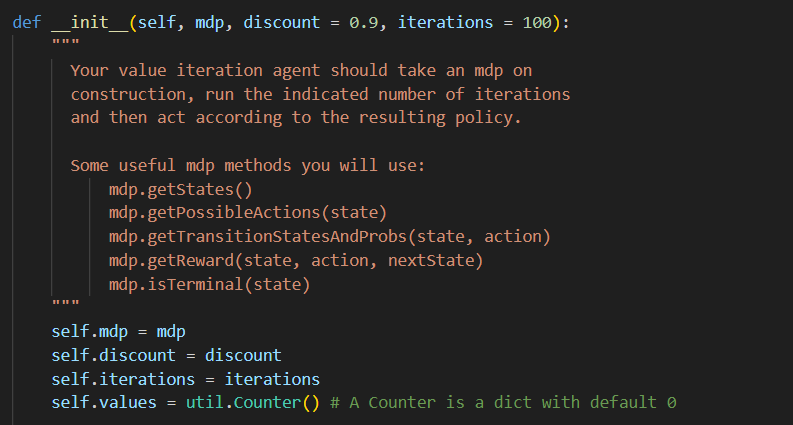
**REPORT LAB 1.5**

**Question 1:** Viết một agent sử dụng thuật toán Value Iteration. Agent này sẽ tính toán các giá trị tối ưu cho mỗi trạng thái trong MDP (Markov Decision Process) và sử dụng chúng để xác định chính sách tối ưu.

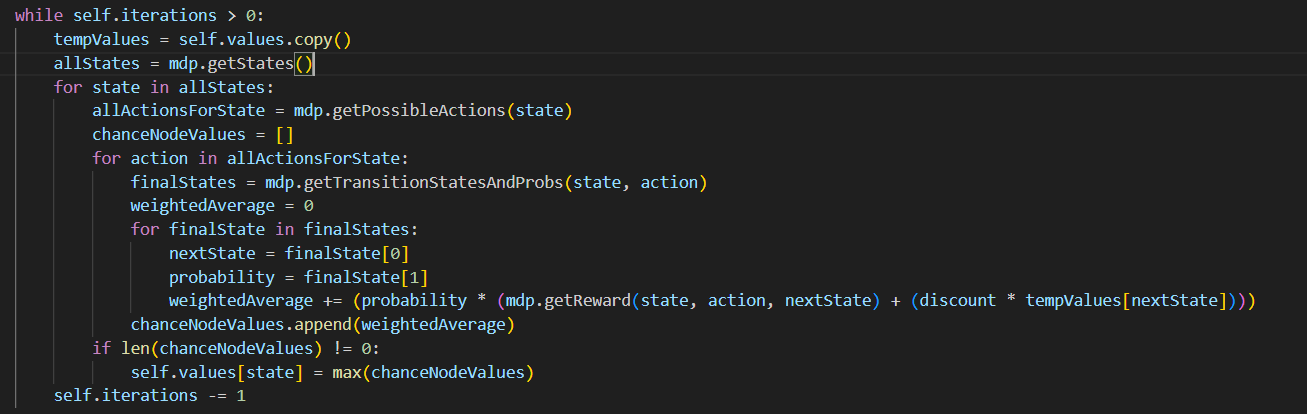


**mdp**: môi trường mà agent sẽ tương tác (Markov Decision Process).

**discount**: Hệ số chiết khấu giúp giảm tầm quan trọng của phần thưởng tương lai

**iterations**: Số lần lặp mà Value Iteration sẽ thực hiện.

**values:** Khởi tạo các giá trị của trạng thái bằng 0



Vòng lặp này thực hiện thuật toán **Value Iteration** để tính toán và cập nhật giá trị của các trạng thái trong một MDP (Markov Decision Process). Mục tiêu là tìm giá trị tối ưu cho từng trạng thái thông qua nhiều lần lặp:

- Lấy danh sách các trạng thái trong môi trường (MDP) để bắt đầu tính toán giá trị cho từng trạng thái.

- Duyệt qua từng trạng thái trong danh sách để tính giá trị cho nó.

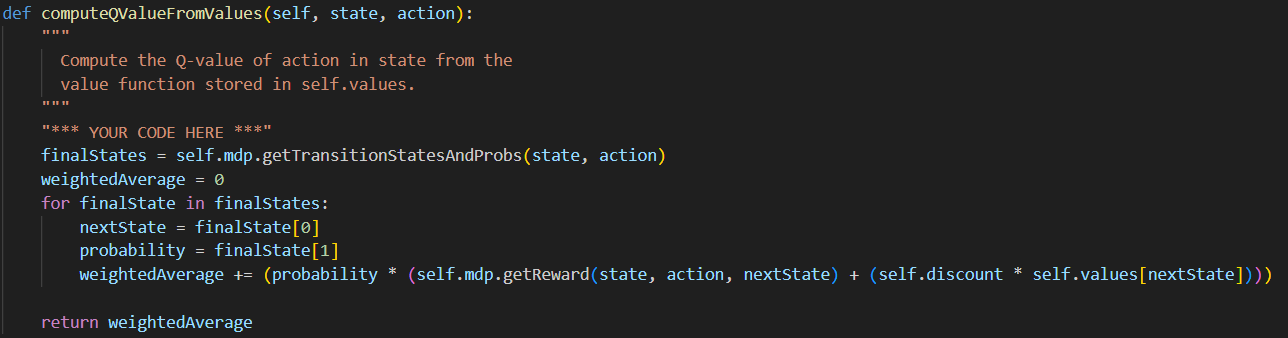
- Với mỗi trạng thái, lấy danh sách các hành động có thể thực hiện.

- Với mỗi hành động, tính giá trị kỳ vọng (expected value) dựa trên xác suất chuyển trạng thái và phần thưởng của các trạng thái tiếp theo.

- Tính xác suất và phần thưởng cho mỗi trạng thái tiếp theo dựa trên hành động từ trạng thái hiện tại.

- Giá trị của trạng thái được cập nhật bằng giá trị kỳ vọng lớn nhất từ các hành động.

