Solving the Word-based Puzzle Game Smartle

Azmi M. Bazeid - 13522109
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
Email: azmibazeid@gmail.com

***Abstract*— This paper explores the application of brute-force/backtracking and greedy algorithms to solve Smartle, a word-based puzzle game requiring the formation of valid five-letter words. The author presents a program designed to "crack" the game, efficiently identifying solutions for any given letter grid. The brute-force/backtracking approach systematically explores all letter combinations, guaranteeing a solution. In contrast, the greedy algorithm prioritizes high-scoring letter placements, aiming for an optimized solution path. The author analyzes the effectiveness and trade-offs of the approach.**

Keywords—backtracking, bruteforce, greedy, puzzle, word-based

I. INTRODUCTION

Smartle is a captivating word puzzle that challenges your vocabulary and strategic thinking. Imagine a 5x5 grid, where each square holds a letter. Your task is to rearrange these letters by swapping any two tiles by dragging and dropping any two tiles, regardless of their position. However, the true objective lies beyond simple rearranging. You must craft valid five-letter English words across all five rows of the grid, striving for the most efficient solution with the least number of swaps.

This seemingly straightforward premise quickly transforms into a brain-teasing exercise. You'll need to juggle potential word combinations while strategically planning swaps to achieve the optimal solution. Each move can unlock new possibilities, but it can also disrupt previously formed words. Smartle becomes a dance between foresight and adaptation, demanding both a rich vocabulary and the ability to think several moves ahead.

Human players naturally develop strategies when tackling Smartle. One common tactic involves focusing on the least frequent letters in the grid early on. These characters (like J, Q, and X) offer fewer word formation possibilities, so addressing them first can open up more options later. However, the true challenge often arises when forming the final word. With four rows filled, the remaining letters might not readily rearrange into a valid five-letter term. This scenario can leave players feeling stuck, searching for a solution that may not exist.

This is where the beauty of an algorithmic approach lies. By incorporating these human strategies and difficulties, we can create a program that mimics the thought process behind Smartle. The algorithm can prioritize using less common characters early, similar to a human player. Additionally, it can analyze the remaining letters after forming the first four rows, replicating the "stuck" scenario and exploring alternative arrangements to find a valid solution. This integration of human challenges translates into a more robust and adaptable program capable of tackling various Smartle puzzles.

This paper delves into efficient solution-finding methods for Smartle, a word puzzle game that challenges players to form valid five-letter words. We will compare and analyze different algorithms to determine the most effective approach. Fortunately, several key characteristics of the game can be exploited to expedite the search for good solutions. These include the finite number of possible five-letter words, the dynamically changing pool of available letters after each swap, and the inherent structure of the 5x5 grid. By leveraging these factors, we aim to develop an algorithm that can efficiently navigate the search space and identify high-quality solutions in a timely manner.

II. THEORY

A. Brute-force Search

Brute-force search, also known as exhaustive search, is a straightforward problem-solving technique applicable to various scenarios. It involves a systematic evaluation of all possible solutions within a defined problem space.

Key characteristics of brute-force search:

- **Guaranteed Solution**: If a solution exists within the defined space, brute-force will find it by examining all possibilities.

- **Computational Cost**: The time and resources required to explore all solutions can grow exponentially with the size of the problem space. This can render it impractical for complex problems with a vast number of possibilities.
- **Limited Heuristics**: Brute-force doesn't rely on any specific knowledge or insights about the problem. It simply tests every option one by one.

The algorithm presented will make use of the brute-force search. The author proposes a more nuanced approach that leverages the characteristics of Smartle to optimize the brute-force search. Instead of blindly exploring all letter arrangements, the algorithm focuses on evaluating complete five-letter words. This initial focus on valid words reduces the search space significantly compared to a naive brute-force approach that considers every letter combination.

B. Backtracking

Backtracking is a powerful algorithmic technique for solving problems that involve finding all possible solutions or a single solution that meets specific constraints.

It employs a systematic, incremental approach to explore a tree of potential solutions, meticulously building candidates and strategically backtracking when dead ends are encountered.

Here's a breakdown of how backtracking works:

1. Initialization

Define the problem space and the constraints that valid solutions must adhere to. Establish a data structure to represent partial solutions (often as a list or array). If necessary, set up a mechanism to track solutions that have already been found.

