REPORT UTS
PROJECT REINFORCEMENT LEARNING

Disusun oleh:

Kelompok 3

1. Rimaya Dwi Atika (G1A021021)
2. Muhamad Rifqi Afriansyah (G1A021023)
3. Ilham Dio Putra (G1A021024)

Dosen Mata Kuliah:

Arie Vatresia, S.T., M.T.I., Ph.D

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS BENGKULU

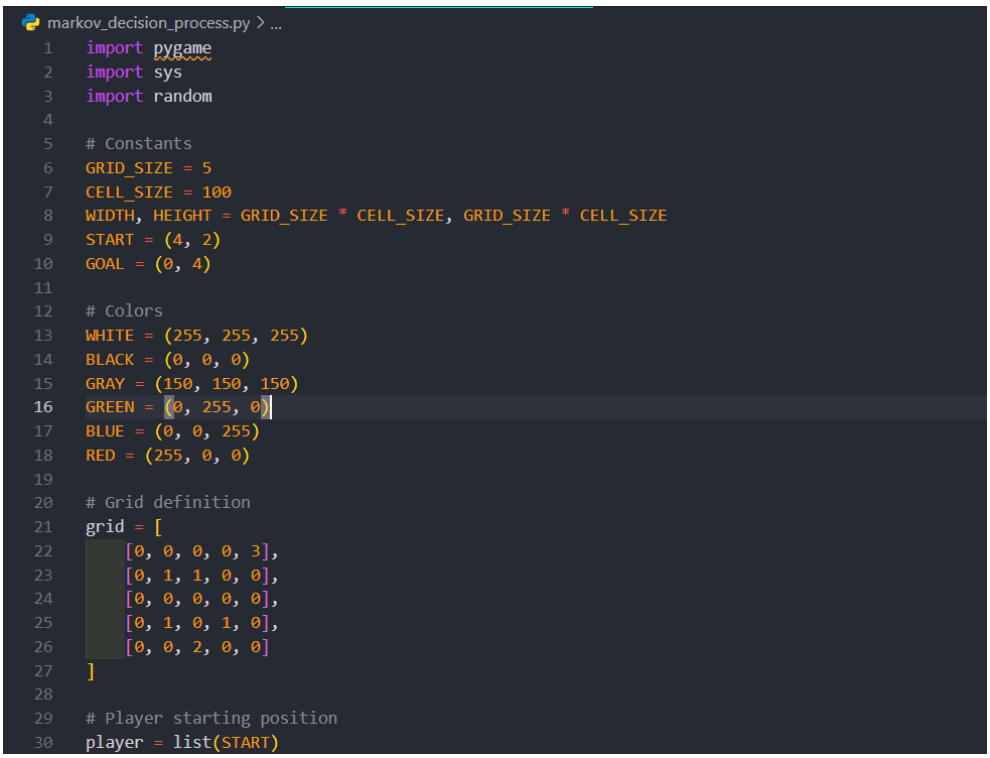
2024

Perbandingan Game Grid World Menggunakan Markov Decision Process (MDP), Policy Bellman Equation, dan Q-Function Value Iteration

Markov Decision Process

Materi : https://github.com/brianspiering/rl-course/blob/main/02_policy__value_iteration/1_markov_decision_process_intro.ipynb

- **markov_decision_process.py**



Penjelasan:

Pada kode ini, kita mengimpor pustaka yang diperlukan, seperti pygame, sys, dan random. Lalu, kita menetapkan beberapa konstanta penting, seperti ukuran grid, ukuran sel, warna-warna yang akan digunakan untuk menggambar elemen visual, dan posisi awal serta tujuan (goal) di dalam grid. Grid juga diinisialisasi dengan sebuah matriks 5x5, di mana setiap nilai merepresentasikan elemen yang berbeda, seperti dinding (nilai 1), special reward (nilai 2), dan goal (nilai 3).

```python
# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('MDP AI Grid')
font = pygame.font.SysFont(None, 24)

# Transition Probability: 80% correct move, 20% random move
MOVE_PROB = 0.8

def draw_grid():
    """Menggambar grid berdasarkan nilai pada matriks `grid`."""
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            color = WHITE
            if grid[i][j] == 1:
                color = GRAY  # Dinding
            elif grid[i][j] == 2:
                color = GREEN  # Reward
            elif grid[i][j] == 3:
                color = BLUE  # Goal
            pygame.draw.rect(screen, color, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))
            pygame.draw.rect(screen, BLACK, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE), 1)

def draw_player():
    """Menggambar pemain sebagai lingkaran merah di posisi pemain saat ini."""
    pygame.draw.circle(screen, RED, (player[1] * CELL_SIZE + CELL_SIZE // 2, player[0] * CELL_SIZE + CELL_SIZE // 2), CELL_SIZE // 2)
```

Penjelasan :

Pada kode ini, Pygame diinisialisasi, dan ukuran layar ditetapkan sesuai ukuran grid. Judul jendela permainan juga diberikan, yaitu "MDP AI Grid". MOVE_PROB merupakan probabilitas transisi yang menentukan apakah pemain akan bergerak sesuai rencana (80% probabilitas) atau secara acak (20%). Fungsi draw_grid() digunakan untuk menggambar grid sesuai nilai dalam matriks grid, dengan dinding, reward, dan goal memiliki warna yang berbeda. Fungsi draw_player() menggambar pemain dalam bentuk lingkaran merah di posisi yang sesuai.

```python
def display_details():
    """Menampilkan detail posisi pemain dan reward dari setiap aksi yang tersedia."""
    details = f"Current Position: ({player[0]}, {player[1]})"
    actions = ['up', 'down', 'left', 'right']
    for action in actions:
        next_state = get_next_state(player, action)
        if is_valid_move(next_state):
            reward = get_reward(next_state)
            details += f"\nMove {action}: Next State ({next_state[0]}, {next_state[1]}), Reward: {reward}"
        else:
            details += f"\nMove {action}: Invalid Move"
    detail_surf = font.render(details, True, BLACK)
    screen.blit(detail_surf, (10, HEIGHT + 10))

def get_next_state(state, action):
    """Mengembalikan posisi berikutnya berdasarkan aksi yang dilakukan."""
    x, y = state
    if action == 'up':
        x -= 1
    elif action == 'down':
        x += 1
    elif action == 'left':
        y -= 1
    elif action == 'right':
        y += 1
    return [x, y]
```

Penjelasan :

Kode ini mengandung fungsi untuk menampilkan detail posisi pemain serta reward dari setiap gerakan yang mungkin. display_details() menampilkan posisi

pemain saat ini, lalu mengevaluasi setiap kemungkinan aksi (atas, bawah, kiri, kanan) dan menentukan apakah gerakan tersebut valid serta reward yang diperoleh. Fungsi get_next_state() digunakan untuk menghitung posisi berikutnya berdasarkan aksi yang dilakukan.

```python
def is_valid_move(state):
    """Memeriksa apakah gerakan valid (tidak mengenai dinding atau keluar grid)."""
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and grid[x][y] != 1

def get_reward(state):
    """Menghitung reward berdasarkan posisi saat ini."""
    x, y = state
    if not is_valid_move(state):
        return -float('inf')  # Invalid move
    elif grid[x][y] == 3:
        return 10  # Goal tercapai
    elif grid[x][y] == 2:
        return 5  # Special reward cell
    else:
        return -manhattan_distance(state, GOAL)  # Penalti jarak ke goal

def manhattan_distance(state1, state2):
    """Menghitung jarak Manhattan antara dua posisi di grid."""
    return abs(state1[0] - state2[0]) + abs(state1[1] - state2[1])
```

Penjelasan :

