Revenge of The Fallen

Technical Report

Farah Al-Nassar, Aya Jouni, Rita Merhi

Abstract— This report discusses the implementation of a state-space search agent for programmable matter simulation developed for the COE 544/744 Intelligent Engineering Algorithms course at the Lebanese American University. The agent simulates programmable matter elements on a two-dimensional grid that can reconfigure themselves to form predefined shapes while optimizing for minimal movement. Multiple search algorithms were implemented and compared, including Breadth-First Search (BFS), A* search, and a Greedy approach. The report details the PEAS conceptual model, the environment design, the algorithms' implementation, and experimental results comparing their performance across different scenarios and grid sizes. The results demonstrate that A* search provides the most optimal paths for element movements while maintaining reasonable computational efficiency for practical applications.

Index Terms—Robotics, state space search, intelligent agents

CONTENTS

1 INTRODUCTION

Programmable matter refers to materials whose physical properties can be programmatically changed to achieve specific

2 BACKGROUND

Programmable matter is a category of materials that can change their physical properties (shape, density, optical properties, etc.) in a controlled manner through the application of external stimuli such as electric or magnetic fields, temperature, or computational commands. The concept was first proposed in the 1990s and has since become an active research area in materials science, robotics, and computer science.
Current approaches to programmable matter include:

- **Modular Robotic Systems**: Composed of many small, identical robotic modules that can rearrange themselves to form different structures.
- **Colloidal Systems**: Particles suspended in a medium that can be controlled to form specific patterns.
- **Meta-materials**: Materials engineered to have properties not found in naturally occurring materials, often through structural rather than chemical changes.

In our simulation, we focus on a simplified model like modular robotic systems, where each element can move independently on a grid. The coordination of these elements to form coherent shapes presents significant algorithmic challenges in path planning and collision avoidance.
Previous research in this area includes work by Zhu et al. [1] on distributed and parallel control mechanisms for self-reconfiguration of modular robots using L-systems and cellular automata. Similarly, Støy [2] explored the use of cellular automata and gradients to control self-reconfiguration. These approaches emphasize distributed decision-making, where each module operates with only partial knowledge of its environment.
Our project builds upon these foundations but focuses on exploring central coordination strategies using state-space search algorithms to find optimal or near-optimal paths for programmable matter elements to form target shapes.

3 Proposal

3.1 Concepts and Building Blocks

Our project, "Transformers: Revenge of The Fallen," implements a state-space search agent for programmable matter on a two-dimensional grid with an immersive visualization system. The agent coordinates the movements of multiple elements to form predefined shapes such as squares, circles, triangles, or hearts while optimizing for minimal movement.

Using the PEAS (Performance, Environment, Actuators, Sensors) conceptual model for our agent is as follows:

- **Performance Measure:** the agent aims to minimize the total number of movements required to form the target shape.
- **Environment:** a two-dimensional grid where each cell can be empty, contain a wall (obstacle), contain a programmable matter element, or be marked as a target position for the desired shape. The grid can have boundaries that limit movement.
- **Actuators:** each programmable matter element can move in four cardinal directions (up, down, left, right) in Von Neumann topology, or in eight directions (including diagonals) in Moore topology, within the grid, provided the destination cell is empty.
- **Sensors:** the agent has full visibility of the grid, including the positions of all elements, walls, and target positions. It can detect collisions and identify valid moves for each element.

Other design choices we considered are the following:

- **Topology:** Von Neumann (4-way movement) vs. Moore (8-way movement)
- **Decision making:** centralized (global control) vs. distributed (local control)
- **Movement strategy:** sequential vs. parallel movement
- **Pathfinding algorithms:** A*, Breadth-First Search, Greedy

The user can choose any combination between the choices presented in each of these design choices.

3.2 Implementation

The implementation follows a client-server architecture with:

1. **Backend Engine:** Handles the state-space search algorithms, grid environment, and element control logic.
2. **Frontend Interface:** Provides an interactive visualization of the grid, elements, and transformation process with a Transformers-inspired theme.

This design allows users to observe the transformation process in real-time while offering control over various parameters of the simulation.

