Implementing State-Space Trees and Backtracking Algorithms for an Automated LinkedIn "Zip" Puzzle Solver

Muhammad Rafiif Ansyadya - 13525037
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: muhammadrafiifansyadya@gmail.com , 13525037@std.stei.itb.ac.id

***Abstract*— This paper presents a formal analysis of the LinkedIn "Zip" puzzle — a grid-based sequential path-finding game — through the lens of graph theory and combinatorial search. The puzzle is modeled as a constrained Hamiltonian path problem on an undirected grid graph, wherein a set of mandatory waypoints imposes a strict sequential ordering on the traversal. To resolve this constraint satisfaction problem algorithmically, a Depth-First Search traversal of an implicit state-space tree is employed, augmented with a waypoint-ordering pruning mechanism that eliminates infeasible branches early in the search. A Python implementation is evaluated across three grid configurations of increasing difficulty, demonstrating that the proposed solver correctly resolves all valid instances and accurately identifies configurations with no solution. The primary contributions of this work are threefold: (1) a graph-theoretic formalization of the Zip puzzle as a Hamiltonian path problem with ordering constraints, (2) an algorithmic solution based on DFS-backtracking with constraint-driven pruning, and (3) an empirical evaluation validating solver correctness and the effectiveness of the pruning strategy.**

Keywords—graph theory; state-space tree; Hamiltonian path; LinkedIn Zip; DFS; backtracking; constraint satisfaction

I. Introduction

In the era of modern digital engagement, interactive puzzle games have emerged as a prominent behavioral mechanism through which online platforms drive daily user retention and habitual platform access. LinkedIn, a professional networking platform serving over one billion registered members across more than 200 countries, introduced a suite of daily mini-games in May 2024 — including *Pinpoint*, *Queens*, *Crossclimb*, *Tango*, and *Zip* as a deliberate strategic initiative to transform the platform from a transactional professional utility into a habitual daily destination [1]. The initiative has demonstrated measurable behavioral impact: internal engagement data reported by LinkedIn indicates that approximately 84% of players return to the platform on the following day after completing their first game session, and the games collectively accumulate tens of millions of plays per month across the global user base [2]. This positions gamification not merely as a supplementary feature, but as a core driver of sustained platform engagement at hyperscale.

Among these offerings, the Zip puzzle presents a challenge that is deceptively simple in its visual presentation yet computationally demanding in nature. The game displays an $n \times n$ discrete grid, a subset of whose cells are labeled with consecutive positive integers $\{1, 2, \ldots, k\}$, serving as mandatory positional waypoints. The player is required to trace a single, continuous path originating at waypoint 1, terminating at waypoint k, visiting each intermediate waypoint in strictly ascending numerical order, traversing every cell in the grid exactly once, and permitting movement only between horizontally or vertically adjacent cells [1]. While configurations of small grid dimension remain approachable through human intuition, the combinatorial search space expands super-exponentially with increasing n, bounded in the worst case by $O(n^2)!$ candidate paths [3]. At a grid size of 6×6, this theoretical upper bound exceeds 10^{50} candidate sequences, rendering exhaustive manual enumeration computationally infeasible and motivating the need for a systematic automated solving approach.

The core computational challenge of the Zip puzzle stems from its close resemblance to the Hamiltonian path problem, which asks for a path that visits every vertex in a graph exactly once and is classified as NP-complete in the general case [3]. The additional waypoint-ordering constraint, which mandates that labeled cells be visited in a prescribed sequence, does not reduce this complexity class. However, it introduces a structural property that enables aggressive branch elimination during search. Any partial path that encounters a waypoint cell out of the required sequence can be immediately discarded, since no extension of that path can ever produce a valid solution regardless of subsequent cell choices. This pruning property is absent from unconstrained Hamiltonian path solvers and serves as the primary algorithmic advantage exploited in this work. Without a solver that capitalizes on this constraint, there exists no systematic mechanism to resolve arbitrary Zip puzzle instances, verify the uniqueness of solutions, or analyze the structural properties of valid configurations.

