Ho Chi Minh City, Viet Nam
Faculty of Computer Science
University of Information Technology

Áp dụng các thuật toán tìm kiếm trong chương trình Pacman

Hồ Hải Thủy - 19522323

Thiết kế hàm lượng giá

Mục tiêu của trò chơi, đó là làm sao để có thể đạt được điểm số cao nhất có thể và ăn hết toàn bộ food để kết thúc trò chơi.

Bảng điểm

Mỗi hành động mà Pacman thực hiện sẽ ảnh hưởng đến điểm số và điểm kết thúc của trò chơi, do đó ta cần thiết kế hàm lượng giá dựa trên các kết quả mà hành động này mang lại.

Object	Score
Time per second	-1
Food	10
Win	500
Eat Ghost	200
Lose	-500
Capsule	0

Table 1: Điểm số tương ứng với mỗi hành động của trò chơi

Các đặc trưng sử dụng

- Vì ta cần tối ưu hóa điểm trò chơi, vậy nên đặc trưng đầu tiên em sử dụng, đó chính là **currentGameState.getScore()** nhằm để chọn các state giúp gia tăng điểm số. Tuy nhiên, nếu chỉ có một hệ số này, sẽ dẫn đến tình trạng Pacman không di chuyển khi cây tìm kiếm có độ sâu thấp và các không gian trạng thái xung quanh không giúp gia tăng điểm số.
- Tiếp theo, vì chúng ta cần tối đa điểm số đạt được, nên việc tránh kéo dài thời gian và chiến thắng là điều quan trọng nhất, mà muốn đạt được hai trạng thái này thì ta cần phải kết thúc trò chơi càng nhanh càng tốt. Do đó em thêm vào đặc trưng rằng phải ăn hết thức ăn càng nhanh càng tốt: **len(foodList)**.
- Tuy nhiên, độ sâu tìm kiếm vẫn có hạn, nên nếu thức ăn ở xa, sẽ tiếp tục xảy ra hiện tượng Pacman bị đói, do đó em nghĩ cần phải chỉ hướng cho Pacman di chuyển khi gặp tình trạng bị đói. Em thêm vào đặc trưng **foodDistance**
- Tuy nhiên, ta lại tiếp tục đối mặt với một vấn đề, đó là có thể thua, vì Pacman của chúng ta được đặt trong môi trường Adversarial, do đó nếu chạm phải Ghost sẽ có thể dẫn đến kết quả thua cuộc. Do đó, ta cần phải

tránh Ghost để tránh gặp hiện tượng này. Do đó đặc trưng thứ tư em sử dụng là **ghostDistance**, nhằm để tránh xa Ghost, không bị thua game.

- Pacman khi được thiết kế với các đặc trưng trên thì có thể kết thúc trò chơi một cách rất dễ dàng, tuy nhiên đây lại không phải là điểm số tối ưu có thể đạt được. Tuy việc, tiêu thụ Capsule, sẽ không giúp chúng ta tăng điểm số, nhưng sau hành động đó, ta có thể tiêu thụ Ghost trong một thời gian cố định và từ đó có thể có được điểm số bonus rất lớn, lên đến 200 point. Tuy nhiên, việc nhắm đầu vào tiêu thụ Capsule sẽ làm cho Pacman phân tâm vào việc kết thúc trò chơi, và nếu tiêu thụ Capsule không đúng lúc, ghost ở quá xa, sẽ làm cho chúng ta tốn một lượng vô ích, do đó, em thiết kế để nhắm đến Capsule khi có thời điểm phù hợp như sau: **if closetCapsule > 10: closetCapsule = 100** Do đó, Pacman sẽ chỉ tập trung vào Capsule khi ở đủ gần.
- Cuối cùng là đặc trưng **effectGhostDistance**. Khi tiêu thụ Capsule, các Ghost sẽ trở bị dính hiệu ứng đặc biệt và Pacman có thể ăn chúng trong khoảng thời gian là 40 giây, do đó em thêm vào để Pacman hướng vào Ghost nhằm lấy điểm bonus khi dính phải hiệu ứng này.