The state-space search problem is defined as follows:

- **States:** Each state represents the positions of all programmable matter elements on the grid.
- **Initial State:** The initial configuration of elements on the grid.
- **Goal State:** A configuration where elements occupy all target positions defining the desired shape.
- **Actions:** Moving an element in one of the four cardinal directions (up, down, left, right) or eight directions if using Moore topology.
- **Transition Model:** An element can move to an adjacent cell if it is empty.
- **Path Cost:** The number of moves required to transform the initial state into the goal state. While a naive approach might consider the full state space (all possible configurations of all elements), this would lead to a combinatorial explosion. Instead, we decompose the problem into finding paths for individual elements to their assigned target positions, while considering the positions of other elements as obstacles.

I. Backend Architecture

The backend of the system consists of several core components organized in a modular fashion:

1. **Grid Environment:** The Grid class represents the two-dimensional environment where the programmable matter elements operate. It maintains the state of each cell (empty, wall, element, or target) and provides methods for validating positions, checking for occupancy, and retrieving neighboring cells based on the selected topology (Von Neumann or Moore).
2. **Element Management:** The Element class represents individual programmable matter elements with properties such as position and target destination. The ElementController class manages collections of elements, handling their creation, movement, and target assignment.
3. **Shape Generation:** The ShapeGenerator implements algorithms for creating target positions that form different predefined shapes (square, circle, triangle, heart). It ensures that the generated shapes fit within the grid boundaries and adjusts their size based on the number of available elements.
4. **Path Planning:** Separate modules implement the BFS, A*, and Greedy pathfinding algorithms, each receiving the grid environment, start position, and goal position as inputs and returning a computed path.

5. **Simulation Controller:** The ProgrammableMatterSimulation class orchestrates the entire process, initializing the environment, managing elements, invoking the selected search algorithm, and executing the resulting paths. The backend uses a matrix-based representation of the grid, with integer values representing different cell states. Elements are tracked using dictionaries mapping element IDs to their corresponding objects, facilitating efficient lookup and updates during simulation.

II. Pathfinding Algorithms

For the agent to determine the path to take, we used different pathfinding algorithms and compared their time and space complexity.

- **A*:** Combines the advantages of greedy best-first search and Dijkstra's algorithm by using a heuristic function to guide the search toward the goal. Our implementation of the A* algorithm finds the shortest path in a grid environment using a priority queue (heapq) and the Manhattan distance heuristic. It maintains g_score (cost from the start), h_score (heuristic estimate to the goal), and f_score = g + h to guide the search. The algorithm initializes with a start position, checks validity, and explores possible moves, updating scores when a better path is found. A parent dictionary reconstructs the optimal path upon reaching the goal. To handle multiple moving elements, astar_multi_element assigns targets, sorts elements by distance, and iteratively applies A* while temporarily removing obstacles to ensure pathfinding efficiency.
- **Breadth-First Search (BFS):** the BFS algorithm finds the shortest path in a grid environment by exploring nodes in increasing order of depth using a queue (deque). It begins by validating the start and goal positions, then enqueues the starting node and explores possible moves, ensuring each visited position is valid, not a wall, and unoccupied. A parent dictionary records the path, which is reconstructed upon reaching the goal. The bfs_multi_element function extends this to multiple moving elements by assigning targets, temporarily removing obstacles, and sequentially applying BFS for each element. While BFS guarantees the shortest path in an unweighted grid, it can be inefficient in large spaces compared to heuristic-based approaches like A*.
- **Greedy:** A more efficient but potentially suboptimal algorithm that always moves toward the goal based on a heuristic function. The Greedy Best-First Search algorithm finds a path in a grid environment using the Manhattan distance heuristic to prioritize nodes closest to the goal. It begins by validating the start and goal positions, ensuring they are not walls, and checking if the start already matches the goal. The algorithm expands nodes in the direction that minimizes the estimated distance to the target, without considering the cost of reaching them. The greedy_multi_element function applies this approach to multiple moving elements by sorting them based on proximity to their targets and temporarily removing obstacles during pathfinding. While this method is computationally efficient, it may not always find the shortest path due to its tendency to prioritize immediate progress over optimal routing.

This table summarizes the time and space complexity of each of the previous algorithms:

Algorithm	Time Complexity	Space Complexity	Guarantees Shortest Path?
A*	$Between\ O(b^d)\ and\ O(d)$	$O(b^d)$	Yes, with admissible heuristic
BFS	$O(b^d)$	$O(b^d)$	Yes, for unweighted graphs
Greedy Best-First	$O(b^d)$	$O(b^d)$	No, it may take suboptimal paths

Where b is the branching factor, in other words the number of possible moves per node, and d is the depth of the optimal solution.