2. Recursive Exploration

The core of backtracking lies in recursion. A function is designed to explore potential solutions by making choices that extend the current partial solution.At each step, a decision is made, adding a new element or taking an action that expands the partial solution. This decision-making process might involve iterating through all available options or evaluating a condition for each option.

3. Constraint Checking

After a choice is made, a crucial step is to verify if the extended partial solution still adheres to all the problem's constraints. If it does, the exploration continues. If a constraint is violated, it signifies that the current path leads to a dead end.

4. Backtracking

When a constraint violation occurs, the algorithm "backtracks." It discards the most recent choice and returns to the previous decision point in the search tree. This allows exploration of alternative options.

5. Base Cases

The recursive function typically has base cases that determine when to stop the exploration. These cases might involve:

- Reaching the final level of the search tree, indicating a complete solution has been found (and possibly stored or returned).
- Encountering a situation where no more valid choices can be made, signaling an exhausted branch that won't lead to a solution.

Backtracking can be considered an optimized version of the brute-force approach. While brute force exhaustively tries every single possible combination, backtracking intelligently prunes the search space by incorporating constraints and backtracking from dead ends. This significantly reduces the number of paths explored, making it more efficient for problems with well-defined constraints.

C. Greedy Algorithm

A greedy algorithm is an optimization technique that tackles problems by making the choice that seems best at the moment, hoping it will lead to a globally optimal solution in the end. It follows a "seize the best opportunity now" approach without necessarily considering the long-term consequences of those choices.

Core Principle

The fundamental idea behind a greedy algorithm is to iteratively select the option that appears most promising at each step. This selection is based on a specific criterion that defines what "best" means for the given problem.

Structure of a Greedy Algorithm

a. Initialization:

Set up the problem and establish the selection criteria (how to determine the "best" option).

b. Iterative Choice Making:

The algorithm enters a loop where it iteratively makes choices based on the selection criteria. At each step:

- It evaluates the available options.
- It selects the option that appears to be the most optimal according to the criteria.

c. Solution Construction:

As the algorithm makes choices, it builds up a solution incrementally. The chosen options are typically added to a data structure like a list or array.

d. Termination:

The loop continues until a stopping condition is met, which might involve:

- Reaching the maximum number of allowed choices.
- Fulfilling a specific goal condition.

Advantages:

- **Simplicity**: Greedy algorithms are often easy to understand and implement due to their step-by-step approach.
- **Efficiency**: In many cases, greedy algorithms can find good (sometimes even optimal) solutions quite efficiently compared to exhaustive search methods like brute force.

Disadvantages:

- **Non-Optimality**: Greedy algorithms don't guarantee finding the absolute best solution in all cases. The focus on the "best" option at each step might lead the algorithm down a path that overlooks a better overall solution.
- **Problem Dependence**: The effectiveness of a greedy algorithm heavily depends on the specific problem being solved. It works well for problems that exhibit a "greedy choice property," where the locally optimal choices at each step genuinely lead to a globally optimal solution.

Heuristics

Heuristics are essentially "rules of thumb" or educated guesses that provide a way to make decisions within an algorithm. They are based on experience, knowledge of the problem domain, or common-sense observations. While they don't guarantee the best possible solution, they offer a practical approach to navigate the search space effectively. Heuristics offer the following benefits:

- Reducing Search Space: By leveraging heuristics, algorithms can prioritize exploration of more promising areas of the solution space, significantly reducing the number of paths to be examined. This is particularly beneficial for problems with vast or exponential search spaces.
- Finding Good (or Optimal) Solutions: Well-designed heuristics can often lead algorithms to good, near-optimal, or even optimal solutions in a reasonable amount of time. This makes them valuable for problems where finding the absolute best solution might not be feasible due to time or resource constraints.
- Guiding Exploration: Heuristics can act as a compass, directing the algorithm towards areas that are more likely to contain solutions based on domain-specific knowledge. This targeted exploration enhances efficiency compared to random or exhaustive search approaches.

D. Depth-first Search

Depth-first search (DFS) is an algorithm for traversing a tree or graph data structures. It's a systematic exploration technique that delves as deeply as possible along each branch before backtracking to explore other options. Here's a comprehensive explanation of DFS:

Core Idea:

DFS starts at a root node (in a tree) or any arbitrary node (in a graph) and explores its connected neighbors. It then recursively visits each neighbor's unvisited neighbors, essentially going down one path at a time until it reaches a dead end (a node with no unvisited neighbors). Once at a dead end, the algorithm backtracks, returning to the most recent node that has unvisited neighbors, and explores those instead. This process continues until all possible paths have been explored.

