Dynamic Threat-Aware A* for Pacman

This document contains both the implementation of a Dynamic Threat-Aware A* agent for Pacman and instructions on how to integrate and test it.

1. Agent Implementation

Copy the following code into your `multiagent.py` file:

python

```python
class ThreatAwareAStarAgent(Agent):
    """
    A* agent that avoids ghosts dynamically by computing threat zones
    and adjusting its path planning based on ghost proximity.
    """
    def __init__(self, beta=10.0, k_steps=3):
        self.beta = beta  # Weight for ghost threat in heuristic
        self.k_steps = k_steps  # Threshold distance to adjust beta
        self.last_beta = beta  # Track last beta for reporting

    def registerInitialState(self, gameState):
        """
        Initialize agent parameters and compute initial ghost threat map.
        Called once at the beginning of the game.
        """
        self.walls = gameState.getWalls()
        self.start = gameState.getPacmanPosition()
        self.corners = self.findCorners(self.walls)
        self.ghost_threat_map = {}  # Will store distance to closest ghost for each position
        # Initialize move tracking for evaluation
        self.moves_made = 0
        self.path = []
        # Compute initial ghost threat map
        self.ghost_threat_map = self.computeGhostThreatMap(gameState)

    def findCorners(self, walls):
        """Find safe corners in the maze (positions where Pacman can hide)."""
        corners = []
        width, height = walls.width, walls.height
        for x in range(1, width - 1):
            for y in range(1, height - 1):
                # A corner is a position with walls on two adjacent sides
                if not walls[x][y]:  # If this position is not a wall
                    wall_count = 0
                    adjacent_walls = 0
                    for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                        if walls[x + dx][y + dy]:
                            wall_count += 1
                            if wall_count > 0 and (walls[x + dx][y] and walls[x][y + dy]):
                                adjacent_walls += 1
                    if wall_count >= 2 and adjacent_walls > 0:
                        corners.append((x, y))
        return corners

    def computeGhostThreatMap(self, gameState):
```

```
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]

    # Initialize the threat map
    threat_map = {}
    width, height = self.walls.width, self.walls.height

    # Multi-source BFS
    fringe = Queue()
    for pos in ghost_positions:
        # Round ghost positions to nearest integer
        pos = (int(pos[0]), int(pos[1]))
        fringe.push((pos, 0))  # (position, distance)
        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
            # Check if position is valid and not already in threat map
            if 0 <= x < width and 0 <= y < height and not self.walls[x][y] and next_pos not
                threat_map[next_pos] = dist + 1
                fringe.push((next_pos, dist + 1))

    return threat_map

def getGhostDistance(self, pos, threat_map):
    """
    Get the distance to the closest ghost for a given position.
    Returns infinity if position is not in the threat map.
    """
    return threat_map.get(pos, float('inf'))

def threatHeuristic(self, current_pos, goal_pos, threat_map, beta):
    """
    Compute the threat-aware heuristic:
    h(n) = dist_to_goal(n) + beta / (1 + dist_to_ghost(n))
    """
    dist_to_goal = manhattanDistance(current_pos, goal_pos)
    dist_to_ghost = self.getGhostDistance(current_pos, threat_map)
```

```
        # If ghost is too close, return a very high value
        if dist_to_ghost <= 1:
            return float('inf')

        # Compute threat component
        threat_component = beta / (1 + dist_to_ghost)

        return dist_to_goal + threat_component

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

        # Increase beta when ghost is within k_steps
        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

    def findSafestCorner(self, gameState, threat_map):
        """Find the safest corner based on distance from ghosts."""
        pacman_pos = gameState.getPacmanPosition()

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

        for corner in self.corners:
            dist_to_corner = manhattanDistance(pacman_pos, corner)
            dist_to_ghost = self.getGhostDistance(corner, threat_map)

            # Score favors corners far from ghosts but close to Pacman
            score = dist_to_ghost - 0.5 * dist_to_corner

            if score > best_score:
                best_score = score
                best_corner = corner

        # If no good corner found, return a random one
        if best_corner is None and self.corners:
            return random.choice(self.corners)
```

```
    return best_corner

def aStarSearch(self, gameState, goal_pos):
    """
    A* search with the threat-aware heuristic.
    """
    start_pos = gameState.getPacmanPosition()
    threat_map = self.computeGhostThreatMap(gameState)

    # Initialize data structures
    fringe = PriorityQueue()
    closed_set = set()

    # Store (position, path) in the fringe, where path is a list of actions
    fringe.push((start_pos, []), 0)
    g_scores = {start_pos: 0}  # Cost from start to node

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

        # If goal reached, return the path
        if current_pos == goal_pos:
            return path

        # Skip if already explored
        if current_pos in closed_set:
            continue

        closed_set.add(current_pos)

        # Dynamic beta adjustment
        beta = self.adjustBeta(current_pos, threat_map)

        # Generate successors
        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)

            # Check if valid move
            if not gameState.hasWall(next_pos[0], next_pos[1]) and next_pos not in closed_s
                # Tentative g score
                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
```

```
                    # Compute f = g + h
                    f_score = tentative_g + self.threatHeuristic(next_pos, goal_pos, threat

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

        # If no path found, return empty list
        return []

    def getAction(self, gameState):
        """
        Get the next action based on A* search to the safest corner.
        """
        # Update ghost threat map
        threat_map = self.computeGhostThreatMap(gameState)

        # Find safest corner
        goal = self.findSafestCorner(gameState, threat_map)

        if goal is None:
            # If no safe corner is found, just avoid ghosts
            legal = gameState.getLegalPacmanActions()
            pacman_pos = gameState.getPacmanPosition()

            best_action = Directions.STOP
            best_ghost_dist = -1

            for action in legal:
                if action == Directions.STOP:
                    continue

                # Get successor position
                dx, dy = Actions.directionToVector(action)
                next_x, next_y = int(pacman_pos[0] + dx), int(pacman_pos[1] + dy)
                next_pos = (next_x, next_y)

                # Distance to ghost from next position
                ghost_dist = self.getGhostDistance(next_pos, threat_map)

                if ghost_dist > best_ghost_dist:
                    best_ghost_dist = ghost_dist
                    best_action = action

            return best_action

        # Get path to goal
```

```
        path = self.aStarSearch(gameState, goal)

        # If path is empty, choose a random legal action
        if not path:
            legal = gameState.getLegalPacmanActions()
            if Directions.STOP in legal:
                legal.remove(Directions.STOP)
            if legal:
                return random.choice(legal)
            else:
                return Directions.STOP

        # Return first action in the path
        return path[0]


# Add helper class for Directions to vector conversion
class Actions:
    # Directions
    _directions = {Directions.NORTH: (0, 1),
                   Directions.SOUTH: (0, -1),
                   Directions.EAST:  (1, 0),
                   Directions.WEST:  (-1, 0),
                   Directions.STOP:  (0, 0)}

    @staticmethod
    def directionToVector(direction):
        return Actions._directions[direction]
```

