Use of Backtracking Algorithm with Constraint Propagation for Solving Dot Connect Puzzle

Angelina Efrina Prahastaputri - 13523060
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: efrinaprahastaputri@gmail.com , 13523060@std.stei.itb.ac.id

***Abstract*—In order to solve Dot Connect puzzle, a player must connect all open dots on the board without the connections overlapping. This requires player to consider obstacles or constraints while maintaining to connect all the open dots. This paper explores the use of backtracking algorithm with constraint propagation for solving Dot Connect Puzzle. This paper will explore each aspect of backtracking algorithm and how backtracking algorithm works on generating the solution to the challenging Dot Connect Puzzle with constraint propagation. The result of this research aims to get a better understanding about backtracking algorithm and its advantages on solving puzzle such as Dot Connect Puzzle.**

Keywords—Backtracking Algorithm, Constraint Propagation, Dot Connect Puzzle, Solution

I. Introduction

In a world where different varieties of games exist with varying genres, ranging from simple and relaxing ones to complex and challenging ones, puzzle games are still incredibly popular, consistently maintaining high ranking among the top genres of games. Puzzle games are beloved for their compelling logic challenges, therefore building problem-solving skills and improving cognitive abilities of the players. They challenge the players to think critically, foresee consequences, and manage a complex set of rules of constraint to achieve the goal of the game. A particular category of the puzzle games is path-finding puzzle or popularly known as "connection" puzzle. One of which puzzle will be discussed in this paper.

The Dot Connect puzzle is just like the classic connect the dots game but without number or structures to guide the player. Starting from the initial dot, the player must connect all the dots on the board. Connections are made vertically, horizontally, and without overlapping until all the dots on the board are connected [6]. Besides the initial dot, there are also barriers that can limit connections between certain dots. While this puzzle certainly has simpler rules than other connect the dots game, solving it requires the player to have significant foresight and logical deduction, as a single early mistake can make completing the puzzle quite difficult because the player has to rebuild the path that follows the rule and satisfy the goal of this puzzle. That's why the complexity of this puzzle intrigues the author to use it as a study case about the implementation of certain algorithm strategy for automated solving in this paper.

The author chose the title "Use of Backtracking Algorithm with Constraint Propagation for Solving Dot Connect Puzzle" not only because it is one of the popular categories of puzzle game and is challenging to solve, but also because the puzzle's nature directly lends itself to a computational solution that can be achieved by trial-and-error approach. It is important to explore algorithm strategies that can navigate through vast number of possible paths to find the actual solution of the puzzle efficiently. This paper will explore the use of backtracking algorithm, one of algorithm strategies for exploring all possible candidates of the solution, with constraint propagation, a method to prune the search space and avoid dead ends intelligently. Through implementation and analysis of it for automated solving, the author hopes that this paper can provide a better understanding of the backtracking algorithm with constraint propagation and its powerful application in solving complex and constraint-based logic puzzles.

II. Theoretical Basis

A. Backtracking Algorithm

Backtracking algorithm can be viewed as: a phase inside *Depth First Search* algorithm, or a systematic and structured problem-solving technique [3]. In contrast to exhaustive search algorithms that explore and evaluate every single possible solution, backtracking algorithm works more efficiently. Whereas in the backtracking algorithm, only choices that lead to a solution are explored, choices that don't lead to a solution are no longer considered [3]. This algorithm is first introduced by D. H. Lehmer in 1950.

There are several common properties in a backtracking algorithm:

1. Solution space, all possible solutions of the problem, written as a set of vectors with n-tuple
2. Generator function, a function that generates a value x_k that is a component of the solution vector
3. Bounding function, a boolean function that returns true whenever a set of value leads to a solution and not violating any constraints [4]

Backtracking algorithm process can be visualized as a search within a state-space tree. All possible solutions of the problem are called the solution space [3]. The solution space is organized into a rooted tree structure [3]. Each node in this tree represents a specific state while each edge in this tree represents certain choices. All paths from root to leaf represent possible solutions and they create the solution space.