Kode ini mendefinisikan tiga fungsi utama. is_valid_move() mengecek apakah gerakan ke sebuah posisi valid atau tidak, dengan memastikan posisi tersebut tidak menabrak dinding atau keluar dari grid. get_reward() menghitung reward berdasarkan posisi pemain saat ini; jika pemain mencapai goal, reward-nya 10, jika mengenai cell spesial reward adalah 5, dan jika tidak, penalti dihitung berdasarkan jarak Manhattan ke goal. manhattan_distance() adalah fungsi untuk menghitung jarak Manhattan antara dua posisi pada grid.

```python
def stochastic_transition(player, action):
    """Melakukan transisi stochastik dengan probabilitas transisi."""
    if random.random() < MOVE_PROB:  # 80% chance the intended action happens
        return get_next_state(player, action), action
    else:  # 20% chance to take a random action
        random_action = random.choice(['up', 'down', 'left', 'right'])
        return get_next_state(player, random_action), random_action

def ai_move(player):
    """Fungsi AI yang menggerakkan pemain berdasarkan jarak terdekat ke goal."""
    actions = ['up', 'down', 'left', 'right']
    best_move = None
    best_distance = float('inf')
    chosen_action = None

    for action in actions:
        next_pos, action_taken = stochastic_transition(player, action)  # Menggunakan transisi stochastik
        if is_valid_move(next_pos):
            distance = manhattan_distance(next_pos, GOAL)
            if distance < best_distance:
                best_distance = distance
                best_move = next_pos
                chosen_action = action_taken

    if best_move:
        print(f"Player moves {chosen_action} to position {best_move}")  # Output pergerakan
    return best_move if best_move else player
```

Penjelasan :

Dalam kode ini, fungsi stochastic_transition() menentukan pergerakan pemain dengan probabilitas transisi stokastik, di mana 80% kemungkinan akan bergerak sesuai rencana dan 20% kemungkinan akan bergerak secara acak. Fungsi ai_move() menggerakkan pemain secara otomatis berdasarkan pendekatan AI yang menghitung jarak terdekat ke tujuan (goal) menggunakan jarak Manhattan. Pemain akan bergerak ke posisi dengan jarak minimum ke tujuan, jika gerakan tersebut valid.

```
# Main game loop
running = True
clock = pygame.time.Clock()
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # AI automatically moves the player using MDP
    next_pos = ai_move(player)
    if next_pos:
        player = next_pos

    # Check for goal reached
    if grid[player[0]][player[1]] == 3:
        print('Goal reached!')
        running = False

    screen.fill(WHITE)
    draw_grid()
    draw_player()
    display_details()
    pygame.display.flip()

    clock.tick(1)  # AI moves once per second

pygame.quit()
sys.exit()

```

Penjelasan :

Pada kode ini, terdapat loop utama permainan (main game loop) yang terus berjalan hingga pemain menekan tombol keluar atau mencapai tujuan (goal). Di dalam loop, AI secara otomatis menggerakkan pemain menggunakan fungsi ai_move(). Setelah setiap gerakan, layar di-update untuk menggambar ulang grid dan pemain, serta menampilkan detail posisi. Jika pemain mencapai goal, game berakhir.

- **Penjelasan Bagian Markov Decision Process (MDP)**

1. States (Keadaan/Posisi)

```
29    # Player starting position
30    player = list(START)
```

Penjelasan :

State dalam MDP mewakili posisi pemain di dalam grid. Pada kode ini, state didefinisikan sebagai koordinat posisi pemain di dalam grid, yang direpresentasikan oleh variabel player. Setiap state merepresentasikan posisi spesifik di grid yang saat ini ditempati pemain.

2. Actions (Aksi/Pergerakan)

```
actions = ['up', 'down', 'left', 'right']
```

Penjelasan :

Aksi dalam MDP adalah tindakan yang bisa diambil dari satu state ke state lainnya. Dalam kode ini, aksi berupa pergerakan pemain ke salah satu dari empat arah: up (atas), down (bawah), left (kiri), atau right (kanan). Fungsi get_next_state() menentukan perubahan posisi pemain berdasarkan aksi yang diambil.

3. Transition Model (Model Transisi)

```
38    # Transition Probability: 80% correct move, 20% random move
39    MOVE_PROB = 0.8
```

Penjelasan :

Model transisi dalam MDP menentukan bagaimana pemain berpindah dari satu state ke state lainnya, berdasarkan aksi yang dilakukan. Pada kode ini, transisi bersifat stokastik (mengandung ketidakpastian), di mana terdapat probabilitas sebesar 80% (nilai MOVE_PROB) untuk bergerak sesuai aksi yang direncanakan, dan 20% untuk bergerak secara acak. Hal ini diimplementasikan dalam fungsi stochastic_transition(), yang menentukan apakah pemain bergerak sesuai aksi atau mengambil aksi acak.

4. Rewards (Imbalan/Reward)

```
91  def get_reward(state):
92      """Menghitung reward berdasarkan posisi saat ini."""
93      x, y = state
94      if not is_valid_move(state):
95          return -float('inf')  # Invalid move
96      elif grid[x][y] == 3:
97          return 10  # Goal tercapai
98      elif grid[x][y] == 2:
99          return 5  # Special reward cell
100     else:
101         return -manhattan_distance(state, GOAL)  # Penalti jarak ke goal
```

Penjelasan :

Reward dalam MDP adalah imbalan atau penalti yang diterima pemain setelah mencapai state tertentu sebagai hasil dari aksi yang diambil. Dalam kode ini, reward dihitung melalui fungsi get_reward(). Jika pemain mencapai goal (dengan nilai grid 3), reward positif sebesar +10 diberikan. Jika pemain mencapai cell dengan special reward (nilai grid 2), reward +5 diberikan. Selain itu, jika pemain bergerak ke state lain yang valid, penalti diberikan berdasarkan jarak Manhattan ke tujuan. Penalti ini membantu pemain untuk secara alami berusaha mendekati tujuan (goal).

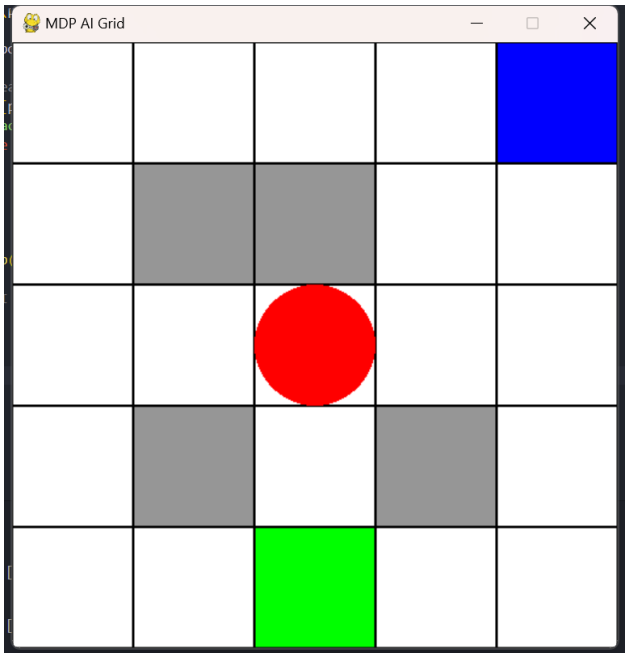
5. Policy (Kebijakan/Strategi)

```python
def ai_move(player):
    """Fungsi AI yang menggerakkan pemain berdasarkan jarak terdekat ke goal."""
    actions = ['up', 'down', 'left', 'right']
    best_move = None
    best_distance = float('inf')
    chosen_action = None

    for action in actions:
        next_pos, action_taken = stochastic_transition(player, action)  # Menggunakan transisi stochastik
        if is_valid_move(next_pos):
            distance = manhattan_distance(next_pos, GOAL)
            if distance < best_distance:
                best_distance = distance
                best_move = next_pos
                chosen_action = action_taken

    if best_move:
        print(f"Player moves {chosen_action} to position {best_move}")  # Output pergerakan
    return best_move if best_move else player
```

Penjelasan :

Kebijakan dalam MDP adalah strategi atau aturan yang menentukan aksi apa yang akan diambil dari setiap state. Dalam kode ini, fungsi ai_move() digunakan sebagai kebijakan sederhana yang memandu pemain untuk bergerak ke arah yang meminimalkan jarak ke tujuan (goal). AI menghitung jarak Manhattan dari setiap state yang valid setelah aksi dilakukan, dan memilih aksi yang mendekatkan pemain ke goal. Meskipun strategi ini berbasis greedy (rakus), di mana AI selalu memilih aksi yang memperpendek jarak, hal ini tetap relevan dalam konteks MDP karena AI mempertimbangkan reward yang mungkin diterima dari setiap aksi.