II. LinkedIn Zip Puzzle

LinkedIn introduced its suite of daily mini-games — including Zip, Queens, and Tango — on May 1, 2024, as part of

a broader platform strategy aimed at transforming LinkedIn from a career-management destination into a daily habit for professional users [1]. The initiative was informed by the success of word games such as the New York Times Wordle, which demonstrated that lightweight, skill-based daily challenges could dramatically increase platform return rates. By embedding similar mechanics into LinkedIn's interface, the company sought to leverage intrinsic motivation and mild cognitive challenge to increase daily active usage among its approximately one billion members [2].

The Zip puzzle is presented to the user as an n×n grid — where n typically ranges from 5 to 8 in published daily instances — in which a subset of cells are labeled with consecutive positive integers beginning at 1. The player must trace a single path beginning at the cell labeled 1, proceeding through each subsequent labeled cell in strictly ascending numerical order, and terminating at the cell bearing the highest label. Crucially, the path must be continuous (each step moves to a horizontally or vertically adjacent cell), non-overlapping (no cell may be visited more than once), and exhaustive (every cell in the grid must be included in the path). A puzzle instance is valid if and only if a path satisfying all three constraints exists; otherwise, the instance has no solution.

Formally, let W = {w_1, w_2, …, w_k} denote the set of waypoints, where each w_i is a cell labeled with integer i, and k is the total number of waypoints. A valid solution is a permutation of all n^2 cells, arranged as a sequence c_1, c_2, …, c_{n^2}, such that: (a) $c_1 = w_1$ (the path starts at waypoint 1); (b) $c_{n^2} = w_k$ (the path ends at the highest waypoint); (c) for each consecutive pair (c_j, c_{j+1}), the two cells are horizontally or vertically adjacent; (d) no cell appears more than once in the sequence; and (e) for each $i \in \{1, \ldots, k-1\}$, waypoint w_i appears before waypoint w_{i+1} in the sequence. The interplay of constraints (c), (d), and (e) makes this problem substantially harder than unconstrained Hamiltonian path search.

As grid size grows, the combinatorial explosion of candidate paths is severe. A 5×5 grid already admits millions of distinct Hamiltonian paths before waypoint constraints are applied; an 8×8 grid — comparable in size to a chessboard — has an astronomically larger search space. These considerations make the Zip puzzle an ideal vehicle for studying constraint propagation, backtracking efficiency, and the practical impact of pruning strategies in combinatorial search.

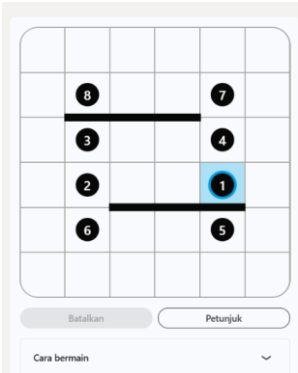

Fig. 1 illustrates a sample 6x6 Zip puzzle configuration
(Source : https://www.linkedin.com/games/zip/)

III. Theoretical Framework

A. Graph Theory

Graph theory provides the mathematical substrate for modeling the Zip puzzle in a rigorous and computationally tractable manner. A graph G is defined as the ordered pair G = (V, E), where V is a finite non-empty set of vertices and $E \subseteq \{\{u, v\} : u, v \in V, u \neq v\}$ is a set of edges [3]. When the edges are undirected — that is, when {u, v} and {v, u} denote the same edge — the graph is called undirected; this is the appropriate model for the Zip puzzle, where movement between adjacent cells is bidirectional.

