Universidade da Beira Interior

Departamento de Informática

Projeto Inteligência Computacional

Elaborado por:

Bruno Gonçalves - M15183, Rui Gil - M15777

11 de dezembro de 2025

Conteúdo

Capítulo

1

Agente 2 - Bruno Gonçalves

Neste agente o foco principal foi a tentativa de encontrar sempre o caminho mais rápido entre a base e as bandeiras.

Começou-se então por uma simples definição de regras básicas para o agente entender o funcionamento do jogo, e qual o seu papel no ambiente em questão.

Sabendo que o agente só pode fazer 6 ações:

- 0: fica no mesmo sítio sem se mexer;
- 1: sobe
- 2: desce
- 3: anda para a esquerda
- 4: anda para a direita
- 5: atordoar o adversário
- 6: largar bandeira na base

Temos que lhe ensinar qual a importância que tem cada célula, ou seja, neste caso o agente tentaria evitar ao máximo todas as células que fossem paredes (valor = 50), e dependendo do objetivo ou das prioridades do agente num dado estado, poderia também evitar células onde estivessem adversários.

Com a adição de alguns ciclos de decisão *if-else*, conseguiu-se que o agente numa fase inicial se desvisasse das paredes e andar livremente pelo mapa. Com isto, o agente já não ficava preso quando encontrava uma parede, mas não tinha ainda um "objetivo", que no caso deste jogo é encontrar e levar o máximo número de bandeiras para a base, surgiu um problema que é o agente ficar em loop pelo facto de andar de forma aleatória.

Sendo assim recorremos à visão limitada do agente para quando ele visse bandeiras guardasse numa lista e fosse buscar a mais próxima. Ele iria em direção à bandeira desviando-se de todas as paredes que encontrasse pelo caminho, e voltaria à base quando ela aparecesse no seu campo de visão para largar a bandeira fazendo todos os passos reversos do caminho que fez para a mesma.

Esta primeira abordagem por si só continha vários problemas como o loop caso apanhasse uma bandeira mas logo ao lado estivesse outra, o que faria logo com que o caminho reverso ficasse bastante denso com movimentos inúteis, e muitas das vezes se ele não visse

nada no seu campo de visão entrava mais uma vez em loop de movimentos aleatórios, ou em modo "exploração"em zonas bastante longe das bandeiras.

Sendo assim para não estar sempre a recorrer ao caminho reverso, ao modo exploração de bandeiras (que muitas das vezes explorava sempre a mesma área), e a tentativa *hard-coded* de ir até a bandeira, usou-se o algoritmo A* (A estrela) com recurso à distância de manhattan, para uma otimização de caminho para a bandeira e para a base, de modo a que fosse mais rápido a sua captura e largada, para passar logo à proxima.

```
def manhattan_dist(x, y):
    return np.sum(np.abs(np.array(x) - np.array(y)))

def path_rebuild(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.insert(0, current)
    return path

def a_star(grid, start, end):
    open_list = []
    heapq.heappush(open_list, (manhattan_dist(start, end), start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: manhattan_dist(start, end)}

    while open_list:
        current = heapq.heappop(open_list)[1]

        if current == end:
            return path_rebuild(came_from, current)

        neighbors = [
            (current[0]-1, current[1]),
            (current[0]+1, current[1]),
            (current[0], current[1]-1),
            (current[0], current[1]+1)
        ]

        for neighbor in neighbors:
            i, j = neighbor
            if i < 0 or i >= grid.shape[0] or j < 0 or j >= grid.shape[1]:
                continue
            if grid[i, j] == 50:
                continue

            tentative_g_score = g_score[current] + 1

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
```

```
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + manhattan_dist(neighbor, end)
                heapq.heappush(open_list, (f_score[neighbor], neighbor))

    return []
```

Mas dependendo do sítio onde o agente estiver, como ele tem a posição **relativa** e não absoluta das células, vai haver momentos em que o A* não faz sentido porque é um algoritmo que precisamos de conhecer o ponto de partida e de chegada, mas o agente quando está na base não consegue ver as bandeiras e quando está ao pé de bandeiras normalmente também não consegue ver a base, por isso temos que ir alternando entre o A* quando temos em vista a nossa posição final e o modo exploração quando não sabemos onde está a nossa posição final.

