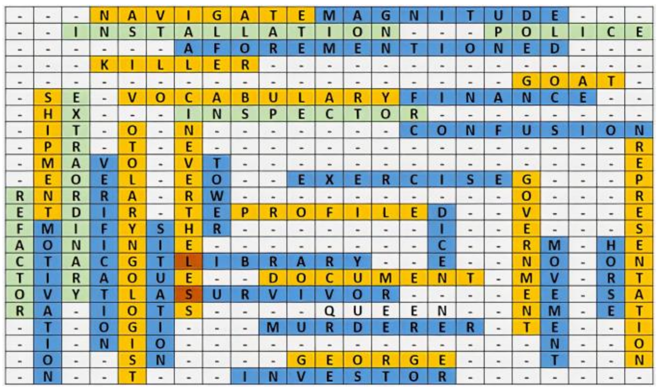
VI. Performance

The worst-case complexity of our approach is equivalent to the exhaustive search complexity. When there is only one valid placement of the words in the game board, then WoSeCon may try all possible placements until it ends up with the valid one. However, there is rarely one solution; therefore, WoSeCon will run faster as it needs to find only one valid placement. In practice, our algorithm constructs a puzzle in fractions of a second, even in the case of dense puzzles. Next, we present three puzzles, each constructed by a WoSeCon implementation in C++11, which has run on an i7 Intel with 16 GB RAM and 64-bit Windows 10 operating system. All three puzzles use English words. However, the current implementation of WOSeCon supports all languages that are written left-to-right, whereas it can be easily extended to support languages that are written right-to-left.

Figure 1 shows a dense puzzle. Cells containing a letter represent positions where a word letter has been placed. Cells indicated by the character '-' may contain any random character. This puzzle consists of 14 words placed on an 11x11 board and occupy about 58% of the total board space. The average construction time for 100 executions is about 0.004 seconds; the minimum time is about 0.001 seconds, and the maximum one is 0.03 seconds.

However, if we reduce the board dimensions to 10x11, the average time becomes 0.36 seconds, the minimum time is 0.005 seconds, and the maximum time is 31 seconds. Depending on the application, the construction time of 31 seconds may not be suitable. Therefore, for this set of words, boards smaller than 11x11 push the WoSeCon performance to its limits.

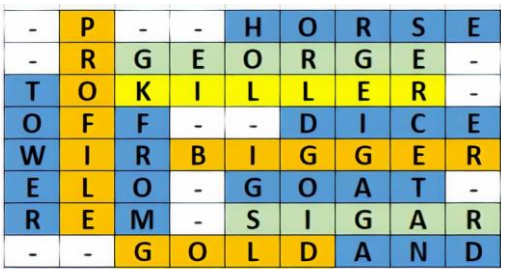
Figure 2 presents a larger puzzle. It consists of 23 lines and 23 columns, and 34 words have been placed. This set of words includes longer words in comparison with the puzzle of figure 1. The words here occupy about 54% of the entire puzzle board. The average construction time for 100 executions is about 0.093 centiseconds. The minimum construction time is 0.023 seconds, and the maximum construction time is about 3.7 seconds. These times are also considered acceptable. However, if we reduce the size of the board, then similarly to the first case, the maximum construction time becomes of the order of tens of seconds, which may render the algorithm performance unacceptable.