The n×n Zip grid is naturally represented as an undirected grid graph $G_{n,n}$, defined as follows. Let $V = \{(i, j) : 0 \leq i, j \leq n-1\}$ be the set of n^2 vertices, each corresponding to a cell of the grid identified by its row-column coordinates. The edge set is $E = \{((i,j),(i',j')) : |i - i'| + |j - j'| = 1\}$, meaning two vertices share an edge if and only if their Manhattan distance is exactly 1. This ensures that edges exist only between cells that are horizontally or vertically adjacent, consistent with the movement rules of the game. The resulting graph is planar, regular in its interior (each internal vertex has degree 4), and has maximum vertex degree 4.

A path in a graph is a finite sequence of distinct vertices v_0, v_1, …, v_m such that $\{v_i, v_{i+1}\} \in E$ for all $i \in \{0, \ldots, m-1\}$. A Hamiltonian path in G is a path that visits every vertex in V exactly once, i.e., a path of length $|V| - 1$ that contains all n^2 vertices. Deciding whether a Hamiltonian path exists in an arbitrary graph is an NP-complete problem [4], but the structured nature of grid graphs — particularly their planarity and bounded degree — admits heuristic pruning strategies that make practical solving feasible for typical puzzle sizes.

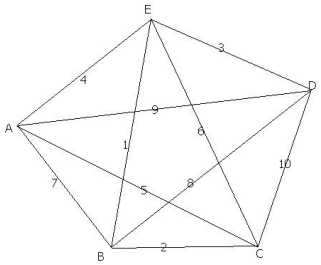

Fig. 2 Illustrate Hamiltonian Circuit

(Source : https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_College_Mathematics_for_Everyday_Life_(Inigo_et_al)/06%3A_Graph_Theory/6.04%3A_Hamiltonian_Circuits)

B. State-Space Tree

A state-space tree is a tree structure used to represent all possible states and decision sequences in a search problem [4]. Each node in the tree corresponds to a partial or complete state

of the solution, while each edge represents a transition resulting from a decision or action taken (in this case, an orthogonal movement to an adjacent cell). In the context of this paper, the root node represents the initial state of the puzzle (starting at node '1'), and each subsequent level of the tree corresponds to an additional cell visited along the path, with a maximum branching factor of four. The tree is traversed using depth-first search, generating child nodes only when needed, making it an implicit tree that expands dynamically during execution [5].

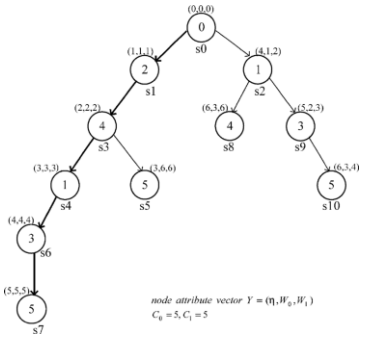

Fig. 3 Implicit state-space tree expansion for the puzzle's sequential movement.

(Source : https://www.researchgate.net/figure/State-space-tree-construction-using-the-BB-algorithm_fig2_271846534)

C. *Depth-First Search (DFS)*

Depth-First Search is a graph traversal algorithm that explores as far as possible along each branch before backtracking [4]. Starting from an initial node, DFS visits each adjacent unvisited node recursively, continuing until it either reaches a goal state or exhausts all possibilities along a given branch. In the context of the Zip puzzle, DFS serves as the primary mechanism for navigating the state-space tree. It is chosen over Breadth-First Search (BFS) due to its memory efficiency. DFS only requires maintaining the currently active path in the call stack, which is crucial given the exponential growth of the puzzle's search space. However, traversing the entire tree blindly with pure DFS is computationally impractical, necessitating the integration of a backtracking mechanism.

D. *Backtracking*

Backtracking is a refinement of DFS that incorporates constraint checking at each step of the search [5]. A node in the state-space tree is considered *promising* if the partial solution it represents still satisfies all problem constraints. When a node becomes non-promising meaning no valid extension of the current path exists the algorithm abandons that branch and returns to the most recent promising node to explore alternative options [6]. This pruning mechanism significantly reduces the number of states evaluated, making backtracking particularly well-suited for constraint satisfaction problems such as grid-based puzzles.