Para isso foi adicionado ao código uma lista de sítios por onde o agente já passou de forma a obrigá-lo sempre a ir para sítios diferentes e assim já nunca irá ocorrer loops. Por fim e mais importante imlementou-se técnicas de sistemas **difusos**.

1.0.1 Sistema difusos

Os sistemas de lógica difusa (*fuzzy logic*) foram introduzidos como uma extensão da lógica booleana, permitindo lidar com informação imprecisa, incompleta ou ambígua. Ao contrário da lógica tradicional, onde uma condição é simplesmente verdadeira ou falsa, a lógica difusa permite graus de pertença entre 0 e 1. Isto torna possível criar sistemas de decisão mais flexíveis, particularmente úteis em ambientes dinâmicos ou parcialmente observáveis, como é o caso deste jogo.

Em contextos onde as informações são incompletas ou relativas, como acontece com o nosso agente, cuja visão é limitada e cuja posição no mapa é sempre relativa, a tomada de decisões baseada em regras rígidas torna-se frágil. A lógica difusa permite que o agente interprete o ambiente de forma mais humana, graduando conceitos como “perto”, “médio” e “longe”, em vez de depender de limites fixos.

```
def fuzzy_distance_to_base(dist):

    # Perto: 0 a 3
    if dist <= 1:
        perto = 1
    elif dist <= 3:
        perto = (3 - dist) / 2
    else:
        perto = 0

    # Médio: 2 a 7
    if 2 <= dist <= 5:
        medio = (dist - 2) / 3
    elif 5 < dist <= 7:
        medio = (7 - dist) / 2
    else:
        medio = 0
```

```
    # Longe: 5 a 20
    if dist >= 10:
        longe = 1
    elif dist >= 5:
        longe = (dist - 5) / 5
    else:
        longe = 0

    return perto, medio, longe


def fuzzy_decision(perto, medio, longe):
    """
    Regras:
    - PERTO → tentar A*
    - MÉDIO → reverse path
    - LONGE → random
    """
    rules = {
        "astar": perto,
        "reverse": medio,
        "random": longe
    }

    decision = max(rules, key=rules.get)
    return decision


def fuzzy_choose_action(dist):
    perto, medio, longe = fuzzy_distance_to_base(dist)
    return fuzzy_decision(perto, medio, longe)
```

1.1 Aplicação de Lógica Difusa no Agente

Para resolver os problemas falados anteriormente, introduziu-se um módulo difuso para decidir qual estratégia de navegação deve ser usada, com base na distância estimada entre o agente e o objetivo.

A distância usada é a distância de Manhattan, calculada entre a posição relativa atual e a posição estimada da base. A partir desta distância, o sistema difuso avalia três conjuntos fuzzy:

- Perto
- Médio
- Longe

A decisão final é simplesmente o conjunto com maior grau de pertença, uma forma mais "humana"de definir distância.

A implementação dos sistemas difusos trouxe várias melhorias importantes:

1. O agente altera entre algoritmo A* e o modo exploração conforme a importância e a distância ao objetivo.
2. Como o agente já não depende exclusivamente de caminhos inversos ou exploração aleatória, passou a evitar caminhos redundantes.
3. Mais eficaz na recolha e entrega de bandeira

Pontos negativos:

- Ás vezes após trocar muitas vezes de A* para caminho reverso, o agente acaba por demorar um bocado mais a chegar à bandeira mesmo estando perto dela.
- Explora demasiado mesmo já tendo ido buscar bandeiras anteriormente.

1.2 Conclusão Agent 2

A introdução de um sistema de lógica difusa no agente revelou-se fundamental para melhorar a eficácia do agente neste ambiente parcialmente observável do jogo. A capacidade de tomar decisões graduadas, com base em distância aproximada e contexto, permitiu também aumentar a eficácia do agente e melhorar os seus trajetos.

A integração do sistema difuso com o algoritmo A* e com o mecanismo de caminho inverso resultou num agente funcional, robusto e capaz de lidar com incerteza de forma inteligente — aproximando-se assim de um comportamento mais adaptativo e realista.