A backtracking algorithm works by recursively exploring all possible solutions to the problem [1]. The core of backtracking algorithm involves generating solution components step-by-step. The solution is sought by generating state nodes to create a path from root to a leaf [3]. This algorithm traverses the tree using a *Depth First Search* approach. It starts at root node and then explores as far as possible along each branch before backtracking. During this traversal, nodes that have been generated are called "live nodes" and the live node that is currently being expanded is called the "expand node" or E-node for short [3]. If the path currently being formed doesn't lead to a solution, then the E-node is "killed" and becomes a "dead node". This "killing process" uses the algorithm's bounding function so that the dead node can no longer be considered in the search process thereby pruning its child nodes. If the path currently being formed ends with a dead node, the search backtracks to its parent node to explore other alternatives and continue to generate other child nodes [3]. The search stops when it finally reaches the "goal node" or the solution to the problem.

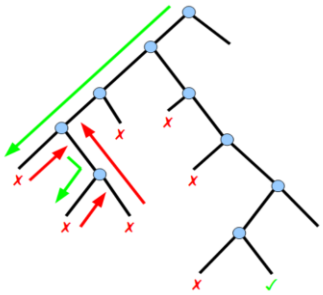


Figure 2.1 Backtracking Algorithm
Source: https://www.w3.org/2011/Talks/01-14-steven-phenotype/

```
HOW TO RETURN (route, state) path.to aim:
   IF state = aim:
      RETURN ("success", route with state)
   IF state in route OR NOT safe state:
      RETURN failure
   PUT route with state IN new.route
   FOR option IN possible.from state:
      PUT (new.route, state altered.for option) path.to aim IN result, route
      IF result = "success":
         RETURN ("success", route)
   RETURN ("failure", {})
```

Figure 2.2 Pseudocode for Recursive Backtracking Algorithm
Source: https://www.w3.org/2011/Talks/01-14-steven-phenotype/

B. *Constraint Propagation*

A key technique for efficiently solving problems in constraint programming is constraint propagation. Constraint propagation is the process of communicating the domain reduction of a decision variable to all the constraints that are stated over this variable [2]. When the possible set of values for one variable is reduced, this information is used by other related constraints to see if they can further reduce the domains of the variables they involve. This process can result in more domain reductions [2].

The purpose of constraint propagation is to reduce the domains of variables, potentially leading to the discovery of a solution or the identification of a failure [2]. This can drastically shrink the search space that a solver like backtracking algorithm needs to explore. The process continues iteratively until a stable state is reached where no more domains can be reduced. An empty domain during the initial constraint propagation means that the model has no solution [2]. This allows a solver like backtracking algorithm to identify whether the path is impossible or doesn't lead to the solution early without having to perform longer search.

C. *Dot Connect Puzzle*

The Dot Connect puzzle has various difficulties ranging from "Beginner" to "Ludicrous" which difficulty increases based on the size of the puzzle board and the number of barriers on the puzzle board. The puzzle board consists of a grid of dots that must all be connected by a single continuous line from the initial dot. The puzzle is solved when all open dots on the puzzle board are connected, no specific ordering or correct path is required. Connections are made vertically or horizontally, and without overlapping [6]. The puzzle board also contains barriers or blocked cells which the path cannot pass through.

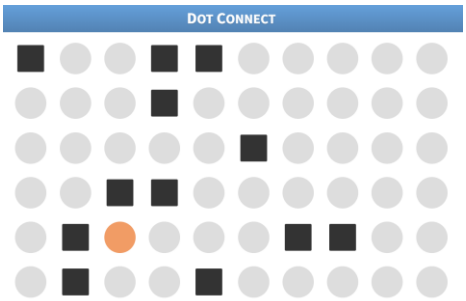


Figure 2.3 Dot Connect Puzzle Configuration
Source: https://api.razzlepuzzles.com/dot_connect

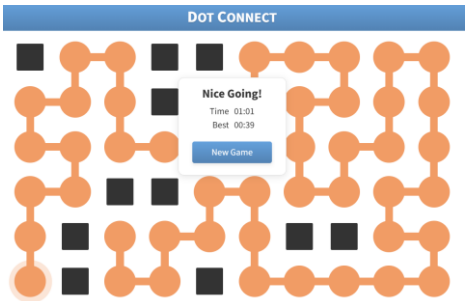


Figure 2.4 Solved Dot Connect Puzzle
Source: https://api.razzlepuzzles.com/dot_connect

Dot Connect puzzle can be classified as a Constraint Satisfaction Problem (CSP). The problem can be represented as finding a valid sequence of moves, where the solution is a vector of positions $P = (p_1, p_2, \ldots, p_n)$ with n being the total number of dots on the puzzle board. The core constraints that this sequence must satisfy are:

1. Each position p_i in the sequence must be adjacent to the previous position p_{i-1}
2. No position can appear more than once in the sequence because the line that connects the dots cannot overlap with each other
3. The path must stay within the puzzle board boundaries and avoid any predefined barriers
4. The final path must contain every single open dot on the puzzle board

A challenging aspect in solving Dot Connect puzzle is that a locally valid move may not lead to a global solution. A move that satisfies all immediate constraints can unintentionally "trap" the line, cutting off other regions of the puzzle board hence making it impossible to connect the remaining dots. This requires a strategy that can foresee or recover from such dead ends.