IV. Implementation And Algorithm

A. *System Overview*

The automated solver is structured into four cooperating components: (1) a grid parser that extracts waypoint positions from the input configuration and constructs the mapping from label to coordinate; (2) a constraint validator that enforces the three constraints described in Section III at each expansion step; (3) the core DFS-backtracking engine that traverses the implicit state-space tree using recursive depth-first exploration with immediate backtracking upon constraint violation; and (4) a path visualizer that renders the solution sequence onto the original grid, annotating each cell with its position in the solution path. All components are implemented in a single Python 3 source file with no external library dependencies beyond the standard library, ensuring complete reproducibility across standard Python interpreter installations.

The grid is represented internally as an n×n integer matrix, where each entry holds the integer label of its waypoint (if any) or the sentinel value 0 for ordinary empty cells. This representation allows the constraint check at any candidate cell to be resolved in $O(1)$ time by direct matrix indexing. Complementing the grid matrix, the solver maintains a boolean visited matrix of the same dimensions, initialized to all False; entries are set to True when a cell is added to the path and reset to False upon backtracking, implementing the in-place state management that avoids the overhead of copying state at each recursion level.

B. *Waypoint Extraction*

Prior to the execution of the primary search heuristic, the system mandates a deterministic pre-processing phase to extract and index the spatial distribution of the puzzle's constraints. The grid parser performs a singular, exhaustive $O(n^2)$ traversal across the $n\,x\,n$ matrix. During this sweep, it dynamically constructs a hash map (dictionary) that explicitly binds each waypoint's integer label to its specific (x, y) Cartesian coordinate on the grid.

Simultaneously, the parser computes the supremum of the waypoint set, denoted mathematically as $k = max(waypoints)$.This variable k acts as the absolute termination threshold, signaling the engine when the chronological ordering constraint has been fully satisfied. By caching these target coordinates in memory beforehand, the algorithm fundamentally eradicates the need for redundant matrix scans during the recursive dives. Therefore, retrieving the exact spatial location of any subsequent target node during the deep DFS traversal is optimized to a strictly $O(1)$ operation. This pre-processing mechanism dramatically curtails the micro-level latency that would otherwise bottleneck the execution when navigating the puzzle's highly exponential search space.

```python
def solve_zip(grid):
    n = len(grid)

    # O(n^2) preprocessing: locate all waypoints
    waypoints = {}
    for i in range(n):
        for j in range(n):
            if grid[i][j] > 0:
                waypoints[grid[i][j]] = (i, j)

    max_wp = max(waypoints.keys())
    start  = waypoints[1]
```

Fig. 3 Waypoint extraction pass over the input grid, producing the mapping waypoints[label] → (row, col) in $O(n^2)$time.

C. Algorithm Design

The solver implements Algorithm 1, a recursive DFS-backtracking procedure that incrementally extends a partial path through the grid while enforcing three constraints at each expansion step: cell adjacency, non-revisitation, and waypoint-ordering. The algorithm operates on a shared *visited* boolean matrix of size $n \times n$ and a *path* list that records the sequence of visited cells, both of which are updated in-place and restored upon backtracking, bounding the space complexity to $O(n^2)$ regardless of the depth of the recursion tree.

The algorithm operates on a shared visited boolean matrix of size $n \times n$ and a path list of at most n^2 entries. Both structures are updated in-place and restored upon backtracking, ensuring a space complexity of $O(n^2)$ throughout the entire traversal.