Hasil Output




```
PS D:\Tugas Kuliah\SEMESTER 7\Rein\UTS RL> python markov_decision_process.py
pygame 2.6.1 (SDL 2.28.4, Python 3.10.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
Player moves up to position [3, 2]
Player moves up to position [2, 2]
Player moves right to position [2, 3]
Player moves up to position [1, 3]
Player moves up to position [0, 3]
Player moves right to position [0, 4]
Goal reached!
```

Penjelasan :

Pada hasil output yang ditampilkan, kita dapat melihat dua bagian utama, yaitu jendela game dan terminal.

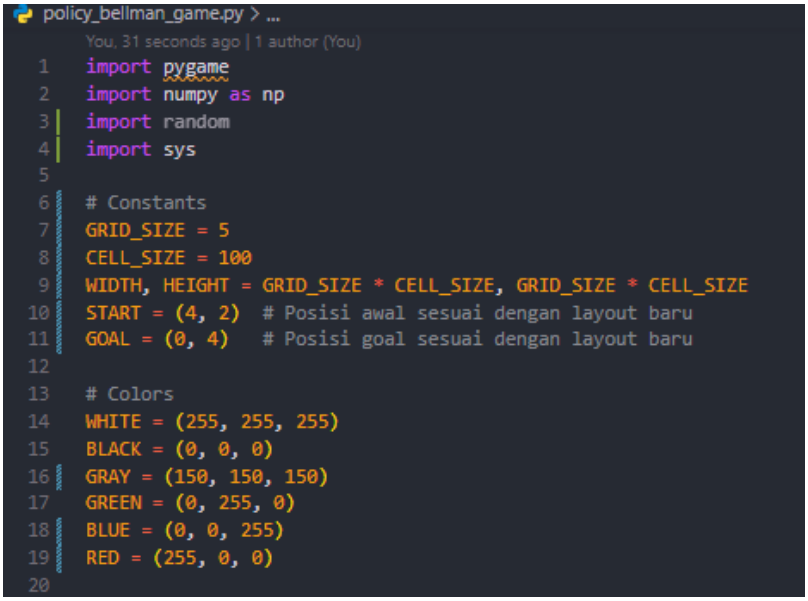
Pada jendela game (game window), terdapat sebuah grid berukuran 5x5 yang berfungsi sebagai area permainan. Tiap cell pada grid ini memiliki warna yang berbeda, yang menunjukkan fungsinya masing-masing. Warna putih menunjukkan cell kosong yang bisa dilewati oleh pemain, warna abu-abu adalah dinding atau penghalang yang tidak bisa dilewati, warna hijau adalah cell special reward yang memberikan poin ekstra, dan warna biru adalah goal (tujuan akhir) yang harus dicapai oleh pemain. Posisi awal pemain ditandai dengan lingkaran merah, dan pemain secara otomatis bergerak di grid mengikuti aksi yang dipilih oleh AI.

Pada terminal, tercetak log pergerakan pemain yang dikendalikan oleh AI. Misalnya, aksi seperti "Player moves up to position [3, 2]" menunjukkan bahwa pemain bergerak ke cell di koordinat (3, 2). AI menggunakan pendekatan Markov Decision Process (MDP) untuk memilih aksi yang bertujuan meminimalkan jarak pemain ke tujuan (goal) berdasarkan jarak Manhattan. Pada setiap langkah, AI memilih aksi terbaik hingga akhirnya pemain mencapai goal di koordinat (0, 4). Setelah mencapai goal, terminal menampilkan pesan "Goal reached!", yang menandakan bahwa permainan telah selesai dan pemain berhasil mencapai tujuan.

Policy Bellman Equation

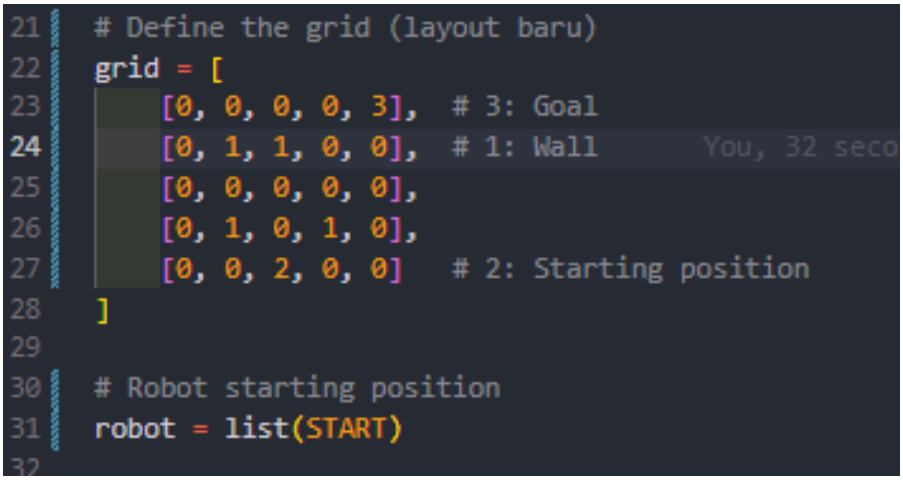
Materi : https://github.com/brianspiering/rl-course/blob/main/02_policy__value_iteration/2_policy__bellman_equation.ipynb

- **policy_bellman_equation.py**



Penjelasan :

Pada bagian ini, kita mengimpor library yang dibutuhkan (pygame, numpy, random, dan sys). Konstanta juga didefinisikan, termasuk ukuran grid, ukuran sel, serta warna yang digunakan untuk menggambarkan elemen-elemen di dalam grid (putih untuk ruang bebas, abu-abu untuk dinding, hijau untuk posisi awal, biru untuk tujuan, dan merah untuk robot).