D. Backtracking Algorithm with Constraint Propagation in Dot Connect Puzzle

Backtracking is one of the core algorithms to solve Dot Connect Puzzle since it fits the criteria of a Constraint Satisfaction Problem. Backtracking algorithm is used for solving Dot Connect Puzzle by recursively finding all possible solutions that satisfy the constraint that has been stated previously. This algorithm explores all possible paths and when a path violates a constraint or hits a dead end, it backtracks to the last decision point to try alternative path. By using constraint propagation that prune invalid paths early, the search process is made efficient.

First, the puzzle board can be represented as a 2D array or a matrix where each cell has one of these following states: "empty", "barrier", or "visited". The solution is an ordered sequence of coordinates or vector of positions $P = (p_1, p_2, \ldots, p_n)$ with n being the total number of dots on the puzzle board and p_i is the coordinate of the i-th dot in the path.

The core of solving Dot Connect puzzle using backtracking algorithm is the recursive process. The first step in the recursive call is to check the termination condition. A complete solution is found when the length of the current path is equal to the total number of open dots on the puzzle board. If this condition is met, then the algorithm has found a valid path and can terminate the process. If not, the algorithm iterates through all adjacent neighbors from the current dot's position. For each neighbor, bounding function is applied to check if the move is valid or not. The bounding function directly implements the puzzle's constraint that has been stated previously. If a neighbor is deemed valid by the bounding function, the algorithm commits to the move, adding the chosen neighbor to the current path and updating its state to "visited". Then, a recursive call is made with the new current path and new current position (the chosen neighbor position). If this call returns true, it means that a complete solution is found and this success is passed up the call stack. If this call returns false, it means that the path from the chosen neighbor led to a dead end. The algorithm then backtracks or undoes the move by removing the neighbor from the current path and changing its state back to "empty" then continues to iterate to try the next neighbor. If for all neighbors have been tried and there is no path found, it means the current position itself is a dead end and the algorithm will bactrack to the previous level.

Constraint propagation enhances the search process of the backtracking algorithm by using logic to make deductions that prune the entire branches of the search tree before they are even explored. If making a move from, let's say position A to position B, creates a closed-off region of unvisited dots that are now disconnected from the rest of the puzzle, that move is deemed invalid. This is because the single continuous line wouldn't be able to enter and fill this region. By applying this propagation rule, the algorithm doesn't just check the validity of its immediate next step but also looks ahead at the consequences of that step by pruning paths that are doomed to fail much earlier, thereby significantly improving performance.

III. IMPLEMENTATION

The author focuses on the implementation of backtracking algorithm which has been stated previously. Firstly, we will look at the way Dot Connect puzzle is represented in the implementation. Besides the main algorithm, there are three main building components: Point, PuzzleBoard, and DotConnect (or the main program).

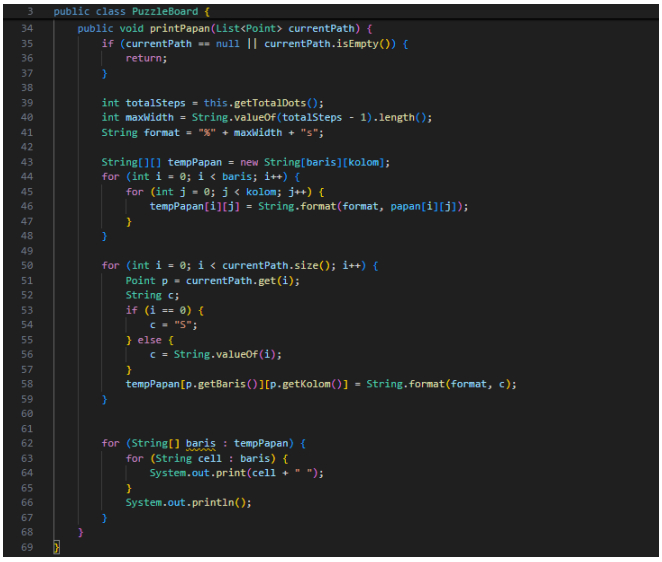
```java
import java.util.Objects;

public class Point {
    private final int baris;
    private final int kolom;

    // constructor
    public Point(int baris, int kolom) {
        this.baris = baris;
        this.kolom = kolom;
    }

    // getter
    public int getBaris() {
        return baris;
    }
    public int getKolom() {
        return kolom;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Point)) return false;
        Point other = (Point) obj;
        return this.baris == other.baris && this.kolom == other.kolom;
    }

    @Override
    public int hashCode() {
        return Objects.hash(baris, kolom);
    }

    @Override
    public String toString() {
        return "Point{ " + "baris=" + baris + ", kolom=" + kolom + " }";
    }
}
```