- Mỗi lần lặp, số lần lặp được giảm đi 1 và quá trình tiếp tục cho đến khi không còn vòng lặp.



Hàm **computeQValueFromValues(self, state, action)** được thiết kế để tính toán **giá trị Q (Q-value)** cho một hành động cụ thể từ một trạng thái đã cho trong Markov Decision Process (MDP). Giá trị Q biểu thị mức độ kỳ vọng về phần thưởng khi thực hiện hành động đó từ trạng thái hiện tại, có tính đến xác suất chuyển tiếp tới các trạng thái tiếp theo và giá trị của chúng.

- Sử dụng phương thức **mdp.getTransitionStatesAndProbs(state, action)** để lấy danh sách các cặp **(nextState, probability)**:

  + **nextState**: Trạng thái tiếp theo mà agent có thể chuyển đến khi thực hiện hành động.

+ **probability**: Xác suất chuyển tiếp từ trạng thái hiện tại tới trạng thái tiếp theo khi thực hiện hành động đó.

- Biến **weightedAverage** được khởi tạo bằng 0 để lưu trữ giá trị Q (giá trị kỳ vọng) cho hành động.

- Hàm lặp qua từng cặp **(nextState, probability):**

+ **nextState:** Trạng thái tiếp theo.

+ **probability:** Xác suất dẫn tới trạng thái đó.

- Tính giá trị kỳ vọng sử dụng công thức:

weightedAverage += probability \* (reward + discount \* valueOfNextState)

+ **reward**: Được tính bởi **mdp.getReward(state, action, nextState)** và biểu thị phần thưởng tức thời khi chuyển từ trạng thái hiện tại tới trạng thái tiếp theo thông qua hành động.

+ **discount**: Hệ số giảm giá, phản ánh tầm quan trọng của phần thưởng trong tương lai.

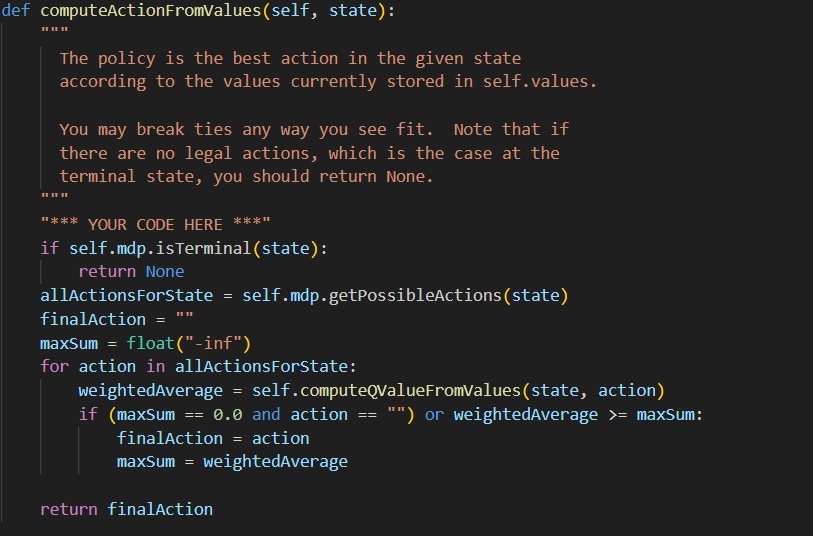
+ **valueOfNextState**: Giá trị của trạng thái tiếp theo đã được tính trước trong quá trình Value Iteration.

- Giá trị Q của hành động được tính bằng cách cộng dồn các giá trị kỳ vọng từ các trạng thái tiếp theo:

Q(state, action) = Σ (probability \* (reward + discount \* valueOfNextState))

- Mỗi phần của tổng đại diện cho giá trị kỳ vọng của việc thực hiện hành động đó, xét tới xác suất, phần thưởng và giá trị giảm giá của trạng thái tiếp theo.

- Sau khi tính toán giá trị kỳ vọng cho tất cả các trạng thái tiếp theo, hàm sẽ trả về giá trị **weightedAverage**. Đây chính là giá trị Q của hành động tại trạng thái đó.



Hàm này có nhiệm vụ tìm và trả về hành động tốt nhất từ một trạng thái cụ thể dựa trên các giá trị (value) đã được tính toán trước đó. Giá trị này được tính toán bằng thuật toán Value Iteration.

- Sử dụng **mdp.isTerminal(state)** để kiểm tra xem trạng thái hiện tại có phải là trạng thái kết thúc không.

- Nếu là trạng thái kết thúc, không có hành động nào cần thực hiện và hàm sẽ trả về None.

- Dùng **mdp.getPossibleActions(state)** để lấy tất cả các hành động khả thi từ trạng thái này.

- Nếu không có hành động nào khả dụng, hàm sẽ không tính tiếp.

- **finalAction**: Khởi tạo là chuỗi rỗng, dùng để lưu lại hành động tốt nhất.

- **maxSum**: Khởi tạo là giá trị âm vô cùng (-inf), dùng để theo dõi giá trị lớn nhất của các hành động.

- Lặp qua từng hành động trong danh sách hành động khả thi.

- Sử dụng hàm **computeQValueFromValues(state, action)** để tính giá trị Q (Q-value) cho từng hành động. Giá trị Q thể hiện giá trị kỳ vọng của hành động dựa trên các trạng thái tiếp theo và xác suất chuyển tới chúng.

- Sau khi tính giá trị Q của hành động, so sánh nó với giá trị lớn nhất hiện tại (maxSum).

- Nếu giá trị Q của hành động lớn hơn hoặc bằng maxSum, cập nhật finalAction thành hành động đó và cập nhật maxSum với giá trị mới.

- Điều này đảm bảo hành động tốt nhất được chọn dựa trên giá trị kỳ vọng lớn nhất.

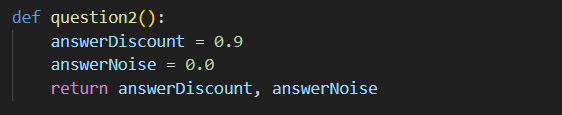
- Sau khi duyệt qua tất cả các hành động, hàm trả về hành động có giá trị Q cao nhất. Đây là hành động tốt nhất mà agent nên thực hiện tại trạng thái hiện tại để tối ưu hóa phần thưởng trong tương lai.