Penjelasan :

Kode ini berfungsi untuk mendefinisikan layout dari labirin menggunakan matriks 2D yang merepresentasikan posisi kosong (0), dinding (1), posisi awal (2), dan tujuan (3). Posisi awal robot juga ditentukan berdasarkan variabel START.

```
33  # Define parameters for Policy Iteration
34  actions = ['up', 'down', 'left', 'right']
35  action_map = {0: 'up', 1: 'down', 2: 'left', 3: 'right'}
36  gamma = 0.9  # Discount factor
37  theta = 0.0001  # Threshold for convergence
38  policy = np.random.choice([0, 1, 2, 3], size=(GRID_SIZE, GRID_SIZE))  # Random initial policy
39  V = np.zeros((GRID_SIZE, GRID_SIZE))  # Initialize value table
```

Penjelasan :

Kode ini berfungsi untuk kita menginisialisasi parameter yang diperlukan untuk menjalankan algoritma Policy Iteration. Pertama, didefinisikan empat aksi yang dapat dilakukan oleh robot: up (ke atas), down (ke bawah), left (ke kiri), dan right (ke kanan). Kemudian, action_map digunakan untuk memetakan indeks aksi ke nama aksi yang lebih mudah dipahami. Parameter gamma digunakan sebagai discount factor yang menentukan seberapa besar robot menghargai reward di masa depan, sedangkan theta digunakan sebagai ambang batas untuk menentukan kapan proses evaluasi kebijakan mencapai konvergensi. Selanjutnya, kebijakan awal (policy) diinisialisasi secara acak dengan memilih di antara aksi-aksi tersebut untuk setiap sel dalam grid. Tabel nilai (V) juga diinisialisasi dengan nol untuk menyimpan nilai dari setiap keadaan (state) dalam grid. Nilai-nilai ini akan diperbarui saat kebijakan dioptimalkan.

```
41  # Pygame setup
42  pygame.init()
43  screen = pygame.display.set_mode((WIDTH, HEIGHT))
44  pygame.display.set_caption("Robot Grindworld (Policy Iteration)")
45  font = pygame.font.SysFont(None, 24)
```

Penjelasan :

Kode ini berfungsi untuk mengatur tampilan menggunakan pygame, termasuk inisialisasi layar dengan dimensi sesuai ukuran grid. Juga menetapkan judul jendela untuk tampilan game.

```python
def draw_grid():
    """Draw the grid grid."""
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            color = WHITE
            if grid[i][j] == 1:
                color = GRAY  # Wall
            elif grid[i][j] == 3:
                color = BLUE  # Goal
            elif grid[i][j] == 2:
                color = GREEN  # Start
            pygame.draw.rect(screen, color, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))
            pygame.draw.rect(screen, BLACK, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE), 1)

def draw_robot():
    """Draw the robot as a red circle."""
    pygame.draw.circle(screen, RED, (robot[1] * CELL_SIZE + CELL_SIZE // 2, robot[0] * CELL_SIZE + CELL_SIZE // 2), CELL_SIZE // 3)
```

Penjelasan :

Kode diatas berfungsi untuk kita mendefinisikan dua fungsi untuk menggambar elemen-elemen permainan di layar. draw_maze() digunakan untuk menggambar labirin sesuai dengan layout matriks maze. Setiap sel dalam grid diberi warna berdasarkan jenisnya: dinding diberi warna abu-abu, tujuan diberi warna biru, dan posisi awal robot diberi warna hijau. Setiap sel kemudian digambar sebagai persegi dengan ukuran yang telah ditentukan oleh CELL_SIZE. Selain itu, draw_robot() menggambar robot di posisinya saat ini sebagai lingkaran merah. Lingkaran ini ditempatkan di tengah-tengah sel tempat robot berada untuk menunjukkan posisi robot dalam labirin secara jelas pada layar permainan.

```python
def get_next_state(state, action):
    """Returns the next state based on the current state and action."""
    x, y = state
    if action == 'up' and x > 0:
        x -= 1
    elif action == 'down' and x < GRID_SIZE - 1:
        x += 1
    elif action == 'left' and y > 0:
        y -= 1
    elif action == 'right' and y < GRID_SIZE - 1:
        y += 1      You, 5 minutes ago • Uncommitted changes
    return [x, y]

def is_valid_move(state):
    """Check if a move is valid (not hitting a wall or out of bounds)."""
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and grid[x][y] != 1

def get_reward(state):
    """Returns the reward for reaching a given state."""
    x, y = state
    if grid[x][y] == 3:  # Reaching the goal
        return 10
    else:
        return -1  # Default step cost
```

Penjelasan :

Kode diatas berisi tiga fungsi yang mengatur logika pergerakan robot dalam labirin dan menghitung reward. get_next_state() menghitung posisi berikutnya berdasarkan posisi saat ini (state) dan aksi yang diambil (action). Fungsi ini

memastikan robot bergerak dalam batas grid dan hanya bergerak sesuai arah yang ditentukan jika memungkinkan. is_valid_move() memverifikasi apakah gerakan yang diusulkan valid, artinya tidak keluar dari batas grid atau menabrak dinding. get_reward() memberikan reward untuk setiap posisi yang dicapai oleh robot. Reward tertinggi diberikan saat robot mencapai tujuan (nilai 3), sementara untuk pergerakan biasa diberikan penalti -1, yang mengurangi nilai untuk setiap langkah yang diambil.

```python
def policy_evaluation():
    """Policy evaluation step: updates the value function based on the current policy."""
    global V
    while True:
        delta = 0
        for x in range(GRID_SIZE):
            for y in range(GRID_SIZE):
                if grid[x][y] == 1:  # Skip walls
                    continue
                v = V[x, y]
                action = policy[x, y]
                next_state = get_next_state([x, y], action_map[action])
                if is_valid_move(next_state):
                    reward = get_reward(next_state)
                    V[x, y] = reward + gamma * V[next_state[0], next_state[1]]
                delta = max(delta, abs(v - V[x, y]))
        if delta < theta:
            break

def policy_improvement():
    """Policy improvement step: updates the policy to be greedy with respect to the value function."""
    policy_stable = True
    for x in range(GRID_SIZE):
        for y in range(GRID_SIZE):
            if grid[x][y] == 1:  # Skip walls
                continue
            old_action = policy[x, y]
            best_action = old_action
            best_value = -float('inf')
            for action_index, action in enumerate(actions):
                next_state = get_next_state([x, y], action)
                if is_valid_move(next_state):
                    reward = get_reward(next_state)
                    value = reward + gamma * V[next_state[0], next_state[1]]
                    if value > best_value:
                        best_value = value
                        best_action = action_index
            policy[x, y] = best_action
            if old_action != best_action:
                policy_stable = False
    return policy_stable
```

Penjelasan :

Dalam kode ini, policy_evaluation() memperbarui tabel nilai V berdasarkan kebijakan saat ini dengan menggunakan Persamaan Bellman. Evaluasi kebijakan bertujuan untuk memperbarui nilai-nilai keadaan (states) hingga perbedaan nilai baru dan lama kurang dari ambang theta, yang menunjukkan konvergensi. policy_improvement() kemudian memperbarui kebijakan dengan memilih aksi yang memberikan nilai tertinggi berdasarkan nilai-nilai yang telah diperbarui. Aksi terbaik ditentukan untuk setiap sel sehingga robot dapat bergerak menuju tujuan secara lebih efisien. Proses ini dilakukan secara berulang sampai kebijakan tidak lagi berubah, yang berarti kebijakan tersebut telah mencapai stabilitas.

```python
def policy_iteration():
    """Performs Policy Iteration: alternating between Policy Evaluation and Policy Improvement."""
    while True:
        policy_evaluation()
        if policy_improvement():
            break

def print_grid():
    """Print the current grid state to the terminal."""
    print("\nCurrent grid State:")
    for i in range(GRID_SIZE):
        row = ""
        for j in range(GRID_SIZE):
            if [i, j] == robot:
                row += " R "  # Robot's position
            elif grid[i][j] == 1:
                row += " # "  # Wall
            elif grid[i][j] == 3:
                row += " G "  # Goal
            else:
                row += " . "  # Free space
        print(row)

# Run Policy Iteration once to get an optimal policy
policy_iteration()
```

Penjelasan :