-	-	-	N	A	V	I	G	A	T	E	M	A	G	N	I	T	U	D	E	-	-	-
-	-	I	N	S	T	A	L	L	A	T	I	O	N	-	-	-	P	O	L	I	C	E
-	-	-	-	-	-	A	F	O	R	E	M	E	N	T	I	O	N	E	D	-	-	-
-	-	-	K	I	L	L	E	R	-	-	-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	G	O	A	T	-
-	S	E	-	V	O	C	A	B	U	L	A	R	Y	F	I	N	A	N	C	E	-	-
-	H	X	-	-	-	I	N	S	P	E	C	T	O	R	-	-	-	-	-	-	-	-
-	I	T	-	O	-	N	-	-	-	-	-	-	-	C	O	N	F	U	S	I	O	N
-	P	R	-	T	-	E	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R
-	M	A	V	O	-	V	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	E
-	E	O	E	L	-	E	O	-	-	E	X	E	R	C	I	S	E	G	-	-	-	P
R	N	R	R	A	-	R	W	-	-	-	-	-	-	-	-	-	-	O	-	-	-	R
E	T	D	I	R	-	T	E	P	R	O	F	I	L	E	D	-	-	V	-	-	-	E
F	M	I	F	Y	S	H	R	-	-	-	-	-	-	-	I	-	-	E	-	-	-	S
A	O	N	I	N	I	E	-	-	-	-	-	-	-	-	C	-	-	R	M	-	H	E
C	T	A	C	G	T	L	I	B	R	A	R	Y	-	-	E	-	-	N	O	-	O	N
T	I	R	A	O	U	E	-	-	D	O	C	U	M	E	N	T	-	M	V	-	R	T
O	V	Y	T	L	A	S	U	R	V	I	V	O	R	-	-	-	-	E	E	-	S	A
R	A	-	I	O	T	S	-	-	-	-	Q	U	E	E	N	-	-	N	M	-	E	T
-	T	-	O	G	I	-	-	-	M	U	R	D	E	R	E	R	-	T	E	-	-	I
-	I	-	N	I	O	-	-	-	-	-	-	-	-	-	-	-	-	-	N	-	-	O
-	O	-	-	S	N	-	-	-	-	G	E	O	R	G	E	-	-	-	T	-	-	N
-	N	-	-	T	-	-	-	I	N	V	E	S	T	O	R	-	-	-	-	-	-	-

Fig. 2. A 23x23 puzzle containing 34 long words

Figure 3 shows a dense puzzle consisting of 8 lines and 9 columns. In this puzzle, 12 words have been placed. This puzzle is relatively small, but it is quite dense as the words occupy about 80% of the total board space. The average construction time for 100 executions of WoSeCon was 0.10 seconds. The minimum construction time was 0.0012 seconds, and the maximum one was 4.9 seconds. In this case, the minimum, the maximum, and the average construction time are considered acceptable. The puzzle cannot be constructed in smaller boards; therefore, WoSeCon fails in smaller boards.



-	P	-	-	H	O	R	S	E
-	R	G	E	O	R	G	E	-
T	O	K	I	L	L	E	R	-
O	F	F	-	-	D	I	C	E
W	I	R	B	I	G	G	E	R
E	L	O	-	G	O	A	T	-
R	E	M	-	S	I	G	A	R
-	-	G	O	L	D	A	N	D

Fig. 3. A very dense puzzle occupying 80% of the board space

According to the analysis of the three cases above, we see that WoSeCon constructs dense puzzles in which words occupy more than 50% of the board at acceptable time rates. Performance issues are observed for denser puzzles, but this cannot be considered a problem as the typical puzzles we encounter in educational or other material are not so dense.

VII. Conclusions and Further Development

The word-search puzzle can be used as an educational game that allows students learning a foreign language to practice their vocabulary. This paper presents a novel and efficient word-search puzzle construction algorithm. Given that our algorithm constructs even dense word searches at acceptable time duration, it is understood that typical puzzles commonly used for entertainment or training are also constructed at acceptable times. Also, note that we present a solution where words are placed vertical or horizontal on the game board. However, other directions, e.g., the diagonal, can be easily added without affecting the algorithm. Next, we will deal with completing a word puzzle search application based on WoSeCon. In particular, we will design the user interface considering the appropriate pedagogical and learning principles. Also, we will add a variety of features that will facilitate vocabulary learning, e.g., the pronunciation of the word revealed by the student. Another line of research inspired by WoSeCon is the development of a general repositioning algorithm with constraints that may seem useful in a variety of problems, such as the automatic time table generation.

References

[1] S. de Freitas, "Learning in immersive worlds: a review of game-based learning," Bristol: Joint Information Systems Committee, 2006, Accessed: Sep. 29, 2019. [Online]. Available: http://www.jisc.ac.uk/media/documents/programmes/elearninginnovation/gamingr eport_v3.pdf

[2] M. Papastergiou, "Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation," Computers & Education, vol. 52, no. 1, pp. 1–12, Jan. 2009, doi: 10.1016/j.compedu.2008.06.004.