**Question 2:** Điều chỉnh một trong hai tham số discount hoặc noise để agent có thể vượt qua cây cầu trong BridgeGrid. Bạn cần thay đổi một tham số để chính sách tối ưu khiến agent cố gắng vượt qua cầu.

**Giảm nhiễu (noise):** Nếu giảm độ nhiễu, các hành động của tác nhân sẽ chính xác hơn, làm tăng khả năng tác nhân đạt được mục tiêu mong muốn. Điều này đồng nghĩa với việc tác nhân có thể tự tin hơn khi chọn con đường băng qua cây cầu hẹp.

**Tăng hệ số chiết khấu (discount):** Khi hệ số chiết khấu tăng, tác nhân sẽ chú trọng nhiều hơn đến các phần thưởng lớn trong tương lai (phần thưởng ở đầu cầu bên kia) thay vì chỉ quan tâm đến những phần thưởng nhỏ gần đó. Điều này có thể thúc đẩy tác nhân mạo hiểm vượt qua cầu để đạt được phần thưởng lớn.

**Đề xuất:** Giảm nhiễu là phương án hợp lý nhất trong tình huống này. Khi giảm độ nhiễu, chẳng hạn từ 0.2 xuống 0.0, tác nhân sẽ ít gặp các sự cố ngoài ý muốn khi di chuyển, giúp tăng khả năng tác nhân chọn con đường qua cầu để đạt được phần thưởng cao hơn.

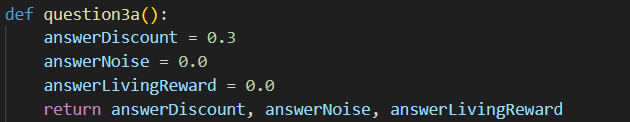


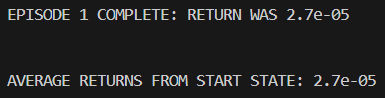


**Question 3:** Điều chỉnh các tham số discount, noise, và living reward trong DiscountGrid để tạo ra các chính sách tối ưu khác nhau. Mỗi phần từ 3a đến 3e yêu cầu bạn trả lời một bộ ba tham số để đạt được hành vi mong muốn.

1. Prefer the close exit (+1), risking the cliff (-10):

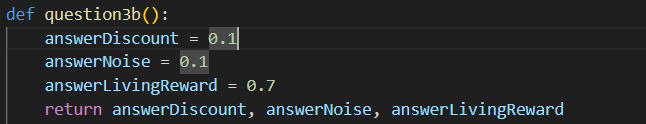
* **Giảm hệ số chiết khấu**: Mức chiết khấu thấp (0.3) khiến tác nhân chú trọng nhiều hơn vào phần thưởng gần (exit +1) thay vì phần thưởng xa hơn.
* **Giảm nhiễu**: Với noise = 0.0, tác nhân sẽ không gặp lỗi trong quá trình di chuyển, điều này giúp tác nhân có thể tự tin đi con đường ngắn, mặc dù nó gần với vách đá (-10).
* **Phần thưởng sống**: Không có phần thưởng sống thêm (living reward = 0.0), nghĩa là tác nhân không có lý do gì để kéo dài hành trình của mình.

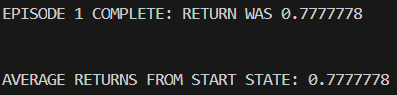


****

1. Prefer the close exit (+1), but avoiding the cliff (-10):

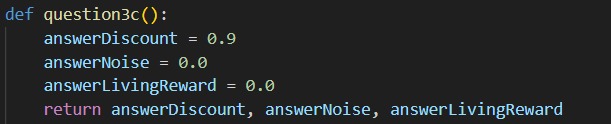
* **Giảm hệ số chiết khấu**: Với discount = 0.1, tác nhân vẫn ưu tiên phần thưởng gần (exit +1), nhưng giá trị nhỏ này không đủ lớn để khuyến khích việc mạo hiểm.
* **Noise vừa phải**: Noise = 0.1 khiến tác nhân thận trọng hơn khi đưa ra quyết định, khiến nó tránh con đường nguy hiểm gần vách đá.
* **Phần thưởng sống**: Living reward = 0.7 cung cấp động lực cho tác nhân chọn con đường an toàn hơn (tránh vách đá).

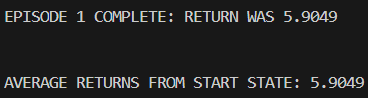




1. Prefer the distant exit (+10), risking the cliff (-10):

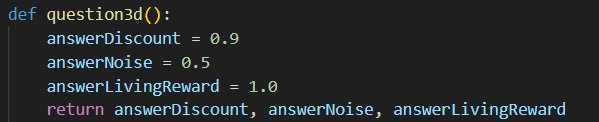
* **Tăng hệ số chiết khấu**: Với discount = 0.9, tác nhân đánh giá cao phần thưởng lớn ở tương lai (exit +10), khuyến khích nó chọn con đường dẫn tới phần thưởng xa.
* **Giảm nhiễu**: Noise = 0.0 đảm bảo tác nhân có thể đi qua con đường ngắn và rủi ro gần vách đá một cách chính xác mà không gặp sự cố.
* **Phần thưởng sống**: Living reward = 0.0 khiến tác nhân không có động lực kéo dài hành trình mà nhắm đến phần thưởng lớn.

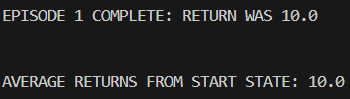




1. Prefer the distant exit (+10), avoiding the cliff (-10):

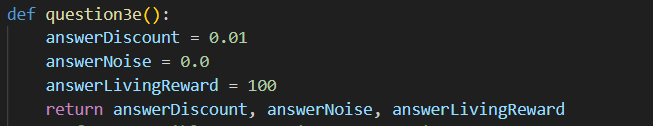
* **Tăng hệ số chiết khấu**: Với discount = 0.9, tác nhân vẫn ưu tiên phần thưởng xa hơn (+10).
* **Tăng nhiễu**: Noise = 0.5 khiến tác nhân phải tránh những lựa chọn rủi ro cao, như đi gần vách đá.
* **Phần thưởng sống cao**: Living reward = 1.0 cung cấp động lực cho tác nhân chọn con đường an toàn (tránh vách đá) mặc dù nó dài hơn.