Di cell ini, policy_iteration() menggabungkan kedua fungsi sebelumnya, yaitu policy_evaluation() dan policy_improvement(), untuk menemukan kebijakan optimal bagi robot. Fungsi ini menjalankan evaluasi dan perbaikan kebijakan secara berulang sampai kebijakan mencapai stabilitas. Fungsi print_maze() digunakan untuk mencetak kondisi labirin saat ini ke terminal, menunjukkan posisi robot (R), dinding (#), tujuan (G), dan ruang bebas (.). Ini membantu dalam memantau perkembangan robot saat mencoba mencapai tujuan berdasarkan kebijakan optimal yang telah ditemukan.

```python
# Main game loop
running = True
clock = pygame.time.Clock()

while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Move the robot according to the optimal policy
    best_action = action_map[policy[robot[0], robot[1]]]
    next_pos = get_next_state(robot, best_action)

    if is_valid_move(next_pos):
        robot = next_pos  # Move the robot

    # Print the current grid state to the terminal
    print_grid()

    # Check if the robot reaches the goal
    if grid[robot[0]][robot[1]] == 3:
        print("Goal reached!")
        running = False

    # Draw the game state
    screen.fill(WHITE)
    draw_grid()
    draw_robot()
    pygame.display.flip()

    clock.tick(2)  # Move the robot every 2 seconds

pygame.quit()
sys.exit()
```

Penjelasan :

Kode ini mengandung loop utama permainan yang menggunakan pygame untuk menampilkan visualisasi dan mengelola input. Loop ini akan berjalan selama

variabel running bernilai True. Di setiap iterasi, event pygame diperiksa untuk mendeteksi jika pengguna ingin menutup jendela permainan. Robot bergerak sesuai dengan kebijakan optimal yang ditemukan oleh policy_iteration(), dengan mengambil tindakan terbaik berdasarkan posisi saat ini. Pergerakan robot divalidasi menggunakan is_valid_move() untuk memastikan robot tidak menabrak dinding atau keluar dari grid. Setiap langkah robot dicetak ke terminal menggunakan print_maze() untuk menunjukkan posisi dan kondisi labirin saat ini. Jika robot mencapai tujuan (posisi dengan nilai 3), pesan "Goal reached!" dicetak dan loop berhenti. Fungsi draw_maze() dan draw_robot() digunakan untuk memperbarui tampilan visual di layar, memperlihatkan posisi terbaru robot dan kondisi labirin. Kecepatan pembaruan tampilan diatur menggunakan clock.tick(2) sehingga robot bergerak setiap 2 detik.

- **Penjelasan Bagian Policy Bellman Equation**

```
def policy_evaluation():
    """Policy evaluation step: updates the value function based on the current policy."""
    global V
    while True:
        delta = 0
        for x in range(GRID_SIZE):
            for y in range(GRID_SIZE):
                if grid[x][y] == 1:  # Skip walls
                    continue
                v = V[x, y]
                action = policy[x, y]
                next_state = get_next_state([x, y], action_map[action])
                if is_valid_move(next_state):
                    reward = get_reward(next_state)
                    V[x, y] = reward + gamma * V[next_state[0], next_state[1]]
                delta = max(delta, abs(v - V[x, y]))
        if delta < theta:
            break
```

Penjelasan :

Pada algoritma Policy Iteration, bagian Bellman Equation muncul dalam proses policy evaluation, di mana nilai dari setiap keadaan diperbarui berdasarkan kebijakan yang sedang berjalan. Dalam kode di atas, ini diimplementasikan pada bagian evaluasi kebijakan (policy evaluation) yang menggunakan persamaan Bellman untuk memperbarui nilai-nilai setiap keadaan (state). Persamaan Bellman ini menentukan nilai dari suatu keadaan sebagai kombinasi dari reward yang diterima dan nilai dari keadaan berikutnya, yang didiskonto menggunakan faktor γ (discount factor).

Dalam kode policy_evaluation(), kita menggunakan Persamaan Bellman sebagai berikut:

$$V(s) = R(s') + \gamma \cdot V(s')$$

Dimana :

V(s) : Nilai dari keadaan saat ini.

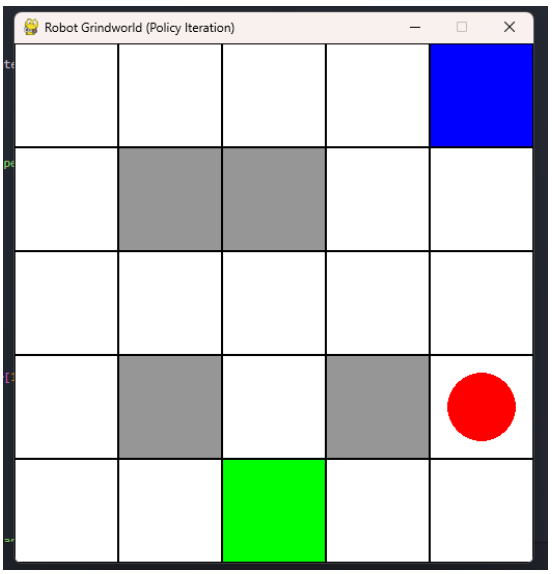
R(s′) : Reward yang didapat setelah mengambil tindakan dan mencapai keadaan berikutnya s′s's′.

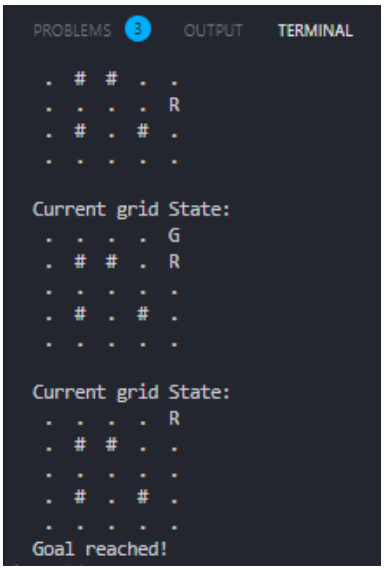
γ : Faktor diskonto yang memengaruhi seberapa besar nilai masa depan dihargai.

s′ : Keadaan berikutnya yang dicapai berdasarkan kebijakan saat ini.

Persamaan Bellman diterapkan ketika memperbarui nilai dari setiap keadaan berdasarkan reward yang diterima dan nilai diskonto dari keadaan berikutnya. V[x, y] diperbarui dengan formula: reward + gamma * V[next_state[0], next_state[1]], di mana reward adalah nilai imbalan untuk mencapai keadaan berikutnya, dan V[next_state[0], next_state[1]] adalah nilai dari keadaan berikutnya tersebut. gamma adalah faktor diskonto yang mengontrol seberapa jauh nilai keadaan masa depan memengaruhi nilai keadaan saat ini. Proses ini terus berjalan sampai perubahan dalam nilai (delta) lebih kecil dari nilai ambang batas theta, yang menunjukkan bahwa fungsi nilai telah konvergen.

- **Hasil Output**





Penjelasan :

Hasil output yang ditunjukkan terdiri dari dua bagian utama: visualisasi grafis dari permainan "Robot Grid World" menggunakan pygame dan representasi tekstual dari posisi robot dalam terminal.

Pada bagian visualisasi grafis, terlihat tampilan grid labirin berukuran 5x5, di mana setiap sel digambarkan dengan warna yang berbeda untuk menunjukkan elemen-elemen di dalamnya. Sel-sel berwarna putih menunjukkan ruang bebas di mana robot dapat bergerak, sedangkan sel berwarna abu-abu menunjukkan dinding yang menghalangi pergerakan robot. Posisi awal robot ditandai dengan lingkaran merah, dan posisi tujuan ditandai dengan warna biru di bagian kanan atas. Sel berwarna hijau menandakan posisi awal robot sebelum pergerakan dimulai. Lingkaran merah menggambarkan robot yang bergerak di sekitar grid mengikuti kebijakan optimal yang diperoleh dari algoritma Policy Iteration.