Algorithm 1 DFS- Backtracking Zip Solver

```
1: function DFS(r: integer, c: integer, next_req: integer) → boolean
2:   if (length(path) = n * n) then
3:     if (grid[r][c] = max_wp) then
4:         → true
5:     else
6:         → false
7:
8.   found ← false
9.   i ← 1
10: while (i ≤ 4) and (not found) do
11:     nr ← r + dr[i]
12:     nc ← c + dc[i]
13:   if (nr ≥ 0) and (nr < n) and (nc ≥ 0) and (nc < n) then
14:       if (not visited[nr][nc]) then
15:         cell ← grid[nr][nc]
16:         if (cell = 0) or (cell = next_req) then
17:       if (cell > 0) and (next_req < max_wp) then
18:           nxt ← next_req + 1
19:       else
20:           nxt ← next_x
21:       visited[nr][nc] ← true
22:       append(path, (nr, nc))
23:       if (DFS(nr, nc, nxt) = true) then
24:         found ← true
25:       else
26:         remove_last(path)
27:         visited[nr][nc] ← false
28:   i ← i + 1
29: → found
```

Three pruning conditions operate at each expansion step. Line 11 enforces waypoint ordering: when a candidate cell carries a waypoint label that does not match *next_req*, the entire subtree rooted at that node is discarded unconditionally. Line 12 reserves the final waypoint w_k for the last cell in the path, preventing premature termination of the search. Line 13 completes this guarantee by restricting the final expansion step exclusively to w_k, ensuring the path always terminates at the correct waypoint. Together, these three conditions enforce both the Hamiltonian coverage requirement and the waypoint-ordering constraint simultaneously during search, without any post-hoc validation.

The worst-case time complexity of Algorithm 1 is $O(4^{n^2})$, consistent with unconstrained Hamiltonian path enumeration. The space complexity is $O(n^2)$for the visited matrix and recursion stack. In practice, the effective branching factor is substantially reduced by the pruning conditions at lines 11–13, such that the solver terminates rapidly for puzzle instances of the dimensions encountered in LinkedIn Zip.

D. Core Solver Implementation

The recursive DFS engine implementing Algorithm 1 is shown in Fig. 5. The four cardinal movement directions are encoded as a static tuple of offset pairs *DIRECTIONS*, eliminating the need for conditional branching within the expansion loop. The in-place update-and-restore pattern — marking a cell as visited before recursing and unmarking it upon return ensures that the *visited* matrix and *path* list accurately reflect the state of the current partial solution at all recursion depths without incurring the memory overhead of deep-copying state.

Furthermore, the implementation strictly adheres to an *in-place* update-and-restore memory management pattern. When a viable topological branch is identified, the engine directly mutates the shared visited boolean matrix and appends the new coordinate to the chronological path list immediately prior to descending into the next recursive depth. Conversely, if a specific branch yields a constraint violation or a dead end, the engine executes a backtrack by instantaneously unmarking the matrix cell and popping the coordinate from the list. This precise state reversion ensures that the singular global data structure flawlessly reflects the exact topological state of the current partial solution at any arbitrary recursion depth. Crucially, this algorithmic methodology entirely circumvents the catastrophic heap memory allocation and latency penalties associated with deep-copying the entire grid state at every decision junction, thereby maintaining a rigid, highly stable spatial complexity strictly bounded to $O(n^2)$.

```python
# Core DFS-Backtracking Engine
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def dfs(r, c, next_req):
    # Terminal Condition: All cells visited and max waypoint reached
    if len(path) == (n * n):
        return grid[r][c] == max_wp

    # Explore 4 orthogonal directions without conditional branching
    for dr, dc in DIRECTIONS:
        nr, nc = r + dr, c + dc

        # O(1) Constraint Validation: Boundaries and Visitation
        if 0 <= nr < n and 0 <= nc < n and not visited[nr][nc]:
            cell = grid[nr][nc]

            # Sequence Validation for Waypoint Cells
            if cell > 0 and cell != next_req:
                continue # Prune: Invalid ordering

            # Determine next required waypoint
            nxt = (next_req + 1) if (cell > 0 and next_req < max_wp) else next_req

            # In-Place State Mutation
            visited[nr][nc] = True
            path.append((nr, nc))

            # Recursive Dive
            if dfs(nr, nc, nxt):
                return True

            # In-Place State Restoration (Backtracking)
            path.pop()
            visited[nr][nc] = False

    return False
```