1. Avoid both exits and the cliff (so an episode should never terminate):

* **Giảm hệ số chiết khấu**: Với discount = 0.01, tác nhân không coi trọng phần thưởng xa trong tương lai.
* **Giảm nhiễu**: Noise = 0.0 để đảm bảo tác nhân có thể tránh rủi ro một cách chính xác.
* **Phần thưởng sống rất cao**: Living reward = 100 sẽ khiến tác nhân muốn kéo dài hành trình vô hạn và tránh mọi lối ra cũng như các trạng thái kết thúc.

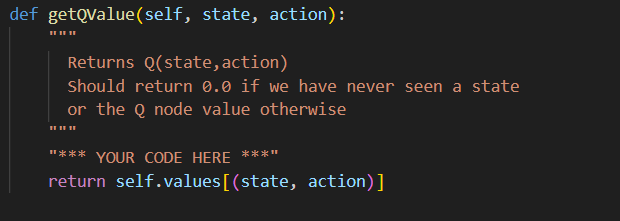


**Question 4:** Viết một agent sử dụng thuật toán Q-learning. Agent này sẽ học từ kinh nghiệm thông qua các tương tác với môi trường và cập nhật các giá trị Q.

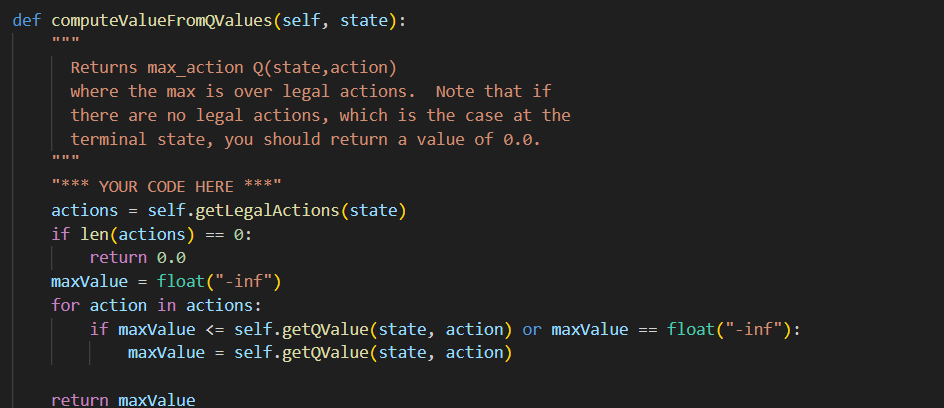
Q-Learning là một thuật toán học tăng cường (Reinforcement Learning) nhằm mục tiêu tìm kiếm chiến lược tối ưu dựa trên việc cập nhật giá trị Q cho các trạng thái và hành động. Có hai lớp chính trong mã: **QLearningAgent** và các lớp con của nó **PacmanQAgent** và **ApproximateQAgent.**

Phương thức chính của **QLearningAgent**:

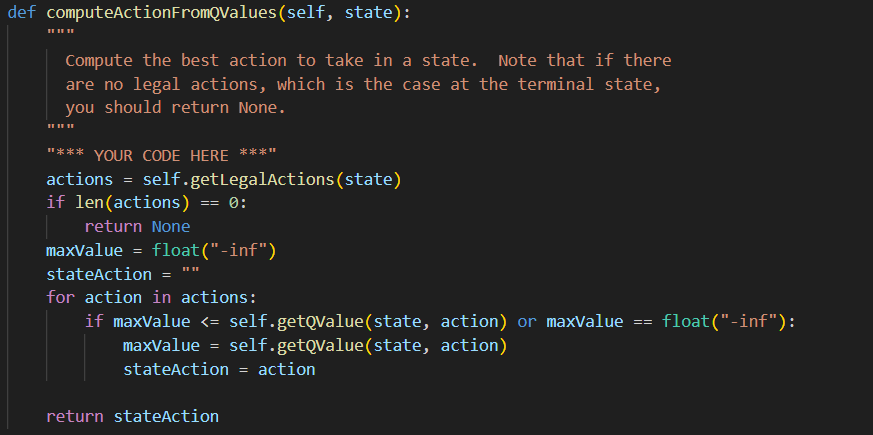
- **getQValue(self, state, action):** Trả về giá trị Q cho cặp (trạng thái, hành động). Nếu trạng thái và hành động chưa được gặp trước đây, Q-value mặc định là 0.



- **computeValueFromQValues(self, state):** Tính giá trị tối đa của tất cả các hành động có thể tại một trạng thái, tức là chọn hành động có Q-value lớn nhất. Nếu không có hành động hợp lệ (trạng thái kết thúc), giá trị trả về là 0.

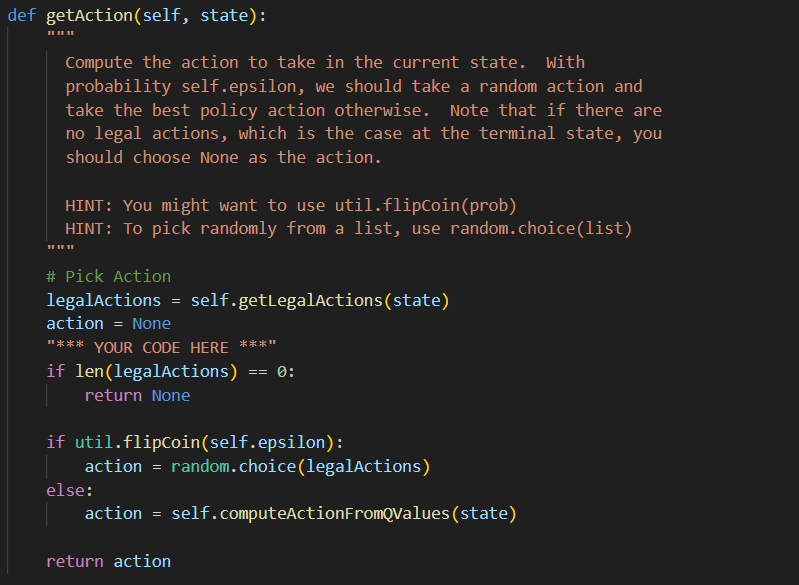


- **computeActionFromQValues(self, state):** Chọn hành động tốt nhất từ trạng thái dựa trên giá trị Q lớn nhất. Nếu không có hành động hợp lệ, trả về None.

