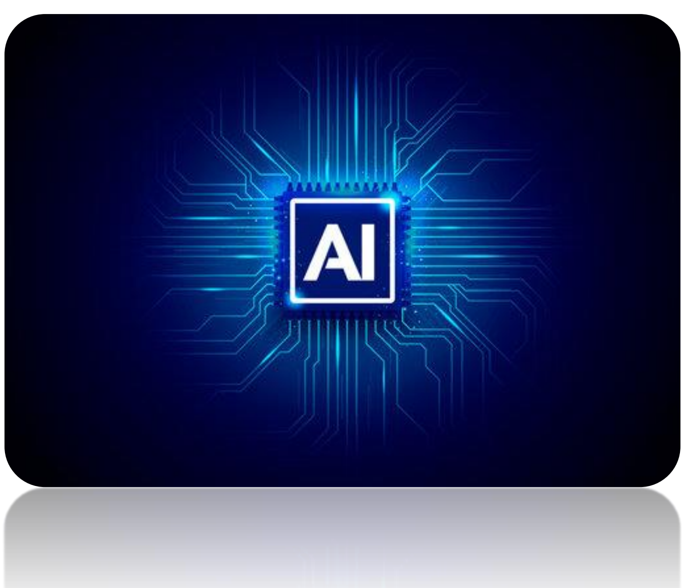
Artificial Intelligence Project Report

Dr. Shahira

Eng. Nour

Dynamic Threat-Aware A*

Students Names
Mohamed Salem Mohamed Soliman
Moslem Sayed Shehata Ali
Mina Magdy Ezzat Ramzy
Abdulrahman Sobhy Hanafy
Hossam Eldin Omar Mohamed Sayed



1. Overview

Dynamic Threat-Aware A*

The Dynamic Threat Aware A* Agent is an intelligent agent for the Pac-Man game that uses A* search with dynamic heuristics to balance three primary objectives:

- Food Collection: Finding and consuming food pellets efficiently
- Ghost Avoidance: Maintaining a safe distance from ghosts
- Optimal Path Planning: Minimizing the total distance traveled

The agent can be configured with different parameters to adjust its behavior, making it adaptable to various game scenarios and player preferences.

2. Code Approach

First, choosing the version of Pac-Man that we will be working on and we choose Multiagent.

Second, adding our custom class DynamicThreatAwareAStarAgent and start implementing the class.

Third, the agent uses three main parameters to control its behavior:

Parameter	Default	Purpose
BETA	10	Weight for ghost threat avoidance
K Steps	3	Threshold distance to dynamically adjust beta
Food Weight	10	Weight for food importance in path planning

These parameters allow for significant customization:

- Setting BETA = 0 creates a food-focused agent that ignores ghosts entirely.
- Higher BETA values make the agent more cautious around ghosts.
- The K Steps parameter creates a dynamic response zone around ghosts.
- Higher Food Weight values prioritize food collection over other objectives.

Core Components:

- Ghost Threat Mapping: Using multi-source BFS to identify dangerous areas
- Food-Aware Heuristic: Balancing goal distance, food rewards, and ghost threats
- Dynamic Beta Adjustment: Creating "danger zones" around ghosts
- Intelligent Target Selection: Choosing the best food pellet or safe corner

3. Implementation

3.1. Ghost Threat Mapping

The agent maintains a comprehensive threat map of the game environment by performing multi-source breadth-first search (BFS) from all ghost positions:

```
def computeGhostThreatMap(self, gameState):
    """
    Perform multi-source BFS from all ghost positions to compute threat map.
    Returns a dictionary mapping positions to distance from closest ghost.
    """
    ghost_states = gameState.getGhostStates()
    ghost_positions = [ghost.getPosition() for ghost in ghost_states]

    threat_map = {}
    width, height = self.walls.width, self.walls.height

    fringe = Queue()
    for pos in ghost_positions:
        pos = (int(pos[0]), int(pos[1]))
        fringe.push((pos, 0))
        threat_map[pos] = 0

    while not fringe.isEmpty():
        (pos, dist) = fringe.pop()
        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            next_pos = (pos[0] + dx, pos[1] + dy)
            x, y = next_pos
            if 0 <= x < width and 0 <= y < height and not self.walls[x][y] and
next_pos not in threat_map:
                threat_map[next_pos] = dist + 1
                fringe.push((next_pos, dist + 1))

    return threat_map
```

This approach efficiently calculates the minimum distance from each position to any ghost, which is essential for threat assessment.