1. Initialization:

- Mark all nodes in the graph or tree as unvisited.
- Choose a starting node (root node in a tree or any node in a graph).

2. Recursive Exploration:

- Mark the current node as visited.
- Iterate through all the unvisited neighbors of the current node.
    - For each unvisited neighbor:
        - Recursively call the DFS function on that neighbor. This is the essence of depth-first exploration.

3. Base Case (Backtracking):

- If the current node has no unvisited neighbors (i.e., a dead end), backtrack. Backtracking involves returning from the current function call, allowing the previous recursive call to explore its unvisited neighbors.

E. Dynamic Programming

Dynamic programming (DP) is a powerful optimization technique used to solve problems by breaking them down into smaller, overlapping subproblems. It solves these subproblems once and stores the results to avoid redundant computations, leading to significant efficiency gains.

The key to dynamic programming is identifying that problem P can be decomposed into smaller subproblems denoted by P_i (where i represents a specific subproblem instance). These subproblems typically share a specific structure or property, where:

Optimal Substructure: Each subproblem P_i has an optimal solution that can be constructed from the optimal solutions to its subproblems. This aligns with the Bellman Principle.

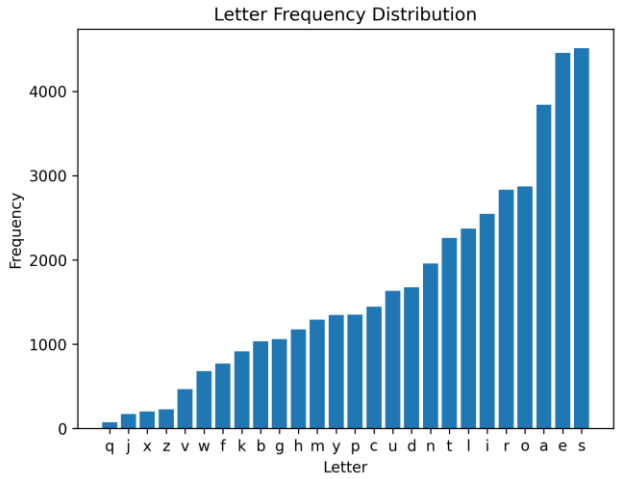
Overlapping Subproblems: There exists a significant overlap between the subproblems P_i. Solving the same subproblem multiple times leads to redundant computations.

III. Algorithm Analysis

Before we start designing the algorithm. It helps to know several things:

a. The number of possible 5 letter words and the distribution of the characters

The implemented program uses a dictionary of around 8600 5-letter words. The letter frequency plot is as follows:



b. The number of possible valid combinations

There are $25! \approx 1.5 \cdot 10^{25}$ possible arrangements of the letters. However, we can only form valid English words, so there are at least $\binom{8600}{5} \approx 3.91 \cdot 10^{17}$ 5 words combinations. On a typical computer, a program can do around 10^6 to 10^9 operations per second, so it is important that we don't naively attempt the problem. Note that the number of possible valid combinations is still less than the number of possible 5 words combinations because we are limited to the possible letters given in the beginning of the game.

Under the following knowledge above, we can start developing the algorithm.

It is best to split the problem into two parts:

1. Finding the ideal 5-word combinations.
2. Finding the minimal number of swaps needed to convert the initial 5x5 grid to the one found in (1).

We will design an algorithm to solve (2) first, and then (1).

(2) is well-known to be NP-hard (it is the same problem as finding the minimal number of swaps needed to convert one string to another string of equal length but different character positions). To efficiently solve (2), we will use greedy-based heuristics that are suitable to the nature of the game. Given two configurations (the original one and the one generated in (1)), the (approximated) minimal number of swaps can be generated as follows:

i. Given the 5 words generated in (1), there are $5! = 120$ possible configurations. Choose the combination with the most letters matched with the initial configuration. After choosing the specified configuration, we can count the (approximated) minimal number of swaps needed.

ii. Any letter in the correct position should not be swapped.

iii. If there are 2 cells where swapping increases the number of letters matched by two, then swap those two cells.

iv. Repeat (iii) until it is not possible.

v. Assuming (ii) and (iii) are not possible, find the first cell such that it is not matched. Then swap that cell with the (another) first cell so that when it is swapped, the number of letters matched is increased by 1. Repeat step (ii) - (v) until every letter is matched.