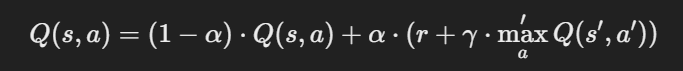


**- getAction(self, state):** Đây là phần quan trọng của việc ra quyết định:

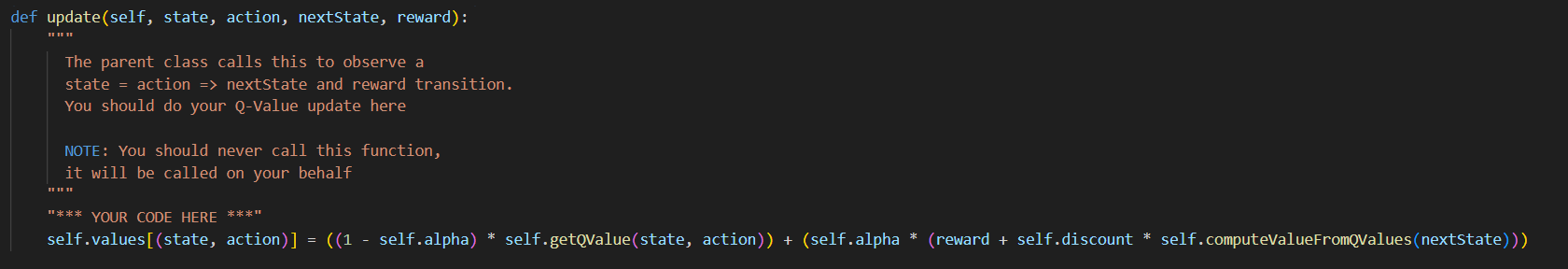
* Với xác suất epsilon, tác nhân sẽ chọn hành động ngẫu nhiên (khám phá).
* Ngược lại, tác nhân chọn hành động tốt nhất dựa trên Q-values đã biết (khai thác).
* Nếu không có hành động hợp lệ, trả về None.



- **update(self, state, action, nextState, reward):** Phương thức này thực hiện việc cập nhật giá trị Q dựa trên công thức Q-learning:

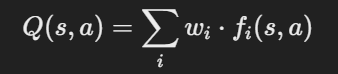


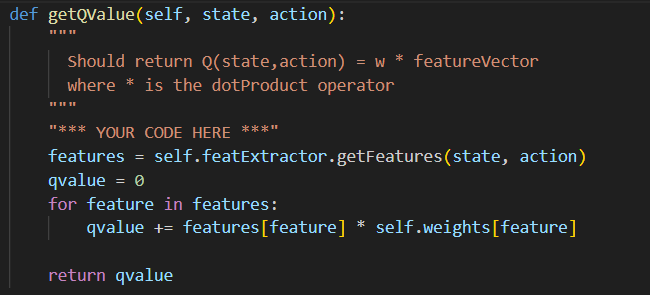
Ở đây, giá trị Q cho cặp trạng thái và hành động được điều chỉnh dựa trên phần thưởng nhận được và giá trị dự đoán ở trạng thái tiếp theo nextState.



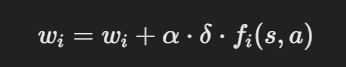
Phương thức chính của **ApproximateQAgent**:

- **getQValue(self, state, action)**: Tính giá trị Q dựa trên tích vô hướng giữa vector trọng số và vector tính năng (features). Công thức là:

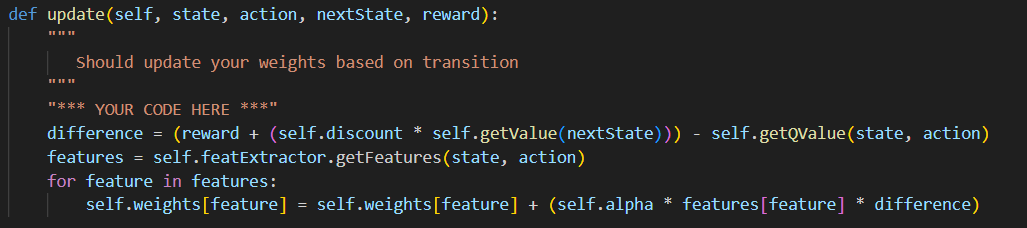




- **update(self, state, action, nextState, reward):** Cập nhật các trọng số dựa trên sự khác biệt giữa giá trị Q hiện tại và giá trị dự đoán từ trạng thái tiếp theo. Công thức cập nhật trọng số:



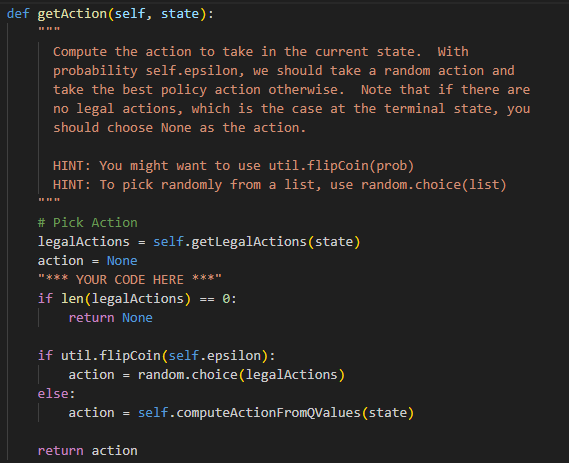
Với δ là sự khác biệt giữa giá trị dự đoán và giá trị Q hiện tại.