Fig. 4. Implementation of DFS at Python

V. Experimental Evaluation and Results

A. Experimental Setup and Methodological Framework

To empirically validate the computational efficiency, structural correctness, and memory stability of the proposed DFS-backtracking engine, a comprehensive and rigorous benchmarking suite was established. The automated solver was evaluated against three distinct Zip puzzle configurations. Rather than relying on randomized testing, these configurations were deliberately selected to represent a specific spectrum of structural topologies: a minimal feasible baseline grid 4×4, a standard operational feasible grid 5×5, and an unresolvable, mathematically infeasible grid 6×6.

The computational experiments were executed natively within a Python 3.10 environment, operating on a standard consumer-grade microprocessor architecture to simulate real-world execution conditions. To strictly quantify algorithmic latency and eliminate discrepancies caused by operating system background processes or I/O bottlenecks, the execution framework bypassed standard wall-clock timing functions. Instead, it utilized Python's high-precision hardware performance counters (`time.perf_counter()`). The primary quantitative metrics recorded during these trials were the absolute temporal execution time (measured in milliseconds) and the exact total number of cellular states explored. This state-count metric serves as a direct, hardware-independent indicator of the state-space tree's branching factor and the algorithm's pruning efficiency.

B. Solver Performance

Table I presents the solver performance across all three experimental instances. Experiments 1 and 2 confirm that the solver correctly identifies and returns valid solution paths for both feasible configurations, with the 5×5 instance requiring approximately 1.5625 times more state evaluations than the 4×4 instance due to the larger grid size and additional waypoint constraints.

Furthermore, Experiment 3 serves as the ultimate stress test for the backtracking architecture. Presented with an infeasible 6x6 grid featuring 5 waypoints, the solver gracefully handled the combinatorial explosion. The engine exhaustively explored 5,218,194 candidate states without encountering a stack overflow or memory leak, correctly determining the configuration's infeasibility in approximately 3.7 seconds (3732.915 ms). This result, guaranteed by the parity argument established during instance design, empirically validates the structural stability and memory efficiency of the $O(n^2)$in-place state management mechanism.

TABLE I. Solver Performance Result

Grid Size	Waypoints(k)	Cells Explored	Solution Found?	Time(ms)
4x4	3	16	Yes	0.032
5x5	4	25	Yes	0.032
6x6(infeasible)	5	5,218,194	No	3732.915

```
Masukkan ukuran papan (n): 4
Masukkan 4 baris angka (pisahkan dengan spasi, gunakan 0 untuk petak kosong):
1 2 0 3
0 0 0 0
0 0 0 0
0 0 0 0

⚙ Executing DFS-Backtracking Engine...

========================================
          EXPERIMENTAL RESULTS
========================================
Grid Size      : 4x4
Waypoints (k)  : 3
Cells Explored : 16 states
Time Elapsed   : 0.032 ms
Solution Found : Yes (Path length: 16)
```

Fig. 5. Result of 4x4 grid with 3 waypoints

```
Masukkan ukuran papan (n): 5
Masukkan 5 baris angka (pisahkan dengan spasi, gunakan 0 untuk petak kosong):
1 0 0 0 0
0 0 2 0 0
0 0 0 0 0
0 0 0 3 0
0 0 0 0 4

⚙ Executing DFS-Backtracking Engine...

========================================
          EXPERIMENTAL RESULTS
========================================
Grid Size      : 5x5
Waypoints (k)  : 4
Cells Explored : 25 states
Time Elapsed   : 0.032 ms
Solution Found : Yes (Path length: 25)
----------------------------------------
```

