Snake agent

Bruno Tavares 113372

Diogo Costa 112714

Arquitectura do agente

Armazenamento de estados

A classe Snake mantém o estado do agente, incluindo:

- **body**: O corpo da cobra, representado como uma lista de coordenadas.
- **snake_head**: Coordenadas da cabeça da cobra.
- **snake_sight**: Visão do ambiente, indicando a presença de alimentos e obstáculos.
- **food_positions**: Armazena as posições dos alimentos disponíveis.
- **walls_positions**: Mantém as posições das paredes.

Estratégia de Decisão

- O agente avalia as direções disponíveis com base na segurança (evitar colisões) e na oportunidade (buscar alimentos).
- A lógica de decisão é adaptativa, ajustando-se às condições do jogo e ao comportamento dos adversários.

Atualização de estados

- Método **update()**: Este método atualiza o estado do agente a partir das informações recebidas do servidor. Ele processa a visão do ambiente e identifica posições de alimentos e obstáculos.
- Método **update2()**: Atualiza informações sobre o tamanho do mapa, permitindo que o agente adapte sua lógica de movimento conforme necessário.

Lógica de Colisões e Desvios

- Passos Menores que 2800:
  - A cobra come alimentos normais (valor 2) e evita super foods (valor 3). O estado traverse é sempre True.
- Passos Entre 2800 e 3000:
  - A cobra consome tanto alimentos normais quanto super foods, adaptando-se ao final do jogo.

Funções auxiliares

- Método **check_collision()**: Verifica se a nova posição colide com o corpo da cobra.
- Método **wrap_distance()**: Calcula a distância entre a cabeça da cobra e um alvo, considerando as bordas do mapa (movimento em loop).
- Método **check_nearby_snakes()**: identifica e evita áreas com outras cobras

Lógica de Movimentação

- Movimento com Pontuação Menor que 50:
  - Desce 3 casas a cada 24 passos para a direita, evitando colisões.
- Movimento com Pontuação Maior ou Igual a 50:
  - Desce 3 casas a cada 20 passos, adotando uma postura mais agressiva para maximizar a coleta de alimentos.
- Movimento em Ziguezague:
  - Quando traverse é False, a cobra move se em ziguezague para evitar colisões e navegar com segurança.

Algoritmo utilizado

Busca Heurística

- A função decide_direction implementa uma busca heurística, considerando proximidade de comida (usando **Distância de Manhattan** considerando o wrapping à volta do mapa), segurança (evitando colisões com corpo, paredes e outras cobras, com listas **safe_directions**, **risky_directions** e verificação de **dangerous_positions**), e priorização de comida (valor 2 até 2800 passos, depois valor 2 e 3).

Tomada de Decisões Baseada em Regras

- O agente segue regras para evitar super foods (valor 3) e cobras próximas (valor 4) até 2800 passos, e implementa estratégias de movimento como descer a cada 20/24 passos no modo traverse true (depois de alguns testes, escolhemos esta movimentação pois a cobra ocupa mais espaço do mapa e passa mais vezes perto de food)

Greedy Search com Heurísticas de Segurança

- A escolha inicial da comida mais próxima usa Greedy Search, mas é modificada por heurísticas de segurança (listas **safe_directions**, **risky_directions** e verificação de **nearby_snakes**) e priorização de comida, tornando a estratégia mais robusta do que uma simples greedy search.

Tratamento de situações de bloqueio

- O código inclui lógica para lidar com situações em que a cobra fica presa, tentando direções alternativas

Agente reativo

- O agente é principalmente reativo, pois ele olha para o estado atual do jogo (visão, posição das cobras, comida) e toma uma decisão imediata.

Benchmark

Na primeira entregas, o agente colidia com outras cobras frequentemente, pelo que nesta entrega tentámos resolver esse problema com a implementação da lógica de **check_nearby_snakes** e mudanças na função **decide_direction**. Para além disso, a lógica de descida foi melhorada para verificar a presença de outras cobras nas posições de destino.

Depois de alguns testes reparamos que o nosso score no singleplayer tanto na primeira entrega como nesta segunda não varia tanto, rondando dos 65 aos 100 pontos (isto caso a cobra não falhe por algum motivo de encurrulamento). No que diz respeito ao multiplayer foi a nossa grande mudança nesta entrega onde depois de varios testes reparámos que a nossa cobra já tem a capacidade de se desviar do corpo de outra cobra evitanto a sua colisão.

Apenas considerando a última entrega, depois de vários testes reparámos que, devido à nossa lógica de movimentação a cobra é encurralda com alguma frequência pelo adversário, ficando sem chance de fuga. Isto deve-se ao facto de termos uma lógica de movimento mais “linear”. Para além disso, fica mais difícil conseguirmos encurralar as outras cobras (o que seria importante para aumentar o nosso score em mais 100 pontos)

Conclusões - algoritmo

O que é bom?

Simplicidade e Eficiência:

- Combina greedy search com regras e heurísticas, resultando em decisões rápidas e de baixo custo computacional.

Estratégia de traverse

- O facto de no inicio do jogo só apanharmos food de valor 2 e no fim, quando ja existem multiplas superfoods, apanhá-las todas “de uma vez”, penso que seja uma estratégia eficaz.

Flexibilidade

- Facilidade de alterar a lógica permitindo prototipagem rápida, ajustes, manutenção simplificada e adaptação a novos requisitos, aumentando a robustez e adaptabilidade da solução

O que faríamos diferente

- Gostamos da nossa solução final, mas vejo que em alguns aspetos a nossa solução falha, por exemplo o movimento atual da cobra, de natureza mais linear, demonstra-se menos eficaz em cenários multiplayer, onde a previsibilidade da trajetória da cobra a torna vulnerável a interceptações por outros jogadores. Adicionalmente, embora raramente, a cobra pode se encurralar devido à falta de planeamento (onde algoritmos como A* poderiam ser uteis).

Mudariamos?

- Sim, a greedy search com heurísticas foi uma escolha eficaz devido à sua simplicidade de implementação e baixo custo computacional, permitindo-nos desenvolver um protótipo funcional e bastante eficaz com várias exceções úteis. No entanto, reconhecemos que o nosso agente prioriza apenas o próximo movimento sem considerar as consequências a longo prazo, o que torna inadequado para cenários mais complexos, como o modo multiplayer.