(2) can only be used when step (1) is done. There are many ways to do (1). However, the chosen way to do step (1) is based on an approach like how humans solve the problem. The human-based approach is to initially form the words using rarer letters. If the first four words have been formed, but the last 5 letters can't be used to form a word, then the human-based approach is to modify the first four words. The translation of this human approach to the computing approach is to use backtracking. The backtracking used in the algorithm to solve (1) is as follows:

i. Sort the dictionary of 5 letter words by rarity of the characters.

ii. The recursive backtracking function is called.

iii. If the size of the answer dynamic array is already 5. Then we can add this configuration to the list of configurations that will be checked by part (2) of the algorithm. Return out of the function to the calling function; go to step (iv).

iv. Iterate every word in the dictionary. For every word that is not already visited, it pushes that word (in the sorted dictionary) to the answer dynamic array and calls the recursive backtracking function.

Step (i) of part (1) of the algorithm needs to be elaborated. There are many ways to sort the dictionary. One particularly naive but effective way is to weigh the words by the sum of the frequency of each letter. For instance, if letter 'a' occurs 5 times, 'b' occurs 3 times, and 'c' occurs 2 times, then the word 'abbcc' is given a weight of $1(5) + 2(3) + 2(2) = 15$. We can then sort the words in the dictionary by ascending weights. Another way is to shuffle the dictionary. However, this approach leads to fewer configurations, and thus is not preferred. The opposite approach of sorting the weights descending leads to worse answers, and thus is also not preferred

There are further issues in (1). It is needed to know when to stop generating the configurations since there are many combinations that can't be checked by the average computer. We can specify a limit on the number of configurations generated by (1) so that the search stops when it has reached this

limit. The program's user can also specify this limit according to the amount of time they can afford to.

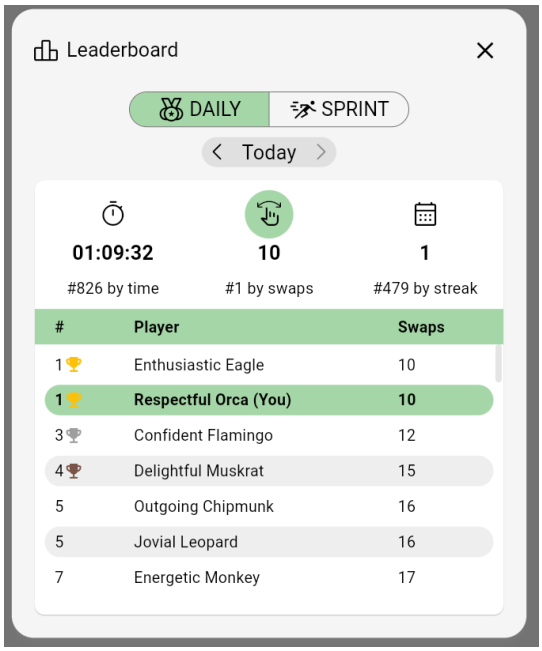
Another completely different approach to (1) is to use dynamic programming. In the dynamic programming approach, there is a bitmask of 25 bits. Each bit corresponds to a letter in the 5x5 grid. For every submask where multiple of 5 bits are set, we can store the dynamic array consisting of all the possible strings that can be formed such that for every string, the remaining characters left (which is also a multiple of 5 since a multiple of 5 subtracted by 5 is still a multiple of 5) can also be used to form 5 letter words. This solution is much more efficient than the brute force solution. However, there are two issues compared to the backtracking solution. The first issue is that it is computationally more expensive in the game with 5x5 grid. The second issue is that it is not flexible as compared to the backtracking solution; the dynamic programming approach doesn't allow to generate a single solution until all of the states have been figured out. For these two reasons, the dynamic programming approach is not used in generating the configurations to be checked (using part (2) of the algorithm).

IV. Implementation Analysis

```
Minimum number of swaps: 10
i(2, 3), o(1, 3)
h(4, 2), a(1, 5)
h(3, 2), s(1, 2)
c(4, 5), o(1, 4)
o(4, 5), s(3, 2)
u(4, 1), d(2, 4)
j(5, 5), d(4, 1)
t(3, 3), o(2, 5)
u(5, 1), o(3, 3)
o(5, 1), v(4, 4)

which
snout
roupy
jatos
viand


real    6m4,318s
user    6m3,588s
sys     0m0,144s
```



For the initial configuration given in the introduction, the implemented program has figured out the (approximated) minimal number of swaps to be 10. This is currently the most optimal score in the leaderboard of the game.

