Application of Breadth-First Search and Exhaustive Search on "Pluszle"

Kevin Nathaniel Wijaya - 13517072[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]kevin.wijaya@students.itb.ac.id

***Abstract*— The paper explores the application of breadth-first search and exhaustive search on "Pluszle". The breadth-first search is based on graph searching and the exhaustive search is based on brute force algorithm. The first part of the text explains the puzzle involved, also the algorithms used. The next part shows the implementation of the breadth-first search and exhaustive search to finish a puzzle created by the "Pluszle" app.**

***Keywords*—Breadth-First Search, Pluszle, Exhaustive Search**

I. Introduction

Everything in this world is a puzzle. There will always be a problem, and people will look for the solution to solve it. That is just a puzzle, in a nutshell, although it might not be as simple as that. We now use the word puzzles to portray a short game that challenges the mind, makes you think. Our brain wants to solve problems, and this is one way we can stimulate the brain.

Puzzles come in different forms and sizes, such as the classic jigsaw puzzles, or even chess tactics. The involvement of math in puzzles are also no longer a mystery. Training math usually takes in the form of puzzles, and there is one game that does that exactly called "Pluszle", a puzzle challenging the arithmetic capabilities of additions.

With the advancement of computers and the computing system, algorithms and processing powers have made it possible to create solutions for problems as simple as reminding us of an event, to creating neural networks for Artificial Intelligence to make its way.

The author will explore the algorithms and the combinations which could solve a puzzle as quick and as efficient as possible, in this case the puzzle from the app "Pluszle". The author will specifically use the breadth-first search and exhaustive search to create the algorithm capable of finding the solution to the puzzle.

II. "Pluszle"

Advertised as the "fresh logic puzzle", the game "Pluszle" was released on 2018 by Huckleberry BV on both the iOS platform and the Android platform. This game revolves around the addition arithmetic method made into a puzzle. The puzzle, shaped as a square, has sums on the right side and bottom side. The goal is to highlight numbers which add up properly per row and column to the corresponding sums on the right and bottom lines. Once the squares are selectively highlighted and equates to the proper sums, the puzzle will be considered solved.

This game is also available in magazine form which can be purchased in certain locations. With its interactive and interesting design, "Pluszle" has accumulated an average rating of 4.2 out of 5 from 628 total ratings on the Apple App Store. [1]

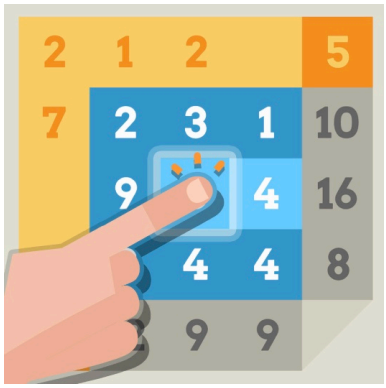

Fig. 1. Screenshot of Game "Pluszle"
(Source: *https://itunes.apple.com/us/app/pluszle-brain-logic-game/id1327839430?mt=8*)

As seen from Fig. 1, there is a sample of the puzzle, a 4-by-4 matrix of numbers, which could be highlighted to be selected as a solution. The grey rightmost row shows the sums needed for the respective numbers on the columns, and the grey bottommost column shows the sums needed for the respective numbers on the rows. This puts a new twist on simple addition puzzles.

III. Exhaustive Search

Exhaustive search is a variation of the brute force algorithm. The brute force algorithm is known as the most straightforward algorithm, the obvious, direct, and clear way to go. This algorithm has a wide range of application, is simple and relatively easy to be understood, tends to be accurate, and is usually the standard of most methods. This algorithm is used in all sorts of sorting methods, prime number testing,

multiplication of 2 matrices, etc. However, this algorithm is also known as a less efficient algorithm, is uncreative, and also relatively slow when compared to other algorithms. [2]

A variation of the brute force algorithm, the exhaustive search, is a solution finding technique for combinatorial problems. It is mostly used for finding solutions involving permutation, combinations, or subsets of a set. The exhaustive search is used in the travelling salesperson problem, the 1/0 knapsack problem, and much more. The search method is as follows:

1. An enumeration of all the possible combinations.
2. Evaluating each combination and keeping the best result.
3. Returning the best combination possible.