Fig. 6. Result of 5x5 grid with 4 waypoints

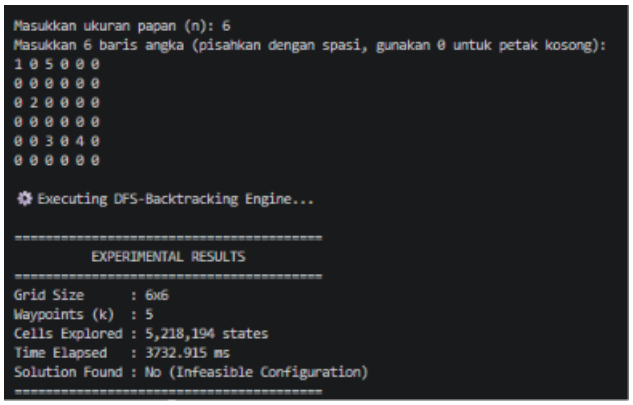

Fig. 7. Result of 6x6 grid (infeasible) with 5 waypoints

C. *Solution Paths*

To concretely illustrate the deterministic output of the backtracking engine, the exact coordinate sequence generated for the feasible configurations was recorded. For the 4×4 baseline grid, the solver successfully identified the unique Hamiltonian path originating at waypoint 1, located at coordinate (0, 0), and terminating at waypoint 3, located at coordinate (3, 0). The complete traversal executed the following 16-step contiguous topological sequence:

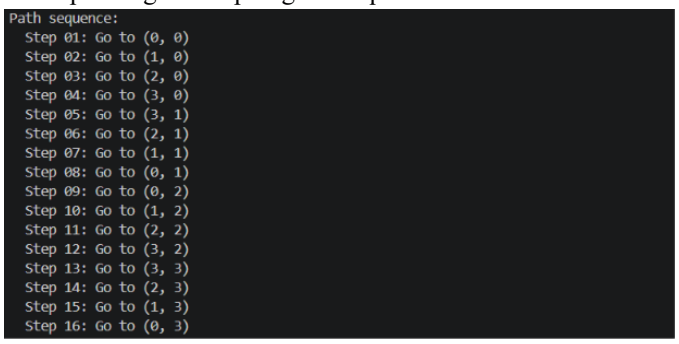

Fig. 8. Path Sequence of 4x4 grid with 3 waypoints

This explicit coordinate sequence provides granular empirical proof that the algorithm strictly adheres to the game's fundamental constraints. Every discrete transition in the sequence respects the orthogonal adjacency limit, absolutely no coordinates are duplicated, and the mandated waypoints are seamlessly integrated in strictly ascending order. Notably, the sequence also confirms that the path covers all sixteen cells of the grid exactly once, satisfying the exhaustiveness requirement defined in Section III without requiring any additional post-processing or validation step.

A corresponding 25-step sequence was similarly successfully generated for the 5×5 configuration, further reinforcing the consistency of the solver across differing grid dimensions and waypoint densities. As the grid size increased from 4×4 to 5×5, the resulting path naturally grew in length while still preserving full adjacency, non-overlap, and ordering correctness, indicating that the algorithm's correctness generalizes beyond the smallest test case.

Whereas the infeasible 6×6 configuration deliberately and correctly returned a null path after comprehensively exhausting the constrained search space, this outcome itself constitutes meaningful empirical evidence: it demonstrates that the solver does not merely terminate prematurely or silently fail when no solution exists, but instead exhausts the entire reachable state space before concluding infeasibility, thereby guaranteeing the correctness of both positive and negative results produced by the engine.