3.2. Food-Aware Heuristic

The agent uses a sophisticated heuristic function that combines distance to goal, food incentives, and ghost threats:

```
def threatHeuristic(self, current_pos, goal_pos, gameState, threat_map, beta):
    """
    Compute the threat-aware heuristic with food consideration:
    h(n) = dist_to_goal(n) + beta / (1 + dist_to_ghost(n)) - food_weight *
food_value(n)

    If beta is zero, ghosts are completely ignored for efficiency.
    """
    dist_to_goal = manhattanDistance(current_pos, goal_pos)

    food_grid = gameState.getFood()
    food_value = 0

    if current_pos[0] < food_grid.width and current_pos[1] < food_grid.height and
food_grid[current_pos[0]][current_pos[1]]:
        food_value += 1

    closest_food_dist = self.getClosestFoodDistance(current_pos, food_grid)
    if closest_food_dist < float('inf'):
        food_value += 1.0 / (1 + closest_food_dist)

    if beta == 0:
        return dist_to_goal - self.food_weight * food_value

    dist_to_ghost = self.getGhostDistance(current_pos, threat_map)

    if dist_to_ghost <= 1:
        return float('inf')

    threat_component = beta / (1 + dist_to_ghost)

    return dist_to_goal + threat_component - self.food_weight * food_value
```

This heuristic can be broken down into three components:

1. **Goal Distance**: Manhattan distance to the target location
2. **Food Component**: Reward for positions with food or near food
3. **Ghost Threat**: Penalty that increases as ghosts get closer

Note that when beta = 0, the ghost threat component is disabled entirely for performance optimization.

3. 3. Dynamic Beta Adjustment

The agent can adjust its ghost avoidance behavior dynamically based on proximity:

```
def adjustBeta(self, current_pos, threat_map):
    """
    Dynamically adjust beta based on ghost proximity.
    Increases beta when ghosts are within k_steps.
    """
    if self.beta == 0:
        return 0
    dist_to_ghost = self.getGhostDistance(current_pos, threat_map)

    if dist_to_ghost <= self.k_steps:
        self.last_beta = self.beta * (self.k_steps / max(1, dist_to_ghost))
        return self.last_beta
    else:
        self.last_beta = self.beta
        return self.beta
```

This creates a "danger zone" around ghosts where the agent becomes increasingly cautious as it gets closer, simulating a more realistic threat response.

3. 4. Intelligent Target Selection

The agent uses a sophisticated algorithm to select the best target (food pellet or safe corner):

```
def findBestTarget(self, gameState, threat_map):
    """
    Find the best target considering food locations and safe corners.
    Prioritizes food locations but considers safety from ghosts.
    If beta is zero, simply targets the closest food.
    """
    pacman_pos = gameState.getPacmanPosition()
    food_grid = gameState.getFood()
    food_list = food_grid.asList()

    if not food_list:
        return self.findSafestCorner(gameState, threat_map)

    if self.beta == 0:
        closest_food = None
        min_distance = float('inf')

        for food_pos in food_list:
            dist = manhattanDistance(pacman_pos, food_pos)
            if dist < min_distance:
                min_distance = dist
```

```python
                closest_food = food_pos

        return closest_food

    best_target = None
    best_score = float('-inf')

    for food_pos in food_list:
        dist_to_food = manhattanDistance(pacman_pos, food_pos)
        dist_to_ghost = self.getGhostDistance(food_pos, threat_map)

        score = -dist_to_food + 0.5 * dist_to_ghost

        if dist_to_ghost <= 2:
            score -= 100

        if score > best_score:
            best_score = score
            best_target = food_pos

    if best_target is None or best_score < -50:
        return self.findSafestCorner(gameState, threat_map)

    return best_target
```

The target selection logic has two modes:

1. When beta = 0: Simply select the closest food
2. When beta > 0: Score each food based on distance to Pacman and safety from ghosts

If all food pellets are too dangerous, the agent falls back to finding a safe corner using the findSafestCorner method.