Pada bagian terminal, kita melihat representasi grid dalam bentuk teks yang menampilkan kondisi grid pada setiap langkah pergerakan robot. Setiap baris "Current grid State:" menunjukkan posisi robot (R), dinding (#), tujuan (G), dan ruang kosong (.). Pada awalnya, robot berada di dekat posisi awal, dan dengan setiap langkah, posisi R bergerak menuju G. Terminal juga menunjukkan pergerakan robot di setiap langkah dengan jelas. Setelah beberapa langkah, robot mencapai sel tujuan yang ditandai dengan G, dan terminal mencetak pesan "Goal reached!", menandakan bahwa simulasi telah berhasil menyelesaikan tugasnya. Robot telah berhasil bergerak dari posisi awal ke posisi tujuan dengan mengikuti kebijakan optimal yang dipelajari melalui iterasi kebijakan.

Q Function Value Iteration

Materi : https://github.com/brianspiering/rl-course/blob/main/02_policy__value_iteration/3_q_function__value_iteration.ipynb

- **q_function_value_iteration.py**

```
import pygame
import sys
import random
import numpy as np

# Constants
GRID_SIZE = 5
CELL_SIZE = 100
WIDTH, HEIGHT = GRID_SIZE * CELL_SIZE, GRID_SIZE * CELL_SIZE
START = (4, 2)
GOAL = (0, 4)

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (150, 150, 150)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
RED = (255, 0, 0)
```

Penjelasan :

Pada bagian ini, kita mengimpor pustaka pygame, sys, random, dan numpy serta mendefinisikan berbagai konstanta, seperti ukuran grid, ukuran setiap sel, posisi awal dan tujuan pemain, serta warna-warna yang akan digunakan untuk menggambar objek dalam permainan.

```
# Grid definition
grid = [
    [0, 0, 0, 0, 3],  # 0: Empty space, 1: Wall, 2: Reward, 3: Goal
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 2, 0, 0]
]

# Player starting position
player = list(START)
```

Penjelasan :

Kode ini menampilkan Grid permainan yang didefinisikan sebagai array 2D yang berisi nilai-nilai yang merepresentasikan ruang kosong (0), dinding (1),

reward (2), dan goal (3). Pemain dimulai pada posisi yang didefinisikan oleh variabel START.

```
# Initialize Q-table and Value Table
Q = np.zeros((GRID_SIZE, GRID_SIZE, 4))  # Q-Table: 4 actions (up, down, left, right)
V = np.zeros((GRID_SIZE, GRID_SIZE))  # Value Table untuk setiap state
gamma = 0.9  # Discount factor
theta = 0.0001  # Threshold untuk konvergensi Value Iteration
alpha = 0.1  # Learning rate untuk Q-Learning
epsilon = 0.2  # Exploration rate untuk Q-Learning
```

Penjelasan :

Pada tampilan kode ini, Q-table digunakan untuk menyimpan nilai dari setiap aksi pada setiap state, dan Value Table (V) menyimpan nilai dari setiap state. Parameter seperti discount factor (gamma), threshold konvergensi (theta), learning rate (alpha), dan exploration rate (epsilon) juga diinisialisasi di sini.

```
# Actions and directions
actions = ['up', 'down', 'left', 'right']
action_map = {0: 'up', 1: 'down', 2: 'left', 3: 'right'}

# Initialize Pygame
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Q-Learning + Value Iteration Grid')
font = pygame.font.SysFont(None, 24)

def draw_grid():
    """Menggambar grid berdasarkan nilai pada matriks `grid`."""
    for i in range(GRID_SIZE):
        for j in range(GRID_SIZE):
            color = WHITE
            if grid[i][j] == 1:
                color = GRAY  # Dinding
            elif grid[i][j] == 2:
                color = GREEN  # Reward
            elif grid[i][j] == 3:
                color = BLUE  # Goal
            pygame.draw.rect(screen, color, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE))
            pygame.draw.rect(screen, BLACK, (j * CELL_SIZE, i * CELL_SIZE, CELL_SIZE, CELL_SIZE), 1)

def draw_player():
    """Menggambar pemain sebagai lingkaran merah di posisi pemain saat ini."""
    pygame.draw.circle(screen, RED, (player[1] * CELL_SIZE + CELL_SIZE // 2, player[0] * CELL_SIZE + CELL_SIZE // 2), CELL_SIZE // 2)
```

Penjelasan :

Kode ini menampilkan menginisialisasi tampilan permainan menggunakan Pygame dan mengatur ukuran layar berdasarkan dimensi grid yang didefinisikan sebelumnya. Dua fungsi draw_grid dan draw_player digunakan untuk menggambar grid permainan dan pemain di layar. draw_grid mengecek nilai dari setiap elemen grid untuk menentukan warna yang sesuai untuk sel tersebut (dinding, reward, atau goal), sedangkan draw_player menggambar pemain sebagai lingkaran merah di posisi saat ini.

```python
def get_next_state(state, action):
    """Mengembalikan posisi berikutnya berdasarkan aksi yang dilakukan."""
    x, y = state
    if action == 'up':
        x -= 1
    elif action == 'down':
        x += 1
    elif action == 'left':
        y -= 1
    elif action == 'right':
        y += 1
    return [x, y]

def is_valid_move(state):
    """Memeriksa apakah gerakan valid (tidak mengenai dinding atau keluar grid)."""
    x, y = state
    return 0 <= x < GRID_SIZE and 0 <= y < GRID_SIZE and grid[x][y] != 1

def get_reward(state):
    """Menghitung reward berdasarkan posisi saat ini."""
    x, y = state
    if not is_valid_move(state):
        return -float('inf')  # Invalid move
    elif grid[x][y] == 3:
        return 10  # Goal tercapai
    elif grid[x][y] == 2:
        return 5  # Special reward cell
    else:
        return -manhattan_distance(state, GOAL)  # Penalti jarak ke goal

def manhattan_distance(state1, state2):
    """Menghitung jarak Manhattan antara dua posisi di grid."""
    return abs(state1[0] - state2[0]) + abs(state1[1] - state2[1])
```

Penjelasan :

Pada kode ini, terdapat beberapa fungsi penting untuk mengelola gerakan dan reward didefinisikan. get_next_state mengembalikan posisi berikutnya berdasarkan aksi yang diambil. is_valid_move memeriksa apakah gerakan tersebut sah (tidak keluar dari grid atau menabrak dinding). get_reward mengembalikan reward berdasarkan state yang dicapai, di mana reward maksimum diberikan ketika pemain mencapai goal. Jarak Manhattan digunakan untuk menghitung penalti jika pemain tidak berada di posisi goal.

```python
def value_iteration():
    """Algoritma value iteration untuk memperbarui nilai dari setiap state."""
    global V
    delta = float('inf')  # Perubahan maksimum antar iterasi

    while delta > theta:  # Lakukan sampai perubahan nilai menjadi sangat kecil (konvergen)
        delta = 0
        for x in range(GRID_SIZE):
            for y in range(GRID_SIZE):
                if grid[x][y] == 1:  # Jika ini dinding, lewati
                    continue
                v = V[x, y]  # Simpan nilai state saat ini

                # Evaluasi untuk setiap aksi, hitung nilai maksimal
                max_value = -float('inf')
                for action in actions:
                    next_state = get_next_state([x, y], action)
                    if is_valid_move(next_state):
                        reward = get_reward(next_state)
                        value = reward + gamma * V[next_state[0], next_state[1]]
                        if value > max_value:
                            max_value = value

                # Update nilai state dengan nilai terbaik dari semua aksi
                V[x, y] = max_value
                delta = max(delta, abs(v - V[x, y]))  # Update delta untuk mengecek konvergensi
```

Penjelasan :