Figure 3.1 Point.java

Source: https://github.com/angelinaefrina/MakalahStima_13523060

The object Point represents each cell of the puzzle board and has attributes: row and column that shows the position or coordinates of said cell.

(a)



(b)

Figure 3.2 PuzzleBoard.java

Source:
https://github.com/angelinaefrina/MakalahStima_13523060

The object PuzzleBoard represents the puzzle board od Dot Connect Puzzle and has attributes: row size, column size, 2D array that contains the cells, total open dots on the puzzle board, and the starting/initial dot position. The function isDalamPapan determines whether or not a dot's position is valid (not outside the puzzle board), the function isBarrier checks whether a dot is a barrier which in this implementation a barrier is represented as "X", and the function printPapan prints the state of current puzzle board with the current path that are being formed or has been formed.

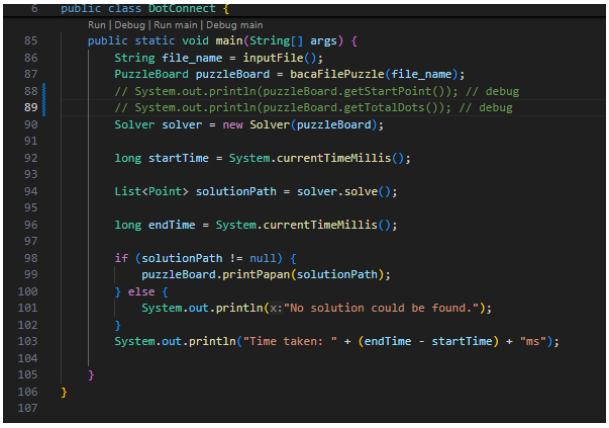


Figure 3.3 DotConnect.java

Source:
https://github.com/angelinaefrina/MakalahStima_13523060

The DotConnect class in this implementation is where the main program runs. Firstly, it will ask the user to input the file path of a .txt file that contains the information of the puzzle board, then it will read the puzzle board and create a new PuzzleBoard object out of it. Afterwards, it will call the solver function and if a solution is found, it will output the solution.

Next, we will look at the main backtracking algorithm that consists of three major components: constraint propagation, bounding function, and the solver (contains generator function and solution space).

Figure 3.4 Constraint Propagation in Solver.java

Source:
https://github.com/angelinaefrina/MakalahStima_13523060

The constraint propagation used for enhancing the automated solving for Dot Connect puzzle is implemented in the isTrappedRegionCreated function. It restricts any move to create trapped region which makes the line can't establish any connection with dots that are in the trapped region. Since the puzzle requires a single continuous line that connects all the dots, any move that create such trapped regions is guaranteed to lead to failure.

```
public class Solver {

    // bounding function: mengecek apakah gerakan valid/tidak melanggar constraints
    private boolean isMoveValid(Point p){
        if (!papan.isDalamPapan(p)) return false; // jika titik tidak dalam papan
        if (papan.isBarrier(p)) return false; // jika titik adalah barrier
        if (visited[p.getBaris()][p.getKolom()]) return false; // jika titik sudah dikunjungi

        // constraint propagation
        if (path.size() + 1 >= papan.getTotalDots()) {
            return true;
        }
        if (isTrappedRegionCreated(p)) {
            return false;
        }

        return true;
    }
```