With the results from the exhaustive search, all possibilities are explored and evaluated to find the solution [2]. For the case of finding solutions for the "Pluszle" game, there needs to be some modifications done to the search. Instead of just keeping one best result, the search needs to keep all permutations which when added together evaluates to the sum. These permutations will later on be the next steps used in the breadth-first search.

The complexity of the exhaustive search that will be used in this paper is

$$O(n) = (n * n!) \quad (2)$$

because there are n comparisons done and permutations have to be done which have a complexity of n factorial.

IV. Breadth-First Search

Breadth-first search (BFS) is a type of graph search, where a graph is a set of vertices connected by edges. There are mainly two types of edges, an ordered pair which has directions and an unordered pair which is undirected (see Fig. 2). A graph could be represented in different forms, such as adjacency lists which stores the neighboring vertices, implicit graphs which uses "Zero" space, object-oriented variations in which each object has a list of neighbors, and incidence lists which stores the edges.

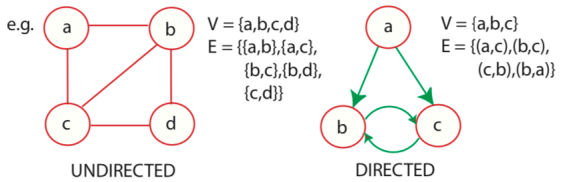

Fig. 2. Illustration of Graphs
(Source: *https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec13.pdf*)

A graph search uses this graph to, for example, find paths which lead to a desired vertex from another vertex. Applications of graph searching includes web crawling for searching on search engines, social networking for finding friends on social media, network broadcast routing, garbage collection, model checking, checking mathematical conjectures, and solving puzzles and games, which is the main focus of this paper.

The vertex on this search could also represent a state, which depicts the condition the program is in now or the solution achieved momentarily. This state would then be used for either comparison with the solution, or checking if the neighboring states have been visited, and so on. From this point on, vertices are going to be replaced by states for the purpose of less confusion, and the accuracy to the method going to be used later on in this paper.

A more specific type of a graph search is breadth-first search. How the breadth-first search works is starting from the start state, the next level(s) would continue to the states that are reachable from the previous state while ignoring the states from previous passing. [3]