Công thức lượng giá được em biểu diễn như sau:

$$\text{score} = 2 * \text{currentGameState.getScore()} - (\text{len(foodList)}) - \text{foodDistanceEuclid}$$
$$-\text{ghostDistance} - 10 * (\text{closestCapsule}) - \text{effectGhostDistance}$$

Nhận xét

Hàm lượng giá này vẫn chưa thực sự tốt và cần phải cải thiện, bởi khó khăn lớn nhất trong quá trình thiết kế các đặc trưng đó là việc xung đột giữa các kết quả, bởi tùy vào layout mà các giá trị này có thể thay đổi rất khác, làm cho thuật toán không hiệu quả. Có những layout số lượng Ghost, food hay Capsule rất nhiều dẫn đến tính toán bị chậm đi hoặc có map rất nhỏ, nên khả năng di chuyển bị hạn chế

Thứ hai việc khởi tạo giá trị cũng gây khó khăn bởi nếu đặt không đúng cách Pacman cũng xảy ra tình trạng đói. Ví dụ như sau, em đặt giá trị ban đầu của ClosetCapsule = 0 nếu ở khoảng cách > 10 thì Pacman sẽ không bao giờ tới vị trí của Capsule, bởi nếu đến gần thì lúc này ClosetCapsule sẽ != 0 dẫn đến kết quả lượng giá bị giảm, do đó Pacman sẽ không đến vị trí này.

So sánh

Để việc so sánh được công bằng, mỗi thuật toán sẽ được đo 5 lần (numGames = 5) ở mỗi màn chơi, với các random seed tương tự nhau có dạng 19522323 + số thứ tự lần chơi:

```
def runGames(layout, pacman, ghosts, display, numGames, record, numTraining=0, catchExceptions=False, timeout=30):
    import __main__
    __main__.__dict__['_display'] = display

    rules = ClassicGameRules(timeout)
    games = []

    for i in range(numGames):
        random.seed(19522323 + i)
        beQuiet = i < numTraining
```

Figure 1: Thay đổi code để chạy theo các seed

So sánh giữa các giải thuật với hàm lượng giá mới

Ta có thể so sánh các giải thuật dựa trên bảng tổng hợp sau:

NewEvaluation	MinimaxAgent			AlphaBetaAgent			Expectimax		
Layout	**mean**	**std**	**game_win**	**mean**	**std**	**game_win**	**mean**	**std**	**game_win**
trappedClassic	325.2	413.6	4	325.2	413.6	4	325.2	413.6	4
contestClassic	828.4	780.6	1	828.4	780.6	1	2058.6	313	3
powerClassic	2089.4	1123.3	3	2089.4	1123.3	3	226.6	342.4	0
capsuleClassic	-297.2	175.6	0	-297.2	175.6	0	-345.8	180.5	0
mediumClassic	1599.4	638.6	4	1599.4	638.6	4	504.8	1132.4	2
smallClassic	299.4	769.9	2	299.4	769.9	2	-349.6	120.8	0
minimaxClassic	314.4	403.2	4	314.4	403.2	4	314.4	403.2	4
testClassic	562.8	1.6	5	562.8	1.6	5	560.4	3.6	5
trickyClassic	0	0	0	0	0	0	0	0	0
originalClassic	1095.2	503.7	2	1095.2	503.7	2	204.8	793.5	0
openClassic	1440.2	5.8	5	1440.2	5.8	5	1072.4	446.5	4

Table 2: Kết quả tổng hợp với hàm lượng giá mới

Ta có thể thấy hầu như ở các trường hợp, thuật toán MinimaxAgent và AlphaBetaAgent có phần tốt hơn, nguyên nhân là vì ở Expectimax việc phân chia xác suất sẽ làm cho Pacman đôi khi không nhận biết được nguy hiểm có thể dẫn đến việc bị kết thúc game do Ghost.

Còn hai thuật toán MinimaxAgent và AlphaBetaAgent gần như tương tự nhau, nguyên nhân là do 2 thuật toán này tương tự nhau, nhưng thời gian xử lý của AlphaBetaAgent sẽ tốt hơn rất nhiều so với MinimaxAgent do cơ chế pruning. Ta có thể theo dõi ở đồ thị dưới đây.