Figure 3.4 Bounding Function in Solver.java

Source:
https://github.com/angelinaefrina/MakalahStima_13523060

The bounding function is implemented in the isMoveValid function that checks if the move violates any constraints or not. The constraints: is the cell outside the puzzle board, is the cell a barrier, is the cell has been visited (to avoid the path from overlapping), and is the cell violates the constraint propagation.

```
public class Solver {

    private boolean solveRecursively(Point currentPosition) {
        visited[currentPosition.getBaris()][currentPosition.getKolom()] = true;
        path.add(currentPosition);

        if (path.size() == papan.getTotalDots()) {
            return true; // complete path found
        }

        for (int i = 0; i < 4; i++) {
            Point nextPosition = new Point(currentPosition.getBaris() + rowDir[i], currentPosition.getKolom() + colDir[i]);
            if (isMoveValid(nextPosition)) {
                if (solveRecursively(nextPosition)) {
                    return true;
                }

            }
        }

        // backtrack
        visited[currentPosition.getBaris()][currentPosition.getKolom()] = false;
        path.remove(path.size() - 1);
        return false; // no valid path found from this position
    }

    public List<Point> solve() {
        Point startPoint = papan.getStartPoint();
        if (startPoint == null) {
            throw new IllegalStateException(s:"Titik awal tidak ditemukan pada papan.");
        }

        if (solveRecursively(startPoint)) {
            return path;
        }

        return null;
    }
}
```

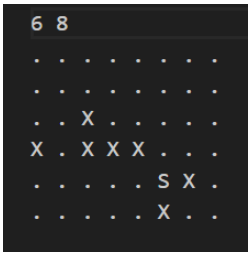
Figure 3.5 Backtracking Algorithm in Solver.java

Source:
https://github.com/angelinaefrina/MakalahStima_13523060

The main backtracking algorithm firstly will add the current dot into the solution path; this means that the starting point will always be the first dot in the solution sequence. It will then check the current dot's neighbors (generator function) iteratively. If a neighbor is deemed valid by the bounding function, it will continue to find other dots that can be added to the solution sequence recursively. If no valid path found from the current dot, the algorithm will backtrack to the previous level and try for the next neighbor. If the length of the solution sequence equals the total open dots on the puzzle board, then a complete solution is found.

IV. Analysis

The author conducts a test to see if the implemented backtracking algorithm with constraint propagation able to solve Dot Connect puzzle efficiently. First, we will see the difference that constraint propagation makes on the backtracking algorithm using the test case in Figure 4.1.

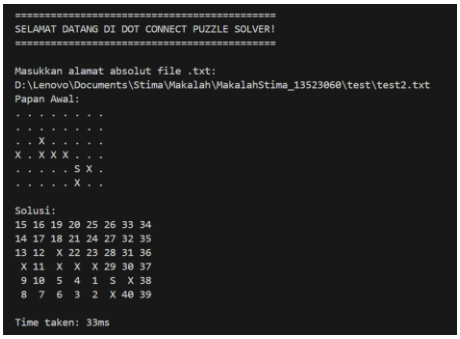


```
6 8
. . . . . .
. . . . . .
. . X . . .
X . X X . . .
. . . . . S X .
. . . . . X . .
```

Figure 4.1 Test Case 1

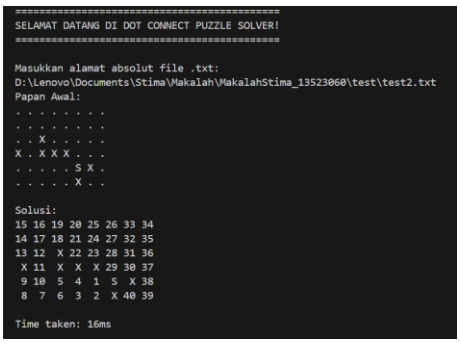
Source: Author's Personal Archive

The test case in Figure 4.1 consists of two components: the first line consecutively contains the row size and the column size of the puzzle board, followed by the puzzle board itself. The "." represents open dots on the puzzle board that must be connected, the "X" represents barriers, and the "S" represents the starting/initial dot.



```
========================================
SELAMAT DATANG DI DOT CONNECT PUZZLE SOLVER!
========================================

Masukkan alamat absolut file .txt:
D:\Lenovo\Documents\Stima\Makalah\MakalahStima_13523060\test\test2.txt
Papan Awal:
. . . . . . . .
. . . . . . . .
. . X . . . . .
X . X X . . .
. . . . . S X .
. . . . . X . .

Solusi:
15 16 19 20 25 26 33 34
14 17 18 21 24 27 32 35
13 12  X 22 23 28 31 36
 X 11  X  X  X 29 30 37
 9 10  5  4  1  S  X 38
 8  7  6  3  2  X 40 39

Time taken: 33ms
```