Pada bagian ini, fungsi value_iteration() digunakan untuk memperbarui tabel nilai (V) menggunakan algoritma Value Iteration. Ini adalah salah satu algoritma untuk menemukan kebijakan optimal dalam grid world. Setiap state akan dievaluasi berdasarkan reward yang didapatkan dari perpindahan ke state lain. Iterasi akan berhenti saat perubahan nilai (delta) menjadi sangat kecil, yang berarti tabel nilai telah mencapai konvergensi. Proses ini melibatkan mengevaluasi semua aksi yang mungkin diambil di setiap state, menghitung nilai maksimum dari aksi-aksi tersebut, lalu memperbarui nilai state berdasarkan nilai terbaik.

```python
def initialize_q_from_value():
    """Inisialisasi Q-table menggunakan value dari Value Iteration."""
    global Q
    for x in range(GRID_SIZE):
        for y in range(GRID_SIZE):
            if grid[x][y] != 1:  # Bukan dinding
                for action_index, action in enumerate(actions):
                    next_state = get_next_state([x, y], action)
                    if is_valid_move(next_state):
                        reward = get_reward(next_state)
                        Q[x, y, action_index] = reward + gamma * V[next_state[0], next_state[1]]
```

Penjelasan :

Kode ini berisi fungsi initialize_q_from_value(), yang menginisialisasi tabel Q (Q-table) berdasarkan tabel nilai (V) yang dihitung dari value iteration sebelumnya. Fungsi ini mengisi tabel Q dengan nilai untuk setiap aksi yang mungkin dari setiap state, di mana nilai tersebut dihitung berdasarkan reward dan diskon dari reward di state berikutnya.

```
def choose_action(state, epsilon):
    """Memilih action menggunakan epsilon-greedy strategy."""
    if random.uniform(0, 1) < epsilon:
        return random.choice([0, 1, 2, 3])  # Explore
    else:
        return np.argmax(Q[state[0], state[1]])  # Exploit
```

Penjelasan :

Kode ini mengimplementasikan strategi epsilon-greedy yang digunakan untuk memilih aksi bagi pemain berdasarkan state saat ini. Jika angka acak yang dihasilkan lebih kecil dari nilai epsilon, maka pemain akan mengeksplorasi (memilih aksi secara acak). Jika tidak, pemain akan mengeksploitasi nilai Q yang telah dihitung sebelumnya dengan memilih aksi yang memiliki nilai Q tertinggi. Strategi ini memberikan keseimbangan antara eksplorasi (mencoba aksi baru) dan eksploitasi (menggunakan pengetahuan yang sudah ada untuk memaksimalkan reward).

```
def update_q_table(state, action, reward, next_state):
    """Update Q-table menggunakan aturan Q-Learning."""
    next_max = np.max(Q[next_state[0], next_state[1]])  # Cari nilai maksimum di state berikutnya
    Q[state[0], state[1], action] = Q[state[0], state[1], action] + alpha * (reward + gamma * next_max - Q[state[0], state[1], action])
```

Penjelasan :

Pada kode ini, fungsi update_q_table() mengimplementasikan aturan pembaruan Q-Learning. Q-Learning adalah algoritma reinforcement learning off-policy yang memperbarui tabel Q berdasarkan reward yang diterima dan nilai maksimum dari state berikutnya. Ini memungkinkan agent untuk mempelajari nilai optimal dari setiap aksi di setiap state.

```python
def print_grid():
    """Print the current grid state to the terminal."""
    print("\nCurrent Grid State:")
    for i in range(GRID_SIZE):
        row = ""
        for j in range(GRID_SIZE):
            if [i, j] == player:
                row += " P "  # Player's position
            elif grid[i][j] == 1:
                row += " # "  # Wall
            elif grid[i][j] == 3:
                row += " G "  # Goal
            elif grid[i][j] == 2:
                row += " R "  # Reward cell
            else:
                row += " . "  # Empty space
        print(row)
```

Penjelasan :

Fungsi ini mencetak kondisi grid saat ini pada terminal dengan menampilkan posisi pemain (dilambangkan dengan "P"), dinding ("#"), tujuan ("G"), sel reward ("R"), dan ruang kosong (" . "). Fungsi ini diulang untuk setiap baris dan kolom dari grid untuk memperlihatkan keadaan penuh dari permainan.

```python
# Main game loop
running = True
clock = pygame.time.Clock()
```

Penjelasan :

Pada kode ini, kita menginisialisasi variabel running sebagai True, yang menandakan bahwa game masih berlangsung. clock dari pygame digunakan untuk mengontrol kecepatan loop utama dan mengatur interval antar frame.

```python
# Lakukan value iteration terlebih dahulu untuk menghitung value table
value_iteration()

# Inisialisasi Q-table dari Value Table hasil Value Iteration
initialize_q_from_value()
```

Penjelasan :

Sebelum memulai game, kita melakukan value_iteration() untuk menghitung value table, yang digunakan untuk menentukan nilai setiap state dalam grid. Setelah itu, fungsi initialize_q_from_value() menginisialisasi Q-table dari

value table yang dihasilkan untuk mendukung algoritma Q-Learning dalam pemilihan aksi yang optimal bagi pemain.

```
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Pilih aksi terbaik untuk pemain menggunakan epsilon-greedy strategy
    action_index = choose_action(player, epsilon)
    action = action_map[action_index]

    next_pos = get_next_state(player, action)

    if is_valid_move(next_pos):
        reward = get_reward(next_pos)
        update_q_table(player, action_index, reward, next_pos)
        player = next_pos  # Pindahkan pemain

        # Cetak kondisi grid ke terminal
        print_grid()

    # Check for goal reached
    if grid[player[0]][player[1]] == 3:
        print('Goal reached!')
        running = False

    screen.fill(WHITE)
    draw_grid()
    draw_player()
    pygame.display.flip()

    clock.tick(1)  # AI moves once per second

pygame.quit()
sys.exit()
```

Penjelasan :

Kode ini adalah bagian dari loop utama permainan AI berbasis Pygame. Dimulai dengan while running, loop ini terus berjalan selama variabel running bernilai True. Setiap event Pygame, seperti penutupan jendela permainan, diperiksa menggunakan pygame.event.get(), dan jika pengguna menutup jendela, nilai running diubah menjadi False untuk mengakhiri permainan.

Selanjutnya, aksi untuk pemain dipilih menggunakan strategi epsilon-greedy melalui fungsi choose_action(player, epsilon), yang menghasilkan indeks

aksi berdasarkan nilai epsilon. Aksi tersebut kemudian diterjemahkan menjadi gerakan dengan get_next_state(player, action). Sebelum bergerak, program memeriksa kevalidan gerakan menggunakan is_valid_move(next_pos). Jika gerakan valid, imbalan dihitung dengan get_reward(next_pos), dan Q-table diperbarui menggunakan update_q_table(player, action_index, reward, next_pos). Posisi pemain diperbarui, dan kondisi grid terkini dicetak dengan print_grid().

Jika pemain mencapai tujuan yang ditandai dengan nilai 3, program mencetak "Goal reached!" dan mengakhiri permainan. Setelah itu, layar dibersihkan, grid dan posisi pemain digambar ulang, dan tampilan diperbarui. Frekuensi gerakan AI diatur menjadi satu kali per detik dengan clock.tick(1). Setelah loop selesai, pygame.quit() menutup jendela permainan, dan sys.exit() memastikan program berakhir dengan benar.