2. Integration with Command Line Arguments

After adding the agent class, you need to modify the Pacman project to accept the beta and k_steps parameters. Add these lines to the `readCommand` function in `pacman.py`:

python

```
# Add just before the return statement in readCommand function
if pacman_type == 'ThreatAwareAStarAgent':
    args['beta'] = float(options.beta) if hasattr(options, 'beta') else 10.0
    args['k_steps'] = int(options.ksteps) if hasattr(options, 'ksteps') else 3
```

And add these options to the argument parser:

python

```python
# Add to the optparse section in readCommand
parser.add_option('--beta', dest='beta', type='float', default=10.0,
                  help=default('Beta parameter for ThreatAwareAStarAgent'))
parser.add_option('--ksteps', dest='ksteps', type='int', default=3,
                  help=default('K-steps parameter for ThreatAwareAStarAgent'))
```

3. Testing and Evaluation

Save the following script as `evaluate_threat_aware.py` to evaluate your agent:

python

```python
import os
import sys
import random
import pandas as pd
import argparse
import numpy as np
import matplotlib.pyplot as plt

# Ensure the Pacman modules are in the path
sys.path.append('.')

# Import the Pacman modules (only after adding to path)
from pacman import runGames
from layout import getLayout

# Import the agent
from multiagent import ThreatAwareAStarAgent

def evaluate_agent(layout_name, beta_values, k_steps=3, num_runs=20, ghost_type='DirectionalGho
    """Evaluate the ThreatAwareAStarAgent with different beta values."""
    # Import ghost agents
    if ghost_type == 'RandomGhost':
        from ghostAgents import RandomGhost as GhostClass
    else:
        from ghostAgents import DirectionalGhost as GhostClass

    # Prepare results storage
    results = []

    for beta in beta_values:
        print(f"Evaluating with beta={beta}, k_steps={k_steps}")

        total_score = 0
        total_steps = 0
        win_count = 0

        for i in range(num_runs):
            # Create the agent
            pacman_agent = ThreatAwareAStarAgent(beta=beta, k_steps=k_steps)

            # Create ghost agents
            ghost_agents = [GhostClass(i+1) for i in range(4)]

            # Create the layout
```

```
layout = getLayout(layout_name)
if layout is None:
    raise Exception(f"The layout {layout_name} cannot be found")

# Run a game
from textDisplay import NullGraphics
games = runGames(layout, pacman_agent, ghost_agents, NullGraphics(), 1, False)
game =
```