The breadth-first search algorithm with the queue method in Python 3 is as follows [4]:

```
# Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices/states as not visited
        visited = [False] * (len(self.graph))
        # Create a queue for BFS
        queue = []
        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices/states of the dequeued
            # vertex/state s. If an adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

As seen from the algorithm above, the breadth-first search is going to first make a list of visited states initialized by the boolean "False", which will be changed to true once the state is visited. Next, a queue is created, and the starting state is added. All the neighboring states are added next and the process repeats until the queue is empty, meaning all the states are visited (see Fig. 3).

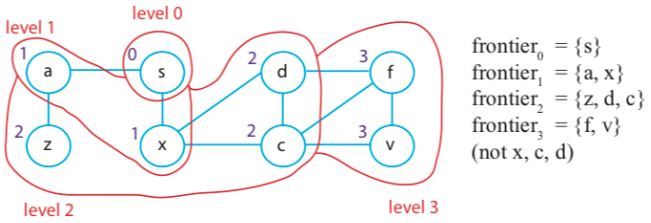

Fig. 3. Illustration of the Breadth-First Search Algorithm seen by the levels
(Source: *https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec13.pdf*)

This kind of implementation is classified as the static graph variation, in which the graph is previously defined and does not change. Another variation of the graph searching is the dynamic graph variation. In this variation, the graph is NOT present in the beginning of the process, rather it is created dynamically whilst executing the algorithm and finding the solution, which is the variant used for finding the solution of the "Pluszle" game. [5]

Before moving on, it is important to know the complexity of the breadth-first search, which is

$$O(n) = (V + E) \quad (1)$$

V being the vertices and E being the edges. This will later be used as a comparison.

V. Breadth-First Search with Exhaustive Search

As mentioned before, the breadth-first search will be used with the dynamic graph variation, in which the graph to be searched is not given. Like the puzzle, the only thing given is the problem, and it is up to us to solve it. Here is where the exhaustive search comes in handy. The beginning state of the breadth-first search will be the problem itself. The depth of this search will be as much as the rows of the square matrix. The neighboring states of the previous state will be determined and created by the exhaustive search method, which will give a list of permutations that when each single number in each permutation is added, will result to the sum on the rightmost side outside of the matrix, which is the targeted sum. These solutions will be highlighted, and each solution will be added to the queue in which the breadth-first search will continue to do. This will keep on going until each column's highlighted sum is equal to the rightmost row outside of the matrix. If that condition is fulfilled, the breadth-first search will check if the highlighted numbers or squares for each row when added is equal to the lowest column which has the targeted sum for each row. If this condition isn't met, then the state is not the solution and is discarded and no longer continued. However, if the condition is met, the search will stop as it has found the solution.

At every state, there will also be a checking of the sums of each column, which has to be lower than or equal to the targeted sum. If this is not fulfilled, the state would be immediately discarded because is considered no longer viable as a solution.

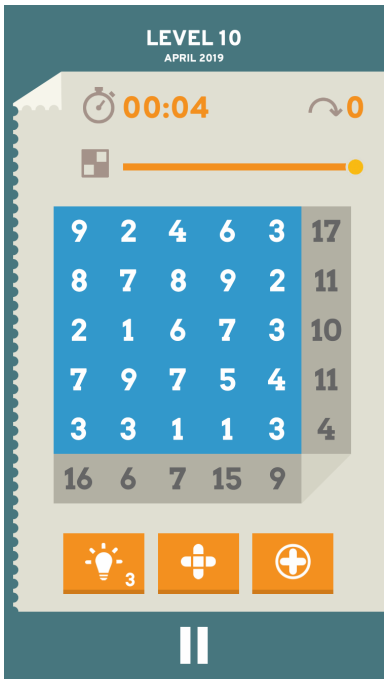

Fig. 4. Screenshot of "Pluszle" App on iOS
(Source: Screenshot on 26 April 2019)

For example, as seen on Fig. 4, we have a 5 by 5 matrix with the number 0-9 on each blue square, and an expected sum for each row and column on the right and lower side. This would be the beginning state or state of the breadth-first search helped by the exhaustive search.