**Question 5:** Hoàn thiện agent Q-learning bằng cách triển khai lựa chọn hành động epsilon-greedy. Agent sẽ chọn hành động ngẫu nhiên với xác suất epsilon và chọn hành động tốt nhất dựa trên các giá trị Q hiện tại với xác suất còn lại.

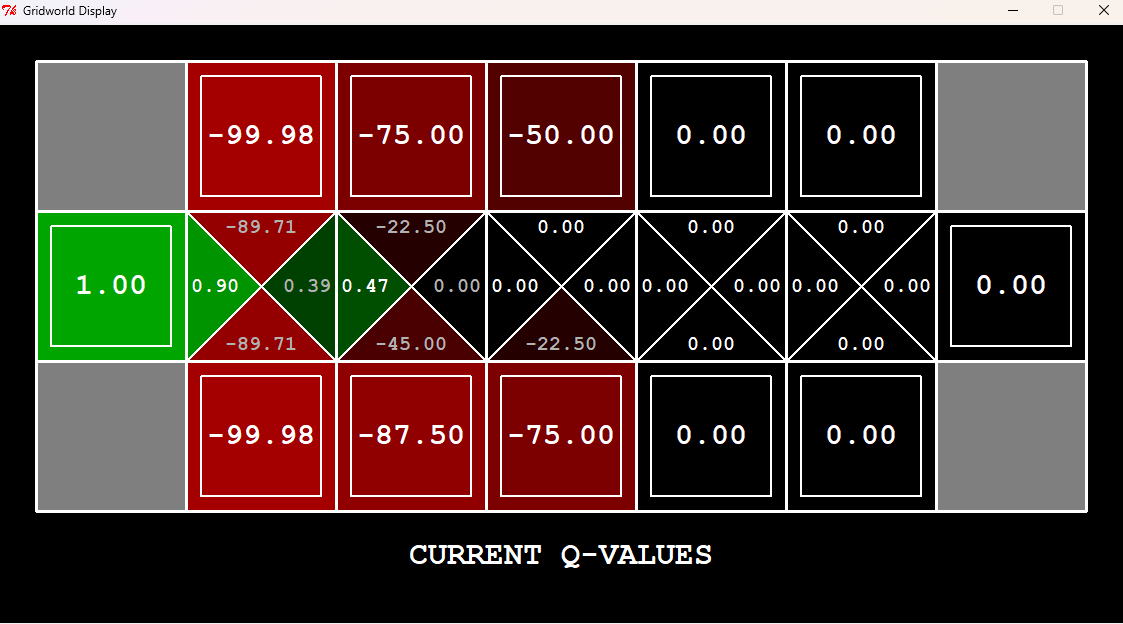
Trong phương pháp Q-learning, hàm **getAction(self, state)** thực hiện nhiệm vụ xác định hành động mà agent sẽ thực hiện tại một trạng thái cụ thể. Hành động này có thể được chọn ngẫu nhiên hoặc được xác định dựa trên chính sách tối ưu đã học, nhằm duy trì sự cân bằng giữa khám phá và khai thác.

* Đầu tiên, hàm **getLegalActions(state)** sẽ trả về một danh sách các hành động hợp lệ mà agent có thể thực hiện trong trạng thái hiện tại **state**. Nếu không có hành động hợp lệ (ví dụ như ở trạng thái kết thúc), danh sách này sẽ rỗng.
* Nếu không có hành động nào hợp lệ, agent sẽ không thực hiện hành động nào và hàm trả về **None**.
* Hàm **util.flipCoin(self.epsilon)** sẽ lật một đồng xu với xác suất thành công là **epsilon**. Nếu kết quả là thành công (tương ứng với việc hàm trả về **True**), agent sẽ chọn một hành động ngẫu nhiên từ danh sách các hành động hợp lệ bằng cách sử dụng **random.choice(legalActions)**. Điều này giúp đảm bảo rằng agent có cơ hội khám phá các hành động mới thay vì luôn chọn hành động đã học.
* Nếu không thực hiện khám phá, agent sẽ chọn hành động tốt nhất dựa trên các Q-values mà nó đã học được từ trạng thái hiện tại bằng cách gọi hàm **computeActionFromQValues(state)**.
* Cuối cùng, hàm trả về hành động mà agent đã quyết định thực hiện (ngẫu nhiên hoặc tối ưu).



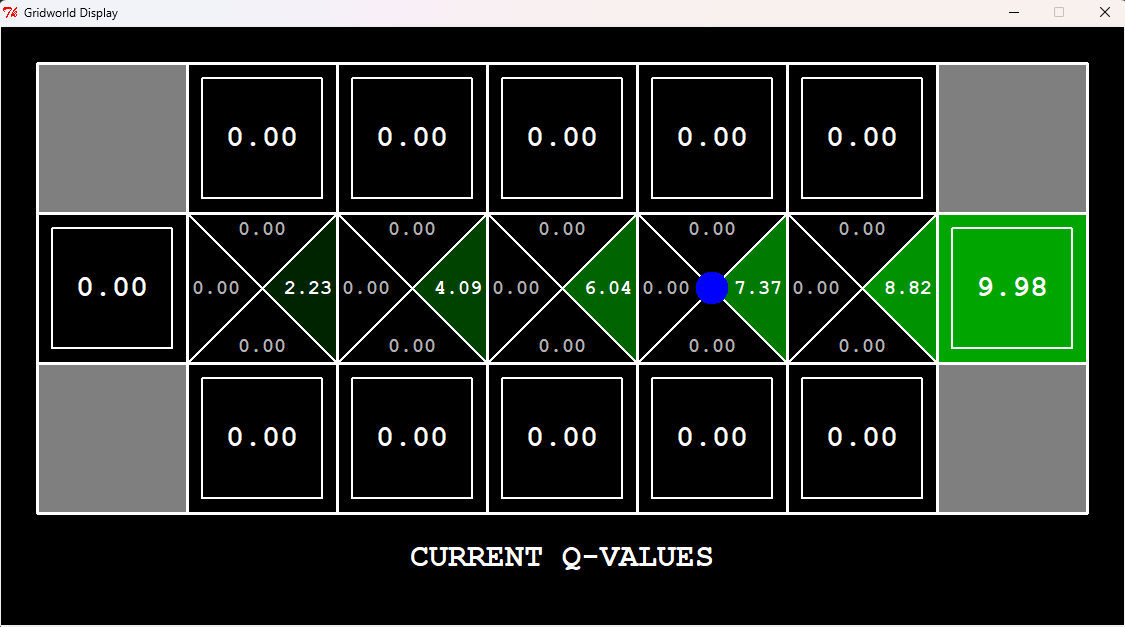
**Question 6:** Thử nghiệm với các giá trị epsilon và learning rate để tìm ra bộ tham số mà agent có thể học được chính sách tối ưu sau 50 lần lặp.