Note that for A*, the **heuristic function** we used is the **Manhattan distance**:

$$h(x, y) = |\ x1 - x2\ | + |\ y1 - y2\ |$$

With this heuristic function, A* significantly reduces unnecessary explorations compared to uninformed searches like BFS, making it more efficient in practice.

Thus, the worst-case time complexity remains O(b^d), but in practical scenarios, it is much faster due to heuristic guidance. In an open grid where every node is accessible, the number of expanded nodes can be closer to O(n), where n is the total number of grid cells.

III. Decision Making

These algorithms were implemented for both individual element pathfinding and for coordinating multiple elements. When dealing with multiple elements, we implemented two control modes:

1. **Centralized Decision Making:** A central controller plans and executes paths for all elements sequentially, temporarily removing elements from the grid during pathfinding and replacing them afterward.
2. **Distributed Decision Making:** Each element makes its own decisions based on its current position and target,

with conflict resolution for simultaneous movements to the same cell.

IV. Movement Strategy

The movement execution could be done in two ways:

1. **Sequential Movement:** Elements move one at a time, in order of their distance to target.
2. **Parallel Movement:** Elements attempt to move simultaneously, with conflicts resolved by priority rules.

IV. Algorithm Flow

The execution flow of the search algorithms begins with initializing the grid and elements in their starting positions. The user selects a target shape and algorithm parameters (A*, BFS, or Greedy). The system then assigns targets to elements using a closest available strategy. For each element, the selected pathfinding algorithm computes an optimal path to its target. In sequential mode, elements move one at a time following their paths, while in parallel mode, multiple elements attempt to move simultaneously with conflict resolution. The process continues until all elements reach their targets or until no further progress can be made.

4 Experimental Evaluation

4.1 Test Protocol

To evaluate the performance of our search algorithms, we conducted experiments with different grid sizes, obstacle configurations, and target shapes. We measured the following metrics:

1. **Total moves:** The total number of moves required to form the target shape.
2. **Execution time:** The time taken by each algorithm to compute the paths.
3. **Nodes explored:** The number of grid cells explored during the search process.

4.2 Test Data

For our experiments, we used the following configurations:

- Grid size: 10×10
- Target shape: Square
- Number of elements: 1-20
- Topology: Von Neumann and Moore
- Control modes: Centralized and Distributed
- Movement types: Sequential and Parallel

4.3 Experimental Results

We found that Moore topology reduces total moves but increases computational cost due to the larger branching factor in search. Also, A* provides the best balance between path optimality and computational efficiency. The results are compared in this table:

Algorithm	Topology	Execution Time (s)	Total Moves	Nodes Explored
A*	Von Neumann	0.141	63	116
A*	Moore	0.187	47	66
BFS	Von Neumann	0.136	61	236
BFS	Moore	0.182	47	256
Greedy	Von Neumann	0.144	61	70
Greedy	Moore	0.189	47	58

5. Conclusion

5.1 Synthesis

In this project, we successfully developed "Transformers: Revenge of The Fallen," a programmable matter search agent that efficiently reconfigures elements to form predefined shapes while providing an engaging visual representation of the process. Our implementation demonstrates the application of classical search algorithms to the domain of programmable matter, with a focus on optimizing movement efficiency.

5.2 Future Work

Several directions for future work can be explored:

1. **Multi-Objective Optimization:** Extending the algorithms to consider multiple objectives beyond minimizing

total moves, such as completion time.
2. **Obstacles:** Enhancing the environment to include obstacles
3. **Learning-Based Approaches:** Exploring reinforcement learning for adaptive path planning and target assignment.
4. **Enhanced Visualization:** Further improving the frontend interface with additional metrics, animation options, and interactive features.

References

[1] Yanhe Zhu, Dongyang Bie, Xiaolu Wang, Yu Zhang, HongZhe Jin, Jie Zhao: A distributed and parallel control mechanism for self-reconfiguration of modular robots using L-systems and cellular automata. J. Parallel Distributed Comput. 102: 80-90 (2017)
[2] Kasper Støy, Using cellular automata and gradients to control self-reconfiguration. Robotics Auton. Syst. 54(2): 135-141 (2006)