(a) Without Constraint Propagation



```
========================================
SELAMAT DATANG DI DOT CONNECT PUZZLE SOLVER!
========================================

Masukkan alamat absolut file .txt:
D:\Lenovo\Documents\Stima\Makalah\MakalahStima_13523060\test\test2.txt
Papan Awal:
. . . . . . . .
. . . . . . . .
. . X . . . . .
X . X X . . .
. . . . . S X .
. . . . . X . .

Solusi:
15 16 19 20 25 26 33 34
14 17 18 21 24 27 32 35
13 12  X 22 23 28 31 36
 X 11  X  X  X 29 30 37
 9 10  5  4  1  S  X 38
 8  7  6  3  2  X 40 39

Time taken: 16ms
```

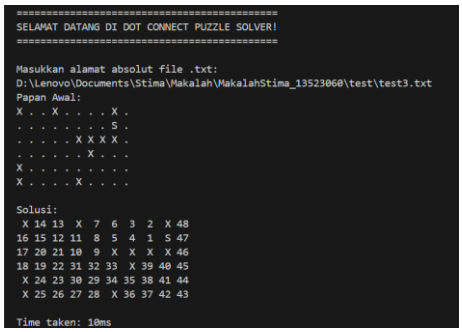
(b) With Constraint Propagation

Figure 4.2 Difference Between Searching Time for Backtracking Algorithm with and without Constraint Propagation

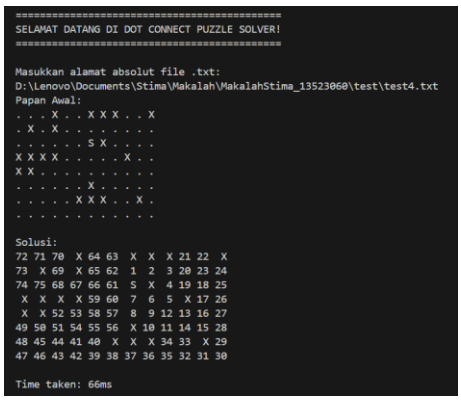
Source: Author's Private Archive

As we see in Figure 4.2, the backtracking algorithm successfully found a complete solution given the test case in Figure 4.1. However, there is a searching time difference between Figure 4.2 (a) and Figure 4.2 (b) and it appears in Figure 4.2 (b) that the backtracking algorithm performance is better than Figure 4.2 (a) with a significant 17 millisecond faster. This proves that constraint propagation indeed enhances the performance of the backtracking algorithm.

Second, we will see how the implementation handles much more complex puzzle configurations by increasing the size of the puzzle board.

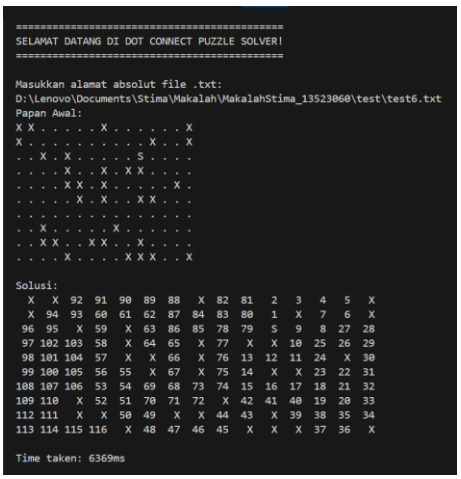


(a) 6 x 10 Puzzle Board



(b) 8 x 12 Puzzle Board

(c) 10 x 12 Puzzle Board



(d) 10 x 15 Puzzle Board

Figure 4.3 Difference Between Searching Time for Various Size of Puzzle Board

Source: Author's Private Archive

As we see in Figure 4.3, the backtracking algorithm successfully found a complete solution given the test case with various sizes of puzzle board. For bigger puzzle board size, the algorithm took longer to search for the solution path. The reason behind this lies in the fundamental nature of the problem and the search algorithm itself.