[3] J. Huizenga, W. Admiraal, S. Akkerman, and G. ten Dam, "Mobile game-based learning in secondary education: engagement, motivation

and learning in a mobile city game," Journal of Computer Assisted Learning, vol. 25, no. 4, pp. 332–344, 2009, doi: 10.1111/j.1365-2729.2009.00316.x.
[4] H.-Y. Sung and G.-J. Hwang, "A collaborative game-based learning approach to improving students' learning performance in science courses," Computers & Education, vol. 63, pp. 43–51, Apr. 2013, doi: 10.1016/j.compedu.2012.11.019.
[5] J. Hamari, D. Shernoff, E. Rowe, B. Coller, J. Asbell-Clarke, and T. Edwards, "Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning," Computers in Human Behavior, Aug. 2016, doi: 10.1016/j.chb.2015.07.045.
[6] F. Mondada et al., "Bringing Robotics to Formal Education: The Thymio Open-Source Hardware Robot," IEEE Robotics Automation Magazine, vol. 24, no. 1, pp. 77–85, Mar. 2017, doi: 10.1109/MRA.2016.2636372.
[7] S. H. Ivanov, "Will Robots Substitute Teachers?," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper, Jun. 2016. Accessed: Jun. 15, 2018. [Online]. Available: https://papers.ssrn.com/abstract=2801065
[8] E. Park and S. J. Kwon, "The adoption of teaching assistant robots: a technology acceptance model approach," Program, vol. 50, no. 4, pp. 354–366, Sep. 2016, doi: 10.1108/PROG-02-2016-0017.
[9] V. Hossein and Z. Marzieh, "Using word-search-puzzle games for improving vocabulary knowledge of Iranian EFL learners," vol. 1, no. 1, pp. 79–85, Jan. 2009.
[10] D. Cheng and N. Dhulekar, "Crossword Puzzle Generator," 2009.
[11] J. Engel, M. Holzer, O. Ruepp, and F. Sehnke, "On Computer Integrated Rationalized Crossword Puzzle Manufacturing," in Fun with Algorithms, 2012, pp. 131–141.
[12] J. Esteche, R. Romero, L. Chiruzzo, and A. Rosá, "Automatic Definition Extraction and Crossword Generation From Spanish News Text," CLEI Electronic Journal, vol. 20, no. 2, p. 6, Aug. 2017, doi: 10.19153/cleiej.20.2.6.
[13] L. J. Mazlack, "The use of applied probability in the computer construction of crossword puzzles," in 1973 IEEE Conference on Decision and Control including the 12th Symposium on Adaptive Processes, Dec. 1973, pp. 497–506. doi: 10.1109/CDC.1973.269214.
[14] M. G. Chan, "Interactive electronic puzzle game and a method for providing the same," Aug. 05, 2003 Accessed: Dec. 13, 2018. [Online]. Available: https://patents.google.com/patent/US6602133B2/en
[15] C. Ditter, "Three-dimensional word-search puzzle and methods for making and playing the three-dimensional word-search puzzle," Nov. 17, 2005 Accessed: Dec. 13, 2018. [Online]. Available: https://patents.google.com/patent/US20050253335A1/en
[16] "algorithm - Simple Word Search Game," Code Review Stack Exchange, Dec. 13, 2018. https://codereview.stackexchange.com/questions/88733/simple-word-search-game (accessed Dec. 13, 2018).
[17] S. Goumas, G. Terzopoulos, D. Tsompanoudi, and A. Iliopoulou, "Wordsearch, an Educational Game in Language Learning," Journal of Engineering Science and Technology Review, vol. 13, pp. 50–56, Feb. 2020, doi: 10.25103/jestr.131.07.
[18] J. Buck, "Buckblog: Generating Word Search Puzzles," The Buckblog, Dec. 13, 2018. https://weblog.jamisbuck.org/2015/9/26/generating-word-search-puzzles.html (accessed Dec. 13, 2018).
[19] "Make your own cipher puzzle," Cipher, Dec. 13, 2018. https://www.armoredpenguin.com/cipher/ (accessed Dec. 13, 2018).
[20] "Word Search Maker," The Word Search, Dec. 13, 2018. https://thewordsearch.com/maker/ (accessed Dec. 13, 2018).
[21] L. Moussiades, lmous/WoSeCon. 2020. Accessed: Dec. 19, 2020. [Online]. Available: https://github.com/lmous/WoSeCon