Capítulo

2

Agente 1 Rui Gil

2.1 Agente (Política IE) - Rui Gil

Este agente foi desenvolvido para operar num ambiente de grelha para simular o jogo da Caça à Bandeira (*Capture the Flag*). O objetivo dos agentes neste jogo é navegar pelo ambiente (tabuleiro) de forma autónoma, capturar bandeiras e retorná-las à base, evitando obstáculos e colaborando passivamente com um colega de equipa (Agente 2), também tem outras ações disponíveis, sendo a mais denominante a ação de *stun*, que se baseia em atordoar um agente da equipa adversária por um determinado tempo e se esse mesmo agente tiver uma bandeira ele larga-a instantaneamente.

A arquitetura deste agente baseia-se em três pilares:

- **Memória de Estado Global** (GPS): Para resolver o problema da visão limitada.
- **Algoritmo A*** (*A-Star*): Para planeamento de rotas eficientes.
- **Otimização por Enxame de Partículas** (OEP): Para afinar os parâmetros de decisão do agente.

2.1.1 Código do Agente

Nesta secção será explicado os procedimentos realizados para a política e a forma como o agente se irá comportar no ambiente de jogo. Os dois primeiros tópicos referidos no início encontram-se neste file do agente.

Primeiramente foi implementada uma classe (`AgentState`) para resolver o problema da visão do agente que é limitada, basicamente esta classe consegue definir uma navegação consistente para o agente com base em:

- `global_pos`: A posição (x, y) do agente num sistema de coordenadas global virtual, que inicia em (0,0).
- `base_global_pos`: A coordenada absoluta da base (a base do agente), assim que é descoberta, que é logo no início visto que os agentes partem da sua base, é guardada permanentemente para permitir o regresso mesmo quando a base sai do campo de visão.

- `visited`: Um conjunto (*set*) de todas as coordenadas globais que já foram visitadas, este conjunto é então utilizado para incentivar a exploração de novas áreas.
- `last_action`: Vai registar o último movimento efetuado para atualizar a variável `global_pos` no próximo ciclo.

2.1.1.1 Algoritmo A*

A função `a_star_pso` é o que permite ao agente o movimento do mesmo pelo ambiente do jogo, em comparação com o tradicional A* onde o custo é fixo, esta implementação para este contexto em específico introduz uma função de custo ponderada que faz com que o comportamento do agente seja dinâmico, mais concretamente, que seja variado com base nos pesos aprendidos pela estratégia de Otimização por Enxame de Partículas (OEP).

```
# Custo acumulado ponderado pelos pesos do PSO (w_g e w_team)
def a_star_pso(grid, start, end, teammate_pos, weights):
  w_g, w_h, w_team = weights
  open_list = [] # fila de prioridade, guarda as celulas a visitar do melhor para o pior
  start_h = manhattan_dist(start, end) * w_h  # custo estimado inicial (f = g + h, onde g=0), ponderado pelo peso w_h
  heapq.heappush(open_list, (start_h, start)) # inicializa a open list com a celula (no) de partida ordenada pelo custo f
  came_from = {}
  g_score = {start: 0}

  # (...)

  tentative_g_score = g_score[current] + (move_cost * w_g)

  for neighbor in neighbors:
    if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
      came_from[neighbor] = current
      g_score[neighbor] = tentative_g_score

      h = manhattan_dist(neighbor, end) * w_h
      f = tentative_g_score + h
      heapq.heappush(open_list, (f, neighbor))
```

Excerto de Código 2.1: Cálculo do custo ponderado no A*

Esta função de avaliação (**f(n)**) para cada nó **n** (célula do tabuleiro), é definida por:

$$f(n) = (w_g \cdot g(n)) + (w_h \cdot h(n)) + C_{extra} \tag{2.1}$$

Onde:

- **g(n)**: O custo verdadeiro do percurso desde o início até ao nó atual.
- **h(n)**: A heurística (distância de Manhattan) até ao objetivo.
- w_g **e** w_h: São os pesos que são otimizados pela OEP, que definem se o agente deve ser mais guloso (*greedy*) ou mais racional em relação em ir em direção ao alvo (bandeira).

- C_{extra}: Esta parte, representa o custo da repulsão (repelir) entre os agentes da mesma equipa, se a célula (nó) for adjacente ao agente *teammate*, adiciona-se o peso w_{team} para evitar possíveis colisões entre os agentes da mesma equipa.

