Multi-Agent Planner for Chessboard Navigation

This report presents the development and implementation of a sophisticated multi-agent pathfinding (MAPF) system, designed to navigate multiple agents—specifically kings on a chessboard—through a complex environment without collisions. The project leverages advanced algorithmic techniques within a constrained computational framework to address the challenge of sequential agent movements on a grid with restricted areas. Utilizing a modified A* search algorithm, extended to include temporal and spatial dimensions (space-time A*), and integrating Conflict-Based Search (CBS) for conflict resolution, this system ensures optimal and collision-free navigation.

The system handles the intricacies of MAPF by enforcing strict movement rules that prevent agents from moving into restricted cells and from ending movements adjacent to each other. The core of the solution involves dynamic constraint generation and management, facilitating the detection and resolution of both vertex and edge type collisions.

Extensive testing on various scenarios confirms the robustness and efficiency of the planner. The algorithm successfully computes viable paths for all agents or accurately detects scenarios where no path exists. This project not only showcases a technical solution to a complex problem but also contributes to the broader field of robotics and automated systems by providing a scalable model for multi-agent navigation in constrained environments.

Algorithm Design and Approach:

The core of my solution utilizes a space-time A* search algorithm, modified to incorporate constraints that prevent collisions both in terms of position and time.

- **Space-Time A* Algorithm**: This involved modifying the traditional A* algorithm to consider time as a dimension, allowing the planner to handle dynamic constraints across different timesteps.
- **Conflict-Based Search (CBS)**: I further enhanced the solution by integrating CBS, an approach that is optimal and complete for solving multi-agent path finding problems. CBS works by detecting conflicts among paths and resolving them by imposing constraints that split the search space. The algorithm mainly identifies two types of collisions:

- **“Vertex Type”**: A vertex type collision occurs when two or more agents occupy the same cell at the same timestep. For example, consider a scenario where:
  - King A is supposed to move to cell (x, y) at timestep t.
  - Simultaneously, Agent B also moves to cell (x, y) at timestep t.

  This results in a vertex collision at cell (x, y) at timestep t, which must be resolved to ensure that the agents can proceed without interference.

- **“Edge type”**: Edge type collisions, also known as swap collisions, occur when two agents attempt to traverse the edge between two cells in opposite directions simultaneously. In essence, they "pass" each other in the grid, resulting in a collision. For instance:
  - Agent A moves from cell (x, y) to cell (x+1, y) at timestep t.
  - At the same time, Agent B moves from cell (x+1, y) to cell (x, y) at timestep t.

  This results in an edge collision between the paths of Agent A and Agent B between timesteps $t-1$ and t, and it must be resolved to avoid a conflict in the paths.

Managing Collisions:

To manage these collisions, strategies like CBS use a two-level search. The high level of the search builds a constraint tree where each node represents a set of constraints derived from detected collisions. When a collision is detected, it is resolved by creating constraints that prevent at least one of the conflicting moves, thereby branching into new nodes in the search tree.

For vertex collisions, the constraint might involve preventing one or more agents from entering the conflicting cell at the specific timestep. For edge collisions, the constraint might prevent one or both of the agents from making their respective moves at the specified timestep.

Both types of collisions and their resolutions are central to ensuring that the path-finding algorithm can find viable paths for all agents without interference, thus guaranteeing that the solution is not only feasible but also optimal in terms of path length and time.

Implementation Details:

The implementation was divided into several Python modules:

- main.py: Orchestrates the overall execution and interaction between different components.
- utils.py: Provides utility functions such as heuristic calculations and constraint management.
- low_level_planner.py and main_planner.py: Handle the core logic of the A* algorithm modified for space-time search and the CBS strategy, respectively.
- animation.py: Optional component used for visualizing the movements of the kings on the chessboard.

Running the Multi-Agent Pathfinding Code:

To operate the code:

- Navigate to the directory containing the code.
- Install required packages using: **pip install -r requirements.txt**
- Execute: **python main.py**
- Modify the path of the files to run the code on different test cases.

Results and Conclusions

The developed planner successfully met the requirements outlined in the problem statement. It was able to compute paths for the kings or detect the absence of viable paths (if present) within the stipulated time frame.

This project not only highlighted my technical skills in algorithm design and optimization but also reinforced my ability to apply theoretical concepts to practical, real-world problems.