**Khi chạy lệnh với epsilon = 1:**



Agent sẽ chỉ chọn các hành động ngẫu nhiên, và trong 50 lần lặp, rất khó để nó học được chính sách tối ưu. Điều này là vì nó không dựa trên giá trị Q đã học.

**Khi chạy lệnh với epsilon = 0:**



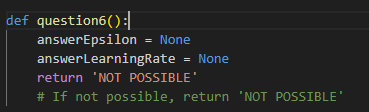
Với epsilon bằng 0, agent sẽ chỉ khai thác các Q-values hiện có mà không khám phá thêm các hành động khác. Tuy nhiên, sau 50 lần lặp, nếu learning rate không phù hợp, Q-values có thể chưa hội tụ, dẫn đến việc agent không tìm ra chính sách tối ưu.

**Kết luận:**

Dựa trên bản chất của thuật toán Q-learning và số lượng lặp chỉ là 50, có khả năng lớn rằng không tồn tại giá trị epsilon và learning rate nào đảm bảo chắc chắn (hơn 99%) rằng agent sẽ học được chính sách tối ưu sau 50 lần lặp. Điều này đặc biệt đúng nếu môi trường phức tạp và agent không có đủ thời gian để cập nhật giá trị Q cho tất cả các trạng thái và hành động.

**Kết quả:**

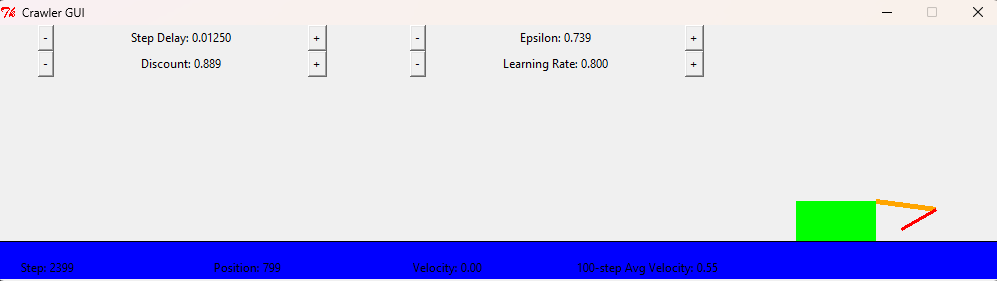
Trong trường hợp này, việc tìm được một cặp epsilon và learning rate mà chắc chắn đạt hiệu quả lớn hơn 99% sau 50 lần lặp có vẻ là không khả thi.



**Question 7:**

Các tham số cần quan tâm:

* **Step Delay**: là tham số điều chỉnh tốc độ của mô phỏng, giúp dễ quan sát hành vi của crawler robot.
* **Learning Rate (alpha)**: Tham số này quyết định mức độ mà crawler cập nhật các Q-values sau mỗi bước. Alpha cao khiến crawler học nhanh hơn nhưng có thể gây không ổn định, trong khi alpha thấp sẽ làm quá trình học diễn ra chậm hơn nhưng ổn định hơn.
* **Epsilon (epsilon)**: Đây là xác suất để crawler thực hiện hành động ngẫu nhiên thay vì chọn hành động dựa trên Q-values hiện tại. Epsilon cao khuyến khích việc khám phá (exploration), trong khi epsilon thấp khuyến khích việc khai thác (exploitation).
* **Discount Factor (gamma)**: Gamma xác định tầm quan trọng của các phần thưởng tương lai. Gamma cao khiến crawler quan tâm nhiều hơn đến các phần thưởng dài hạn, trong khi gamma thấp tập trung vào các phần thưởng ngắn hạn.



Kết quả:

* Discount factor được đặt là 0.889, tức là robot cân nhắc khá nhiều đến phần thưởng dài hạn, nhưng vẫn còn một chút ưu tiên cho phần thưởng gần.
* Epsilon được đặt là 0.739, điều này có nghĩa là robot đang ở giai đoạn khám phá nhiều hơn khai thác, giúp nó tìm hiểu các hành động mới thay vì chỉ dựa vào những hành động đã biết.
* Learning rate là 0.800, tức là robot học tương đối nhanh, nhưng vẫn có một mức độ ổn định nhất định.

**Question 8:** Huấn luyện Pacman sử dụng Q-learning trên các lưới nhỏ và kiểm tra hiệu quả của chính sách học được.

**Quá trình huấn luyện:**

* Pacman sẽ chơi 2000 game để học các giá trị Q tương ứng với từng trạng thái và hành động.
* Trong quá trình huấn luyện, Pacman có xác suất khám phá (epsilon = 0.05), nghĩa là thỉnh thoảng sẽ thực hiện một hành động ngẫu nhiên để khám phá các trạng thái mới. Ngoài ra, learning rate (alpha = 0.2) và discount factor (gamma = 0.8) giúp Pacman cập nhật Q-values dựa trên các phần thưởng nhận được.