The first level would be decided by the result of the exhaustive search done on the first row, with the target sum being 17. Coincidentally, the only permutation available for this [1, 1, 0, 1, 0], 1 being the highlighted squares and 0 being the unhighlighted squares, the content of the first index of the array being the number "9", the content of the second index being "2", and so forth. When the highlighted squares are added up, it results to 9 + 2 + 6 = 17, which is the targeted sum. All other permutations are of course checked, such as 9 + 2 + 4 + 3 = 18, but does not result in the targeted sum and therefore is not chosen as a candidate for the solution.

The breadth-first search adds this candidate to the queue, [[1, 1, 0, 1, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]], as only the first square has highlights. This is one example of the dynamic graph, as the states are dynamically added as the process continues. The next state is now popped from the queue and is processed (see Fig 5).

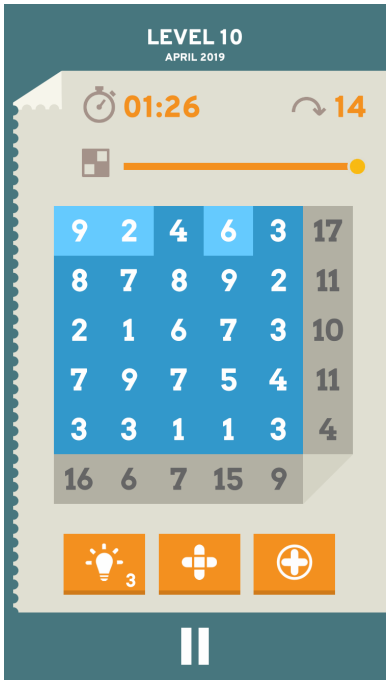

Fig. 5. Screenshot of "Pluszle" App on iOS after Row 1 Highlights
(Source: Screenshot on 26 April 2019)

This is now the current state as popped from the queue previously. Now the second row is processed by the exhaustive search, and this coincidentally is also the case of only 1 solution possible, that is 9 + 2 = 11, or [0, 0, 0, 1, 1]. This will now replace the second row for the state, making the current state [[1, 1, 0, 1, 0], [0, 0, 0, 1, 1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]] (see Fig. 6). This state will then be pushed into the queue for continuation.

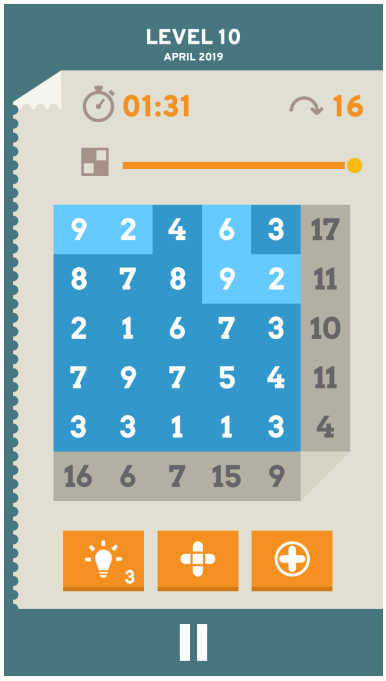

Fig. 6. Screenshot of "Pluszle" App on iOS after Row 1, 2 Highlights
(Source: Screenshot on 26 April 2019)

The state for processing will be popped and then be processed by the breadth-first search with the exhaustive search. On this third row, the exhaustive search found 3 possible solutions which fit, 2 + 1 + 7 = 1 + 6 + 3 = 7 + 3 = 10. Because of this, all 3 possible permutations are going to be pushed into the queue for further processing, with the queue having (left being the head):

[[1, 1, 0, 1, 0],	[[1, 1, 0, 1, 0],	[[1, 1, 0, 1, 0],
[0, 0, 0, 1, 1],	[0, 0, 0, 1, 1],	[0, 0, 0, 1, 1],
[1, 1, 0, 1, 0],	[0, 1, 1, 0, 1],	[0, 0, 0, 1, 1],
[0, 0, 0, 0, 0],	[0, 0, 0, 0, 0],	[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]	[0, 0, 0, 0, 0]]	[0, 0, 0, 0, 0]]

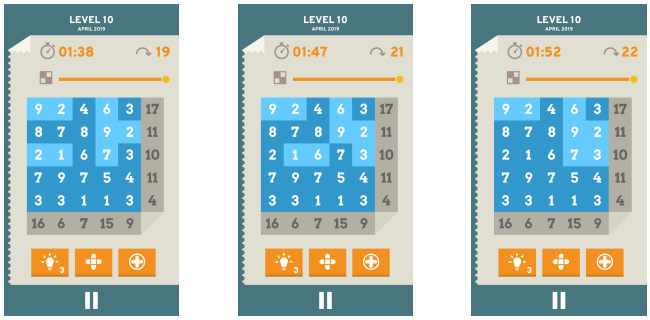

Fig. 7, 8, 9. Screenshot of "Pluszle" App on iOS after Row 1, 2, 3 Highlights
(Source: Screenshot on 26 April 2019)

As the search continues, popping the state from the queue, in this case [[1, 1, 0, 1, 0], [0, 0, 0, 1, 1], [1, 1, 0, 1, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]] (Fig. 7), the fourth row's is not either lower or equal to the targeted sum, which makes this state no longer viable and is discontinued. The search proceeds to the next state popped from the queue, [[1, 1, 0, 1, 0], [0, 0, 0, 1, 1], [0, 1, 1, 0, 1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]] (Fig. 8). As the rows' sum have not exceeded the targeted sum, this state is still considered viable and is continued for the neighbors. The exhaustive search continues to find candidate solutions for the column, 7 + 4 [1, 0, 0, 0, 1] = 7 + 4 [0, 0, 1, 0, 1] = 11. These two candidate solutions are then added to the back of the queue. The queue is again popped for the [[1, 1, 0, 1, 0], [0, 0, 0, 1, 1], [0, 0, 0, 1, 1], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]] (Fig. 9). However, this also has the same problem with the previous 2 state, in which the state's fourth row's sum exceeds the targeted sum. This state is then discarded and the search is left with this queue:

[[1, 1, 0, 1, 0],
[0, 0, 0, 1, 1],
[1, 1, 0, 1, 0],
[1, 0, 0, 0, 1],
[0, 0, 0, 0, 0]]

[[1, 1, 0, 1, 0],
[0, 0, 0, 1, 1],
[0, 1, 1, 0, 1],
[0, 0, 1, 0, 1],
[0, 0, 0, 0, 0]]

Fig. 10, 11. Screenshot of "Pluszle" App on iOS after Row 1, 2, 3, 4 Highlights
(Source: Screenshot on 26 April 2019)

Continuing the search, the next state from the queue, [[1, 1, 0, 1, 0], [0, 0, 0, 1, 1], [1, 1, 0, 1, 0], [1, 0, 0, 0, 1], [0, 0, 0, 0, 0]] (Fig. 10), each row's sum has not exceeded the targeted sum for each row. The exhaustive search continues to find permutations for the column, 3 + 1 [1, 0, 1, 0, 0] = 3 + 1 [1, 0 , 0, 1, 1] = 3 + 1 [0, 1, 1, 0, 0] = 3 + 1 [0, 1, 0, 1, 0] = 3 + 1 [0, 0, 1, 0, 0] = 3 + 1 [0, 0, 0, 1, 1] = 4. These states have reached the number-of-rows deep on the levels and therefore gets checked if the sum of each number on the rows are equal to the expected sum. For the first and second state, 9 + 7 + 3 > 16, so it is discarded. For the third state, the solution is found, with 9 + 7 = 16, 2 + 1 + 3 = 6, 6 + 1 = 7, 6 + 9 = 15, and 2 + 3 + 4 = 9. Therefore, the search comes to a halt with the state being the answer and the solution.

Fig. 12. Screenshot of "Pluszle" App on iOS: Finished!
(Source: Screenshot on 26 April 2019)

VI. Conclusion

Through the use of breadth-first search combined with exhaustive search, the complexity of the method is reduced quite significantly when compared to just brute force or exhaustive search. By using brute force or exhaustive search alone, the complexity would be $O(n * 2^n)$, because there would be checking of the values plus the permutations. By using breadth-first search alone, the complexity would be O(n * n!), because of the number of states available in every single iteration. By combining both exhaustive search and breadth-first search, the complexity was able to be reduced down by O(n * n), as shown by the results of the experiment. There was not a need to check all the states because of the results or states that the exhaustive search provided, and there was no need of backtracking or testing all single boxes because of the breadth-first search which checked every row. All in all, the use of breadth-first search joined with the exhaustive search resulted in a favorable outcome.

Acknowledgment

The author would like to thank first of all God as the author was able to finish writing this paper well. The author would also like to thank lecturer Dr. Ir. Rinaldi Munir, MT. from the Strategic Algorithm IF2211 class for his lectures and support. Also, the author would like to express his gratitude for his family and friends for their constant support.

REFERENCES

[1] "Pluszle: Brain Logic Game." *App Store*, Apple Inc. Web, itunes.apple.com/us/app/pluszle-brain-logic-game/id1327839430?mt=8. Accessed 25 Apr. 2019.

[2] Munir, Rinaldi. "Algoritma Brute Force." Program Studi Informatika, 2014. Web, informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Brute-Force-(2016).pdf . Accessed 25 Apr. 2019.

[3] "Lecture 13: Graphs I: Breadth First Search." MIT OpenCourseWare, MIT. Web, ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec13.pdf. Accessed 26 Apr. 2019.

[4] "Breadth First Serach or BFS for a Graph." *GeeksforGeeks*, GeeksforGeeks. Web, www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Accessed 26 Apr. 2019.

[5] Munir, Rinaldi. "Breadth/Depth First Search (BFS/DFS)." Program Studi Informatika, 2015. Web, informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/BFS-dan-DFS%20(2018).pdf. Accessed 25 Apr. 2019.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2019

Kevin Nathaniel Wijaya
13517072