2.1.1.2 Lógica de Decisão e Navegação (*Policy*)

A principal função, `policy`, é a função que se responsabiliza pelo comportamento do agente, seguindo uma arquitetura de prioridades:

1. **Atualização e Calibração do GPS**: O agente segue uma ideia de *Dead Reckoning*, que basicamente é um processo de navegação em que o agente baseia-se em apenas na sia última posição conhecida, na direção para a qual se moveu e a distância que percorreu, para conseguir estimar a sua posição global com base na sua última ação, sempre que a base do agente entra no seu campo de visão, o **GPS**, que é o sistema de coordenadas global que permite ao agente ter uma maior noção de todo o ambiente, é recalibrado para eliminar o erro de *drift*, que possa fazer com que o agente depois não saiba voltar para a base quando tiver uma bandeira.

2. **Mecanismo de Proteção**: Este mecanismo foi adicionado para fazer com que o agente não confunda as bases no momento em que vai entregar a bandeira, basicamente, se a base tiver a uma distância superior a 15, é ignorada (base inimiga), prevenindo assim que o agente se desloque para a base contrária.

```
1  dist = abs(my_state.base_global_pos[0] -
       observed_base_global[0]) + \
2                  abs(my_state.base_global_pos[1] -
                       observed_base_global[1])
3
4  if dist < 15: # Our base, recalibrate the GPS
5      my_state.base_global_pos = observed_base_global
6      print("MY BASE!!!!")
7  else:
8    print("ENEMY BASE!!!")
```

Excerto de Código 2.2: Filtro de calibração do GPS

3. **Seleção do Alvo**: Quando o agente está a segurar uma bandeira, o objetivo é a coordenada global da base, como a base pode estar fora da matriz de visão (o que impediria o funcionamento do A*), utiliza-se uma técnica de *Clamping*. Esta técnica faz a projeção da coordenada do alvo na borda do campo de visão, criando uma assim uma espécie de "bússola"que aponta na direção correta.

4. **Execução e Segurança**: Se o movimento calculado pelo A* resultar numa colisão iminente com o colega de equipa, a ação é cancelada e o agente recorre à exploração aleatória para se desviar.

2.1.2 Código do treino da OEP

Para treinar o algoritmo da otimização por enxame de partículas (OEP) utilizou-se um ficheiro á parte para se conseguir obter os pesos afinados (w_g, w_h, w_{team}) para uma boa solução para o agente maximizar a sua pontuação e a da sua equipa consequentemente.

2.1.2.1 Função de Fitness

A capacidade das partículas (a sua qualidade) foi avaliada através da diferença da pontuação média em três jogos simulados:

$$Fitness = \frac{1}{N}\sum_{i=1}^{N}(Score_{Red} - Score_{Green}) \tag{2.2}$$

Isto faz com que o agente não se preocupe apenas em fazer pontos, mas também a conseguir superiorizar-se ao adversário.

2.1.2.2 Estabilidade e Convergência

Para garantir a estabilidade do enxame e evitar possível divergência das partículas, a inércia (w) foi configurada para respeitar a condição teórica de convergência (retirada dos slides da disciplina):

$$w > \frac{1}{2}(c_1 + c_2) - 1 \tag{2.3}$$

Com as constantes de aceleração $c_1 = 1.5$ (*personal*) e $c_2 = 1.5$ (*group*), a condição impõe que $w > 0.5$. Por conseguinte, implementou-se um decaimento gradual da inércia de 0.9 até 0.51 (para seguir a fórmula e permitir que o agente percorra um espaço maior para aumentar a probabilidade de obter uma melhor solução), garantindo que o sistema permanece sempre dentro da região de estabilidade.

2.1.2.3 Resultados Obtidos

Após 50 iterações com um enxame de 30 partículas (valores *default* dos hiperparâmetros), o algoritmo convergiu para os seguintes pesos, que foram integrados no agente final:

- **Peso do Custo (w_g):** $\approx$ 2.36 (Prioriza caminhos curtos).
- **Peso da Heurística (w_h):** $\approx$ 1.84 (Forte atração pelo objetivo).
- **Peso de Repulsão (w_{team}):** $\approx$ 0.01 (Significa isto que, o agente aprendeu que desviar-se do colega é ineficiente, confiando assim no sistema reativo definido na `policy` do código do agente, que é basicamente no caso de acontecer a colisão entre os agentes da mesma equipa, o agente fica parado e passa assim para o próximo método de decisao, neste caso a exploração, para evitar colisões diretas).