Group Assignment IA Report

Development of an autonomous agent for the game Snake

Martim Santos (114614), Gonçalo Sousa (108133), João Roldão (113920)

Agent Architecture

Movement

Purpose: Central decision-making module.

Implementation Details:

- Checks for loops (via detect_loop) and breaks them if found.
- Depending on the snake's state calls one of the specialized methods:
  - *avoid_danger(...)*
  - *navigate_to_food(...)*
  - *explore(...)*
- Danger Avoidance:
  - Uses BFS-based distance checks plus connected-component checks to ensure enough space for the snake's body.
- Food Navigation:
  - Finds the closest safe food tile by BFS distance.
  - Verifies whether the tile is in a safe zone (component size ≥ snake length).
- Exploration:
  - If there is no danger or food, tries to move into less-visited tiles to expand knowledge.
  - Picks the tile with the highest "time since visit" within BFS reach.

Removed or Simplified:

- We used to have additional heuristics for looping and fallback. We pruned them to simpler BFS distance checks. This was because when we capped our processor we felt the fell off was too big so we simplified the code.

Agent Architecture

MapKnowledge

Purpose: Holds the snake's mental representation of the map.
Implementation Details:

- Maintains a 2D array of tiles, each storing *(tile_type, last_visit_step)*.
- Updates the map as the snake gains vision information.
- Uses BFS to compute layers/distance grids *(compute_bfs_layers(...))*, labeling each tile with a BFS distance from the snake's head.
- Performs component analysis to identify connected areas, storing each cell's component ID and size.
- Collision Caching: Uses a cache *(collision_cache)* to quickly check collisions and reduce recomputation.

Removed or Simplified:

- We previously had more specialized heuristics and toggles for certain tile updates. Over time, we streamlined the BFS approach *(compute_bfs_layers)*, and rely primarily on it for distance checks.

StateManager

Purpose: Tracks the snake's "state" as either DANGER, TARGETING, or SAFE, based on environment conditions.
Implementation Details:

- *evaluate_state(...)* checks if there is any immediate danger (walls, self collision, or other snake collision), or if food is available.
- The chosen state influences how Movement decides a path:
    - DANGER → Avoid collisions using fallback or BFS to safe spots.
    - TARGETING → Navigate to visible food.
    - SAFE → Explore unknown areas of the map.

Algorithm Explanation

State-Driven Behavior

Our Movement class's *decide_move(...)* runs the core logic:

- **Compute BFS Layers:** We run a BFS from our head to get a distance_grid, labeling each tile with how many steps away it is, or -1 if unreachable.
- **Detect Loops**: Checks the last positions in head_history; if repeated patterns appear, we do a loop-break fallback.
- **Evaluate Danger/Food**: We consult StateManager to set the state:
    - DANGER → avoid_danger(...)
    - TARGETING → navigate_to_food(...)
    - SAFE →explore(...)..

Loop Handling

Tracks movement and position history to identify repeated patterns. Employs BFS or fallback directions to escape the loop and maintain progress.

Avoid Danger

Evaluating its surroundings using MapKnowledge the snake identifies immediate obstacles like, walls, stones or itself or other snakes but it also uses BFS to compute safe zones and chooses the zone that has more space available and multiple exits to avoid potential dead ends or trapping itself.

Algorithm Explanation

Navigate To Food

When food is visible the first thing it does is to verify if food is in a safe area, ensuring that the snake can reach it without becoming trapped; to do this it incorporates component analysis from MapKnowledge.
For pathfinding it uses BFS to calculate the shortest and safest route to the food, dynamically adjusting to new visible food and changes in the map.
If no safe food is found it reverts to exploring.

Explore

In our explore method we use a simple strategy where we focus on getting to areas that aren't visited for the longest time and use this with the snake's range of sight to determine areas that are the best for exploring, the ones with high density of unexplored tiles. We balance this method by checking the level of danger in those zones as well.

Multiplayer

For the multiplayer the snake, using only its sight, tries to find another snake when exploring and if it does it intercepts it, if not it just keeps going.

Benchmark Results

To evaluate our snake's performance, we decided to play 10 single player matches and calculate the average score. After these 10 matches, our snake achieved an average score of 105, which we consider satisfactory compared to our first submission.
As for the testing environment, we limited our machine's processor to 2GHz so that our results would more closely match the professor's.
As the professor's server is currently unavailable, we decided to test the multiplayer mode locally by having our snake face off against itself. Since we used two snakes with identical configurations, we find the results inconclusive.

Conclusion

Our agent effectively uses MapKnowledge to maintain a BFS-based distance grid each turn, combined with StateManager for high-level logic (Danger, Targeting, or Safe). The Movement module orchestrates the actual decision, ensuring the snake:

- Avoids loops via repeated position checks;
- Navigates around immediate threats with BFS;
- Prioritizes reachable food and safe;
- Explores unvisited tiles efficiently;

The overall result is a codebase that is maintainable, with clearly separated roles among modules, and a BFS-driven approach ensuring the snake can adapt to changing map conditions quickly.