When the puzzle board's size increases, the number of dots that must be visited also increases. This causes the depth and the potential number of branches in the state-space tree to grow exponentially. For a board with n dots, the theoretical number of paths to explore can be in the order of b^n where b is the branching factor or the number of available moves at each step, which in this puzzle is at most 4. Increases in number of open dots doesn't simply increase the workload by some percentage, rather by many orders of magnitude, often referred to as "combinatorial explosion". Combinatorial explosion occurs when a huge number of possible combinations are created by increasing the number of entities which can be combined [5]. This is the main reason why the search process takes significantly longer for larger puzzle board.

The Dot Connect puzzle is a variation of the Hamiltonian Path Problem, thus is classified as NP-complete. It means, there is no known algorithm that can solve every instance of this problem in polynomial time. The worst-case time complexity for the backtracking algorithm that implemented is exponential. The complexity is formally expressed as $O(b^n)$ where n is the number of dots to visit, and b is the branching factor which in this puzzle is at most 4.

This backtracking algorithm can perform better thanks to the constraint propagation that is implemented. It is able to identify impossible solutions early by pruning the state-space tree, so the algorithm doesn't waste time exploring other paths that are guaranteed to fail. However, even with this intelligent pruning, the fundamental nature of the problem remains exponential which is why the search time still increases significantly with larger puzzle sizes.

V. Conclusion

This paper discussed how backtracking algorithm with constraint propagation can be efficiently used in solving Dot Connect puzzle. The implementation successfully proved that backtracking algorithm capable of finding complete solutions for various puzzle configurations and constraint propagation capable of enhancing the backtracking algorithm performance. The backtracking algorithm systematically explores the state-space tree recursively, extending the path from one dot to its next, eventually finding the solution path. The implementation of constraint propagation "trapped region rule" elevates the backtracking algorithm by actively identifying and pruning branches of the search tree that would lead to impossible paths.

While the fundamental worst-case time complexity remains exponential due to the Hamiltonian Path Problem, this project shows that intelligent heuristics can make the problem tractable for a wide range of complex instances. The algorithm's ability to solve progressively larger puzzles with increased time highlights the direct correlation between the puzzle's size and its computational complexity

Future enhancements could involve exploring more advanced heuristics and other constraint propagations. Additionally, comparing this algorithm's performance with other algorithms could provide valuable insight into broader applications of these algorithms.

Video Link at Youtube

https://youtu.be/2W2yAGzTjns?si=EVCuhEaTv1eEEn_f

Acknowledgment

The author would like to express their gratitude to several parties that helped the making of this paper. First and foremost, sincere thanks to God for guiding the author through the entire process of making this paper from learning, researching, and writing, until eventually this paper is complete. The author also acknowledges the immense support and guidance from the lecturer of IF2211 Algorithm Strategy, Mrs. Nur Ulfa Maulidevi and Mr. Rinaldi Munir, that has significantly helped the author enrich their knowledge. Special thanks also to the author's family, friends, and all the ITB Informatics students for the unwavering support throughout the entire semester.

Through this paper, the author hopes it can bring more knowledge for the author and for the readers on better understanding about the use of backtracking algorithm with constraint propagation for solving Dot Connect Puzzle.

References

[1] GeeksforGeeks. (2024). *Backtracking Algorithms*. https://www.geeksforgeeks.org/dsa/backtracking-algorithms/ accessed on June 24, 2025

[2] IBM. (2023). *Constraint propagation*. Dalam *IBM CP Optimizer 22.1.0*. https://www.ibm.com/docs/en/icos/22.1.0?topic=optimizer-constraint-propagation accessed on June 24, 2025

[3] Munir, Rinaldi. (2024). "IF2211 Strategi Algoritma – Semester II Tahun 2024/2025". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/stmik.htm accessed on June 2, 2025.

[4] Pradipta, Nayotama. (2022). "Implementation of Backtracking Algorithm in Minesweeper". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Makalah/Makalah-IF2211-Stima-2022-K2%20(30).pdf accesed on June 2, 2025.

[5] Principia Cybernetica Web. (2010). *Combinatorial Explosion*. https://web.archive.org/web/20100806122506/http://pespmc1.vub.ac.be/ASC/COMBIN_EXPLO.html accessed on June 24, 2024.

[6] Razzle Puzzles. (n.d.). *Dot Connect*. https://api.razzlepuzzles.com/dot_connect accessed on June 22, 2025.

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, 24 Juni 2025

Angelina Efrina Prahastaputri 13523060