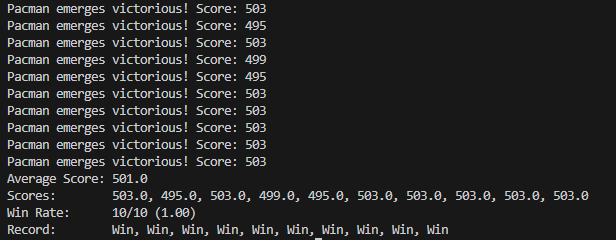
**Quá trình kiểm tra:**

* Sau khi kết thúc huấn luyện, Pacman sẽ thực hiện 10 game kiểm tra với các tham số epsilon = 0 và alpha = 0, tức là Pacman sẽ không còn khám phá nữa mà chỉ sử dụng chính sách đã học được để đưa ra các hành động tối ưu (chỉ khai thác).

**Mục tiêu:**

* Mục tiêu là đảm bảo rằng sau quá trình huấn luyện, Pacman có thể giành chiến thắng ít nhất 80% số game thử nghiệm (cụ thể là thắng ít nhất 8 trong số 10 game cuối cùng).

**Kết quả:**



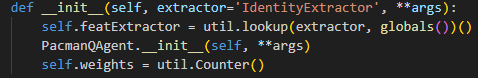
* Pacman đã học được chính sách tối ưu trong quá trình huấn luyện với các thông số mặc định (epsilon=0.05, alpha=0.2, gamma=0.8). Trong quá trình kiểm tra , khi giá trị epsilon và alpha đều được đặt về 0, Pacman không còn khám phá mà chỉ khai thác chính sách đã học, dẫn đến tỷ lệ thắng cao.

**Question 9:** Triển khai một agent Approximate Q-learning sử dụng các trọng số cho các đặc trưng của trạng thái. Agent này sẽ học các trọng số cho các đặc trưng của các cặp trạng thái-hành động.

Câu hỏi yêu cầu bạn triển khai một tác nhân Q-learning gần đúng, cụ thể là lớp ApproximateQAgent trong tệp qlearningAgents.py. Tác nhân này sẽ học các trọng số cho các đặc trưng của trạng thái, điều này có nghĩa là nhiều trạng thái có thể chia sẻ cùng một đặc trưng.

Mục tiêu là xây dựng một phương pháp học hiệu quả hơn so với phương pháp Q-learning truyền thống bằng cách sử dụng đặc trưng của trạng thái thay vì lưu trữ một bảng Q lớn cho từng trạng thái.

**-\_\_init\_\_(self, extractor='IdentityExtractor', \*\*args):** Khởi tạo một tác nhân Q-learning gần đúng.



**extractor**: Tên của bộ trích xuất đặc trưng được sử dụng để lấy các đặc trưng từ trạng thái và hành động. Mặc định là IdentityExtractor.

**self.featExtractor**: Khởi tạo bộ trích xuất đặc trưng, cho phép tác nhân trích xuất thông tin cần thiết từ trạng thái.

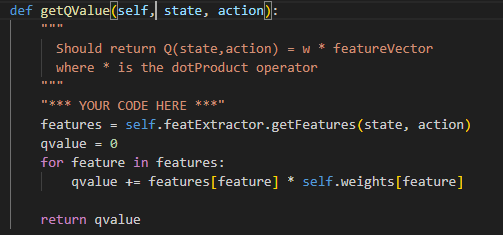
**PacmanQAgent.\_\_init\_\_(self, \*\*args)**: Gọi hàm khởi tạo của lớp cha (PacmanQAgent) để thiết lập các thuộc tính cơ bản.

**self.weights**: Lưu trữ trọng số cho các đặc trưng. util.Counter() là một cấu trúc dữ liệu giống như từ điển, cho phép dễ dàng thao tác với trọng số.

- **getWeights(**): Trả về trọng số hiện tại của tác nhân.



- **getQValue(self, state, action):** Tính toán giá trị Q cho một trạng thái và hành động cụ thể.

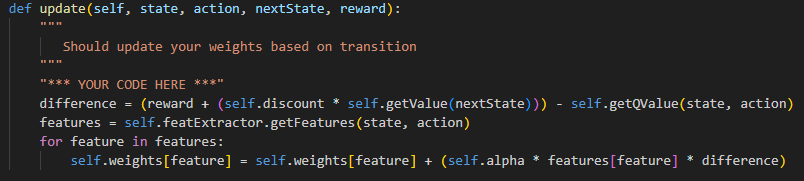


**features = self.featExtractor.getFeatures(state, action)**: Trích xuất các đặc trưng cho trạng thái và hành động hiện tại.

**qvalue**: Biến này được sử dụng để lưu trữ giá trị Q tính toán.

Vòng lặp **for**: Tính toán tổng của tích giữa mỗi đặc trưng và trọng số tương ứng, cho ra giá trị Q cho cặp (state, action).

-**update(self, state, action, nextState, reward)**: Cập nhật trọng số dựa trên quá trình chuyển tiếp giữa các trạng thái



**difference**: Tính toán sai số giữa phần thưởng thực tế và giá trị Q dự đoán, phản ánh mức độ chính xác của dự đoán.

**features**: Trích xuất các đặc trưng cho trạng thái và hành động hiện tại.

Vòng lặp **for**: Cập nhật trọng số cho từng đặc trưng dựa trên sai số, sử dụng tốc độ học (**self.alpha)**. Điều này cho phép tác nhân học từ các hành động đã thực hiện.

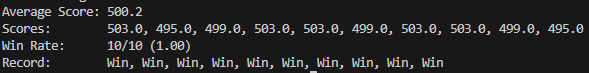
Lớp **ApproximateQAgent** cho phép tác nhân Q-learning học từ các đặc trưng của trạng thái thay vì sử dụng bảng Q truyền thống.

Các phương thức chính như **getQValue** và **update** giúp tính toán và cập nhật giá trị Q dựa trên các đặc trưng, tạo ra một cách tiếp cận linh hoạt và hiệu quả hơn cho việc học.

Điều này rất hữu ích trong các tình huống mà số lượng trạng thái rất lớn, vì nhiều trạng thái có thể chia sẻ cùng một đặc trưng và trọng số.

Chạy câu lệnh sau để kiểm tra với **IdentityExtractor**:

**python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid**



Chạy câu lệnh sau để kiểm tra với **SimpleExtractor**

**python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid**  