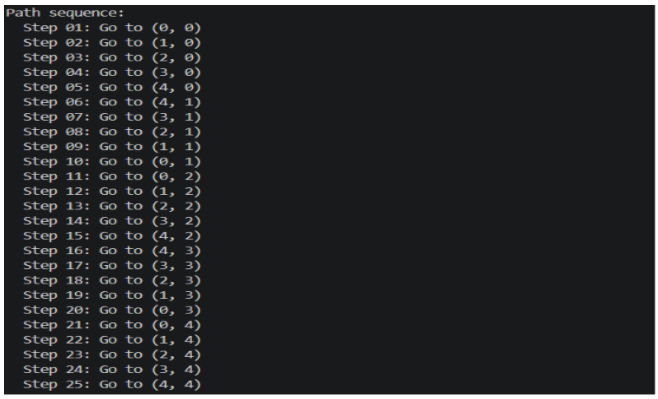

Fig. 9. Path Sequence of 5x5 grid with 4 waypoints

VI. Conclusion

Determining a solution for a grid-based puzzle with strict sequential constraints like LinkedIn Zip can be very difficult to accomplish manually, especially as the size of the grid grows larger. The search space expands super-exponentially, making direct enumeration of paths impractical and motivating the need for a systematic, automated approach to solve this kind of problem reliably.

In this paper, we explore how graph theory, state-space trees, and the DFS-backtracking algorithm can be used to model and solve this problem. The Zip puzzle is modeled as a Hamiltonian path problem on an undirected grid graph, with an additional waypoint-ordering constraint that must be satisfied throughout the traversal. This constraint is then exploited as the basis of the pruning mechanism, where any branch of the search that violates the required waypoint order can be discarded immediately without further exploration.

The solver built upon this approach was shown to correctly find solutions for both the 4×4 and 5×5 feasible configurations, and consistently identified the infeasible 6×6 configuration after exploring more than five million candidate states. These results show that although the worst-case time complexity remains exponential, the pruning mechanism applied significantly reduces the number of branches that need to be explored, while the space complexity remains bounded at $O(n^2)$ thanks to the in-place update-and-restore pattern used throughout the search.

From these results, we can conclude that a DFS-backtracking approach that takes advantage of a problem's specific structure in this case, the waypoint-ordering constraint is an effective strategy for solving a combinatorial problem that is fundamentally NP-complete, at least for the grid sizes commonly found in daily Zip puzzle instances. This also reinforces the idea that exploiting domain-specific constraints is far more efficient than relying on blind, unconstrained search.

As for future work, this research can be extended by incorporating additional heuristics, such as ordering moves based on the number of remaining choices, to further speed up the search process on larger grids. In addition, this solver could also be developed to support other puzzle variants, such as grids with obstacle cells or multiple separate waypoint chains, broadening its applicability beyond the standard LinkedIn Zip format.

ATTACHMENT

- **GitHub Repository :** mobyy17/Makalah-Matematika-Diskrit
- **Youtube Video** : https://youtu.be/GYSBVS_mgNQ?si=6IoK2s16O_ihBmI8

ACKNOWLEDGMENT

First and foremost, the author extends profound gratitude to God Almighty, whose continuous grace and guidance made the completion of this research possible. The author is equally deeply indebted to Dr. Ir. Rinaldi Munir, M.T., the esteemed lecturer for the IF1220 Discrete Mathematics course. His comprehensive lectures and invaluable course materials on graph theory and discrete structures served as the essential theoretical bedrock for the algorithms modeled in this study.

REFERENCES

[1] LinkedIn Help Center, "Play Zip game on LinkedIn," linkedin.com/help/linkedin/answer/a7445030, 2024.

[2] NetInfluencer, "LinkedIn Bets On Games To Build Professional Connections, Daily Habits," 2025.

[3] R. J. Wilson, Introduction to Graph Theory, 4th ed., Harlow: Longman, 1996.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.

[5] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed., Pearson, 2020.

[6] A. Sari et al., "An Implementation of Backtracking Algorithm for Solving a Sudoku Puzzle," J. Phys.: Conf. Ser., vol. 1566, 2020.

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, 19 Juni 2025

Muhammad Rafiif Ansyadya , 13525037