The program solved the problem in around 6 minutes, by limiting the number of configurations generated by step (1) of the algorithm to exactly 100000 configurations. A shorter limit of 10000 gives a worse solution using 12 swaps but ran in less than 10 seconds. A larger limit of 1000000 configurations gives the same solution using 10 swaps but ran for 104 minutes (about 1 hour 44 minutes) and thus a larger limit in this scenario is not ideal.

V. Code

1. Sort the initial dictionary 'words.txt' based on the weights as defined previously. The output is a sorted dictionary 'priority_words.txt' that will be used in the program in the backtracking part of the code.

```python
with open('words.txt') as file:
    words = file.readlines()
    words = [word.strip() for word in words]
```

```python
alphabet_frequency = {}
for word in words:
    for letter in word:
        if letter in alphabet_frequency:
            alphabet_frequency[letter] += 1
        else:
            alphabet_frequency[letter] = 0
```

```python
word_weight = {}
for word in words:
    weight = 0
    for letter in word:
        weight += alphabet_frequency[letter]
    word_weight[word] = weight
```

```python
word_weight = [t[0] for t in sorted(word_weight.items(), key=lambda x: x[1])]
```

```python
with open('priority_words.txt', 'w') as file:
    file.write('\n'.join(word_weight))
```

2. Stores the possible valid configurations by backtracking.

```cpp
constexpr int TAKEN_COMBINATION_SOLUTION = 10000;

char game[5][5];
char letters[25];
int frequency[26];
vector<string> current;
bool taken[26 * 26 * 26 * 26 * 26];
vector<string> dictionary;

vector<vector<string>> word_combination_solutions;

int char_to_int(char c) {
    return c - 'a';
}

int word_to_int(string word) {
    int result = 0;
    result += char_to_int(word[0]);
    result += char_to_int(word[1]) * 26;
    result += char_to_int(word[2]) * 26 * 26;
    result += char_to_int(word[3]) * 26 * 26 * 26;
    result += char_to_int(word[4]) * 26 * 26 * 26 * 26;
    return result;
}

void read_input() {
    for (int row = 0; row < 5; row++) {
        for (int col = 0; col < 5; col++) {
            cin >> game[row][col];
            letters[5 * row + col] = game[row][col];
            frequency[game[row][col] - 'a']++;
        }
    }
}

void read_dictionary() {
    string filename = "priority_words.txt";

    std::ifstream input_file(filename);

    if (input_file.is_open()) {
        string word;

        while (std::getline(input_file, word)) {
            dictionary.push_back(word);
        }

        input_file.close();
    } else {
        std::cerr << "Error: Could not open dictionary '" << filename << "'" << endl;
    }
}
```

```cpp
void search(vector<string>::iterator current_it) {
    if (current.size() == 5) {
        word_combination_solutions.push_back(current);
        return;
    }

    for (auto it = current_it; it != dictionary.end(); it++) {
        if (word_combination_solutions.size() == TAKEN_COMBINATION_SOLUTION)
            return;

        string word = *it;
        int word_number = word_to_int(word);
        if (taken[word_number])
            continue;

        for (auto letter : word)
            frequency[letter - 'a']--;

        bool valid = true;
        for (auto letter : word)
            if (frequency[letter - 'a'] < 0)
                valid = false;

        if (!valid) {
            for (auto letter : word) {
                frequency[letter - 'a']++;
            }
            continue;
        }

        taken[word_number] = true;
        current.push_back(word);
        search(it + 1);
        taken[word_number] = false;
        current.pop_back();
        for (auto letter : word)
            frequency[letter - 'a']++;
    }
}
```