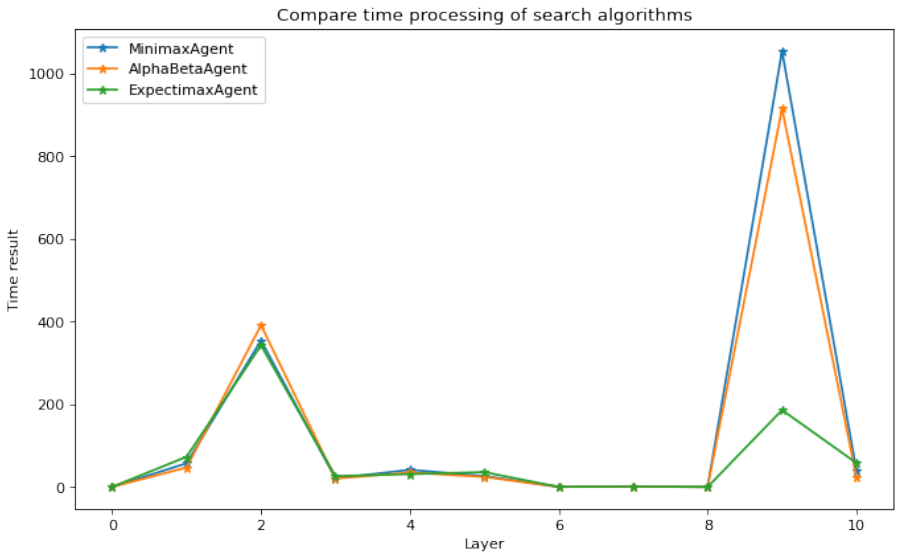

Figure 2: So sánh thời gian xử lý giữa các thuật toán

Bài hạn giới hạn với depth = 2, tuy nhiên dựa vào kết qar ta vẫn có thể thấy thời gian xử lý của thuật toán AlphaBeta tốt hơn so với thuật toán Minimax ở các layer có độ khó cao hơn, nhờ vào kĩ thuật Pruning. Còn ở Expectimax, ta chưa so sánh được, vì có tỉ lệ chiến thắng khác so với hai thuật toán trên.

So sánh hàm lượng giá cũ với hàm lượng giá mới

Sau đây là kết quả tổng hợp trên hàm lượng giá cũ, em đã gộp chung giải thuật Minimax và AlphaBeta vì kết quả của hai thuật toán này là như nhau.

OldEvaluation	**Minimax - AlphaBeta Agent**			**Expectimax**		
Layout	**mean**	**std**	**game_win**	**mean**	**std**	**game_win**
trappedClassic	325.2	413.6	4	325.2	413.6	4
contestClassic	-531.0	132.2	0	-471.0	37.9	0
powerClassic	-415.2	41.9	0	-487.2	13.1	0
capsuleClassic	-491.6	109.0	0	-504.2	9.5	0
mediumClassic	-433.2	66.1	0	-575.0	146.8	0
smallClassic	-553.2	198.9	0	-478.6	28.3	0
minimaxClassic	-495.2	1.46	0	109.6	497.7	3
testClassic	0	0	0	0	0	0
trickyClassic	0	0	0	0	0	0
originalClassic	0	0	0	-574.0	71.1	0
openClassic	0	0	0	-1554.2	600.9	0

Table 3: Kết quả tổng hợp với hàm lượng giá cũ

Ta có thể quan sát qua đồ thị dưới đây để có cái nhìn trực quan hơn:

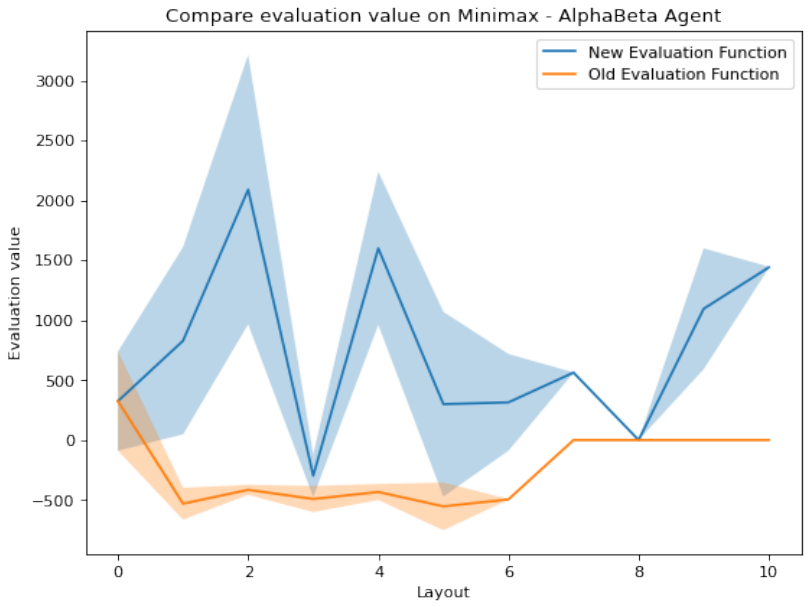

Figure 3: So sánh kết quả trên thuật toán Minimax - AlphaBeta Agent

Ta có thể thấy, hàm lượng giá mới tuy chưa đạt đến ngưỡng tối ưu nhất, nhưng đã tốt hơn rất nhiều so với hàm lượng giá cũ, có nhiều layout mà hàm lượng giác cũ vẫn chưa giải quyết được nhưng với hàm lượng giác mới đã có thể giải khá suôn sẻ. Ta vẫn có thể thấy hạn chế ở hàm lượng giác mới, với khoảng error bar khá lớn, do chưa đạt 5/5 ở các lần thử nghiệm nên kết quả phân bố rộng, dẫn đến error bar lớn.

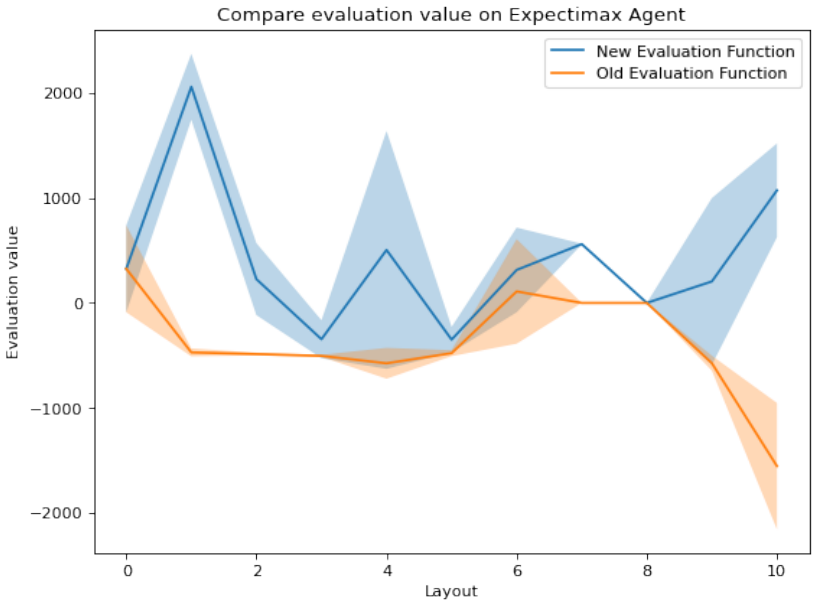

Figure 4: So sánh kết quả trên thuật toán Expectimax Agent

Kết luận

Trong các thuật toán, ta có thuật toán Minimax và AlphaBeta tuy là luôn có cùng kết quả thực nghiệm, nhưng ở AlphaBeta sẽ cho thời gian xử lí thấp hơn so với Minimax nếu các nhánh có thứ tự sắp xết tốt hơn và độ sâu lớn hơn nhờ vào bước cải tiến Pruning so với minimax. Còn ở thuật toán Expectimax, việc thiết kế