- **Penjelasan Bagian Q-Function Value Iteration**

```
def value_iteration():
    """Algoritma value iteration untuk memperbarui nilai dari setiap state."""
    global V
    delta = float('inf')  # Perubahan maksimum antar iterasi

    while delta > theta:  # Lakukan sampai perubahan nilai menjadi sangat kecil (konvergen)
        delta = 0
        for x in range(GRID_SIZE):
            for y in range(GRID_SIZE):
                if grid[x][y] == 1:  # Jika ini dinding, lewati
                    continue
                v = V[x, y]  # Simpan nilai state saat ini

                # Evaluasi untuk setiap aksi, hitung nilai maksimal
                max_value = -float('inf')
                for action in actions:
                    next_state = get_next_state([x, y], action)
                    if is_valid_move(next_state):
                        reward = get_reward(next_state)
                        value = reward + gamma * V[next_state[0], next_state[1]]
                        if value > max_value:
                            max_value = value

                # Update nilai state dengan nilai terbaik dari semua aksi
                V[x, y] = max_value
                delta = max(delta, abs(v - V[x, y]))  # Update delta untuk mengecek konvergensi

def initialize_q_from_value():
    """Inisialisasi Q-table menggunakan value dari Value Iteration."""
    global Q
    for x in range(GRID_SIZE):
        for y in range(GRID_SIZE):
            if grid[x][y] != 1:  # Bukan dinding
                for action_index, action in enumerate(actions):
                    next_state = get_next_state([x, y], action)
                    if is_valid_move(next_state):
                        reward = get_reward(next_state)
                        Q[x, y, action_index] = reward + gamma * V[next_state[0], next_state[1]]
```

Penjelasan :

Pada algoritma game Grid World diatas, bagian Q-Function Value Iteration muncul pada proses Value Iteration dan Q-Function terletak dalam dua fungsi utama: value_iteration() dan initialize_q_from_value(). Value Iteration adalah metode yang digunakan untuk menentukan nilai dari setiap state dalam sebuah lingkungan Markov Decision Process (MDP). Dalam fungsi value_iteration(), setiap state di-grid dievaluasi berulang kali untuk menghitung nilai maksimum dari semua kemungkinan aksi yang dapat diambil, dengan memperhitungkan reward yang dihasilkan dan nilai dari state berikutnya. Proses ini berlanjut hingga perbedaan (delta) antara iterasi menjadi sangat kecil, menandakan bahwa nilai state telah konvergen.

Setelah nilai-nilai state dihitung, Q-Function diinisialisasi dalam fungsi initialize_q_from_value(). Q-Function, yang dinyatakan dalam Q-table, merepresentasikan nilai dari melakukan aksi tertentu di state tertentu. Dalam konteks kode ini, Q-table diisi dengan nilai reward yang diterima dari aksi yang diambil, ditambah dengan nilai discounted dari state berikutnya, sesuai dengan rumus:

$$Q(s,a) = R + \gamma \max_{a'} Q(s',a')$$

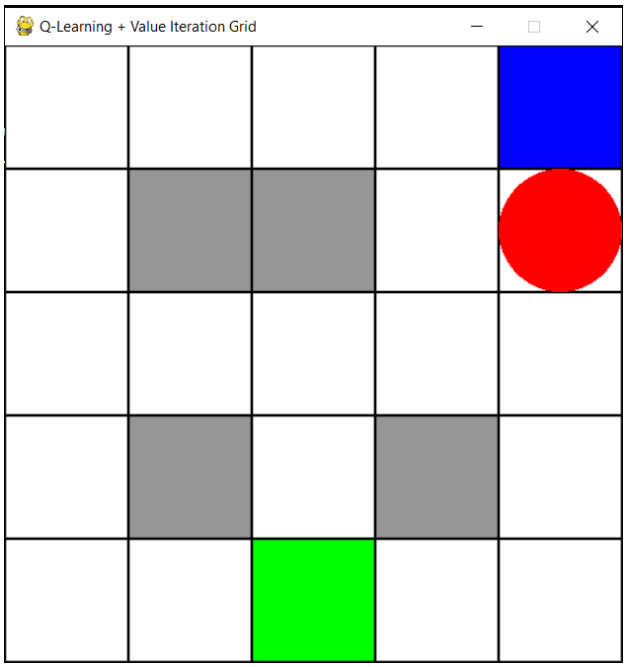
R : Reward dari state saat ini

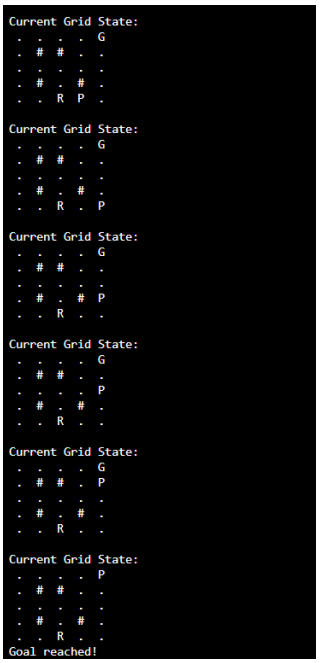
γ : Faktor diskonto (discount factor)

s′ : Keadaan berikutnya yang dicapai berdasarkan kebijakan saat ini.

Rumus ini menciptakan hubungan antara reward yang diterima dan potensi nilai jangka panjang dari melakukan aksi tersebut. Konsep ini berfokus pada pembelajaran seberapa baiknya suatu tindakan dalam memaksimalkan reward seiring waktu, sehingga agent dapat belajar untuk memilih aksi terbaik di setiap state dalam proses pengambilan keputusan. Dengan memanfaatkan kedua teknik ini, Value Iteration untuk memperkirakan nilai state dan Q-Learning untuk memperbaharui Q-Function—agent dapat mengembangkan strategi yang optimal untuk mencapai tujuan tertentu di grid.

➢ Hasil Output





Penjelasan :

Output diatas menunjukkan permainan Grid World yang menerapkan konsep Q-function dan Value Iteration. Dalam grid tersebut, agen ditandai dengan warna merah, lalu posisi awal agen ditandai dengan warna hijau, sedangkan tujuan ditandai dengan warna biru. Sel-sel lain berwarna abu-abu menunjukkan penghalang dan mewakili penalti, sedangkan sel putih adalah sel kosong yang dapat dilalui. Output grafik memperlihatkan status pergerakan agen yang berhasil satu langkah lagi mencapai tujuan, dengan catatan "Goal reached!" yang menunjukkan efektivitas algoritma. Sementara itu, output terminal mencetak status grid pada setiap langkah, di mana nilai Q diperbarui berdasarkan tindakan yang diambil. Konsep Q-function mengevaluasi nilai tindakan untuk memaksimalkan reward, sementara Value Iteration menghitung nilai optimal melalui iterasi berulang, memungkinkan agen merencanakan jalur optimal.

Kesimpulan

Berdasarkan hasil penerapan tiga algoritma pada game Grid World, yaitu Markov Decision Process (MDP), Policy Iteration dengan Bellman Equation, dan Q-function Value Iteration, dapat disimpulkan bahwa masing-masing memiliki keunggulan dan kelemahan. MDP menggunakan transisi stokastik dengan probabilitas aksi tertentu, yang efektif dalam situasi dengan ketidakpastian tetapi cenderung lebih lambat karena banyaknya probabilitas yang harus dihitung. Policy Iteration dengan Bellman Equation memperbarui nilai state secara iteratif hingga mencapai kebijakan optimal, menawarkan stabilitas dan efisiensi yang lebih baik dibandingkan MDP karena fokusnya pada perbaikan kebijakan tanpa memperhitungkan probabilitas stokastik. Sementara itu, Q-function Value Iteration menggabungkan Value Iteration dan Q-learning, memungkinkan pembaruan nilai Q secara iteratif berdasarkan reward yang diterima dan nilai dari state berikutnya. Algoritma ini terbukti paling efisien dan optimal, karena konvergensinya lebih cepat dan mampu menyeimbangkan antara eksplorasi aksi baru dan eksploitasi aksi terbaik. Oleh karena itu, Q-function Value Iteration direkomendasikan sebagai algoritma yang paling efektif untuk digunakan dalam game Grid World.

Link Github :

https://github.com/rifqiafr/UTS-Reinforcement-Learning.git