3. Implements the greedy heuristic as defined previously to evaluate a particular word combination.

```cpp
Solution calculate_swaps(vector<string> combination_solution) {
    vector<int> indices{0, 1, 2, 3, 4};
    vector<int> optimal_indices;
    int matched = -1;
    do {
        int current_weight = 0;
        for (int row = 0; row < 5; row++)
            for (int col = 0; col < 5; col++)
                if (combination_solution[indices[row]][col] == game[row][col])
                    current_weight++;
        if (current_weight > matched) {
            matched = current_weight;
            optimal_indices = indices;
        }
    } while (std::next_permutation(indices.begin(), indices.end()));

    char combination[5][5];
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            combination[i][j] = combination_solution[optimal_indices[i]][j];
    char game_copy[5][5];
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            game_copy[i][j] = game[i][j];

    Solution solution;
    while (matched != 25) {
        for (int i = 0; i < 5; i++)
            for (int j = 0; j < 5; j++) {
                if (game_copy[i][j] == combination[i][j])
                    continue;
                for (int x = 0; x < 5; x++)
                    for (int y = 0; y < 5; y++) {
                        if (i == x && j == y)
                            continue;
                        if (game_copy[x][y] == combination[x][y])
                            continue;

                        if (game_copy[i][j] != game_copy[x][y] && game_copy[i][j] == combination[x][y] && game_copy[x][y] == combination[i][j]) {
                            matched += 2;
                            solution.swaps.push_back({Coordinate(x, y), Coordinate(i, j)});
                            swap(game_copy[x][y], game_copy[i][j]);
                        }
                    }
            }
        for (int i = 0; i < 5; i++)
            for (int j = 0; j < 5; j++)
                if (game_copy[i][j] != combination[i][j])
                    for (int x = 0; x < 5; x++)
                        for (int y = 0; y < 5; y++) {
                            if (i == x && j == y)
                                continue;
                            if (game_copy[x][y] == combination[x][y])
                                continue;
                            if (game_copy[x][y] == combination[i][j]) {
                                matched++;
                                swap(game_copy[x][y], game_copy[i][j]);
                                solution.swaps.push_back({Coordinate(x, y), Coordinate(i, j)});
                                goto done;
                            }
                        }
    done:;
    }
    return solution;
}
```

4. The main program initially calls the backtracking function. Then, each possible word combination is evaluated using the greedy heuristic, and the one with the minimal number of moves is chosen.

```cpp
int main() {
    cout << "Reading input..." << endl;
    read_input();
    cout << "Done reading input..." << endl;
    cout << "Reading dictionary..." << endl;
    read_dictionary();
    cout << "Done reading dictionary..." << endl;
    cout << "Generating possible configurations..." << endl;
    search(dictionary.begin());
    cout << "Done generating possible configurations..." << endl;
    cout << "Checking each possible configuration..." << endl;

    Solution solution;
    solution.swaps = vector<pair<Coordinate, Coordinate>>(26, {Coordinate(), Coordinate()});
    vector<string> optimal_combination;
    for (auto combination_solution : word_combination_solutions) {
        auto current_swaps = calculate_swaps(combination_solution);
        if (current_swaps.swaps.size() < solution.swaps.size()) {
            solution.swaps = current_swaps.swaps;
            optimal_combination = combination_solution;
        }
    }

    cout << "Minimum number of swaps: " << solution.swaps.size() << '\n';

    for (auto swap : solution.swaps) {
        cout << game[swap.first.x][swap.first.y] << "(" << swap.first.x + 1 << ", " << swap.first.y + 1 << "), "
             << game[swap.second.x][swap.second.y] << "(" << swap.second.x + 1 << ", " << swap.second.y + 1 << ")" << '\n';
        std::swap(game[swap.first.x][swap.first.y], game[swap.second.x][swap.second.y]);
    }
    cout << '\n';
    for (int row = 0; row < 5; row++) {
        for (int col = 0; col < 5; col++) {
            cout << game[row][col];
        }
        cout << '\n';
    }
    cout << endl;
}
```

VI. Conclusion

Both backtracking and greedy heuristic approaches prove highly effective in solving the word-based puzzle game Smartle. While these algorithms may not be theoretically optimal or exhaustive, they consistently outperform human players by a significant margin, demonstrating their success. Moreover, analysis of program-generated solutions reveals a surprising level of ingenuity.

Acknowledgment

I would like to express my sincere gratitude to Dr. Ir. Rinaldi, M.T., my lecturer for the IF2211 Algorithm Strategy course. Their guidance and instruction were instrumental in my understanding of the concepts presented in this paper.

Furthermore, I am grateful to Allah for his guidance and providence throughout this endeavor.

References

[1] Smartle - your daily word puzzle. (2024, June 8). Smartle. https://smartle.net/

[2] Munir, R. (2024, June 11). Algoritma runut-balik (backtracking). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian1.pdf

[3] Munir, R. (2024, June 11). Algoritma runut-balik (backtracking). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-backtracking-2021-Bagian2.pdf

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, 12 Juni 2024

Azmi Mahmud Bazeid, 13522109