3. 5. A* Path Planning

The agent uses the A* search algorithm to find optimal paths to selected targets:

```
def aStarSearch(self, gameState, goal_pos):
    """
    A* search with the threat-aware and food-aware heuristic.
    Optimized to skip ghost calculations when beta is zero.
    """
    start_pos = gameState.getPacmanPosition()

    threat_map = {}
    if self.beta > 0:
        threat_map = self.computeGhostThreatMap(gameState)

    fringe = PriorityQueue()
    closed_set = set()

    fringe.push((start_pos, []), 0)
    g_scores = {start_pos: 0}

    while not fringe.isEmpty():
        (current_pos, path) = fringe.pop()

        if current_pos == goal_pos:
            return path

        if current_pos in closed_set:
            continue

        closed_set.add(current_pos)

        beta = 0
        if self.beta > 0:
            beta = self.adjustBeta(current_pos, threat_map)

        x, y = current_pos
        for dx, dy, action in [(0, 1, Directions.NORTH), (1, 0, Directions.EAST),
                               (0, -1, Directions.SOUTH), (-1, 0,
Directions.WEST)]:
            next_pos = (x + dx, y + dy)

            if not gameState.hasWall(next_pos[0], next_pos[1]) and next_pos not
in closed_set:

                tentative_g = g_scores[current_pos] + 1

                if next_pos not in g_scores or tentative_g < g_scores[next_pos]:
                    g_scores[next_pos] = tentative_g
```

```
                    f_score = tentative_g + self.threatHeuristic(next_pos,
goal_pos, gameState, threat_map, beta)

                    new_path = path + [action]
                    fringe.push((next_pos, new_path), f_score)

    return []
```

The A* implementation includes several optimizations:

1. Ghost-related calculations are skipped when beta = 0
2. A closed set prevents re-exploration of positions
3. The custom heuristic guides search toward food while avoiding ghosts

Performance Optimizations

Several optimizations have been implemented to improve the agent's performance:

1. **Conditional Ghost Calculations**: Ghost threat maps are only computed when necessary
2. **Path Caching**: The A* search returns complete paths to targets
3. **Corner Identification**: Safe corners are identified during initialization
4. **Heuristic Pruning**: Positions too close to ghosts are immediately assigned infinite cost
5. **Beta = 0 Fast Path**: Special optimized code path when ghosts are ignored

Behavioral Modes

The agent effectively has two distinct modes of operation:

Food-Focused Mode (beta = 0)

- Ignores ghosts completely
- Directly targets the closest food
- Uses simpler heuristic calculations
- Optimized for performance

Balanced Mode (beta > 0)

- Considers both food and ghosts
- Dynamically adjusts ghost avoidance based on proximity
- Uses sophisticated target scoring
- Falls back to safe corners when food is too dangerous

Fallback Mechanisms

The agent includes several fallback mechanisms to handle edge cases:

1. **No Food**: If no food is left, the agent navigates to safe corners
2. **No Path**: If A* fails to find a path, the agent selects a random valid action
3. **Dangerous Food**: If all food is too dangerous, the agent retreats to safe corners
4. **No Good Targets**: If no good targets exist, the agent maximizes distance from ghosts

Conclusion

The DynamicThreatAwareAStarAgent represents a sophisticated approach to Pac-Man gameplay that balances multiple objectives through intelligent path planning and dynamic threat assessment. By adjusting the parameters beta, k_steps, and food_weight, the agent can be tuned for different play styles from aggressive food collection to cautious ghost avoidance.

The implementation demonstrates several key AI concepts:

- Heuristic search with A*
- Multi-objective path planning
- Dynamic threat response
- Fallback strategies for edge cases

This makes the agent both effective at maximizing score and educational as an example of applied artificial intelligence techniques in game environments.