COSE361(03) Artificial Intelligence

Assignment #1

2022320033 박종혁

a. Capture the result of pacman.py with layout test71.lay.

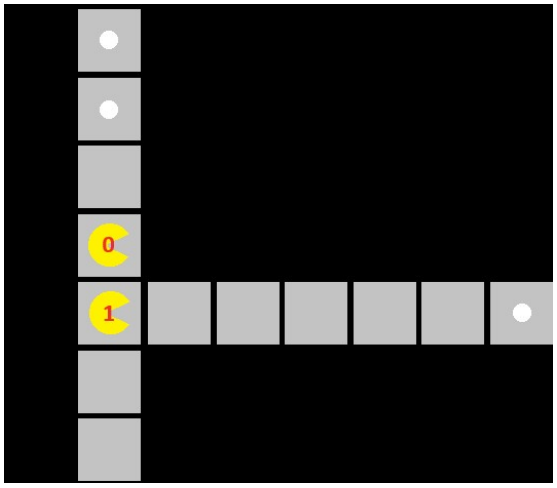
```
PS D:\Users\PC\Desktop\KoreaUniv\COSE361_Artificial_Intelligence\minicontest1> python pacman.py --agent MyAgent --layout test71.lay
Pacman emerges victorious! Score: 794
Average Score: 794.000000000002
Scores:        794.000000000002
Win Rate:      1/1 (1.00)
Record:        Win
```

b. Description of your agents.

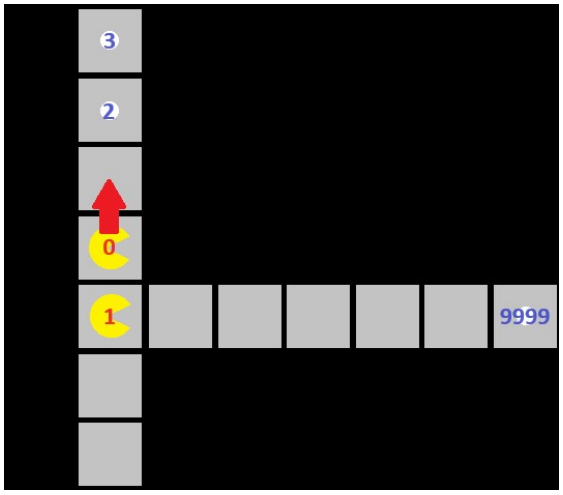
기본적으로 가장 가까이 있는 Dot을 탐색하고, 그 지점을 향해 움직인다. 그러나 ClosestDotAgent와 다른 점은 다른 팩맨 agent들의 위치 정보를 고려하여 행동한다는 것이다. ClosestDotAgent의 가장 큰 문제점은 하나의 Dot을 향해 여러 마리의 팩맨이 움직일 때 발생한다. 이 경우 가장 가까웠던 팩맨이 해당 Dot을 먹고나면 그 Dot을 향해 오던 나머지 팩맨들은 움직임을 낭비한 것이 되어버린다. 따라서 나는 모든 agent들이 공유하는 클래스 변수 Dist를 만들어, 특정 Dot으로부터 가장 가까이 있는 팩맨과의 거리를 저장하도록 하였다. 알고리즘을 대략적으로 설명하면 다음과 같다.

1. 0번 agent의 getAction이 호출되면, 맵 전체 Grid의 사이즈를 가진 2차원 배열 Dist를 선언하고 모든 element를 9999로 초기화한다. 9999는 +INF를 나타내기 위해 임의로 정한 숫자이다.
2. 현재 agent로부터 가장 가까운 Dot을 BFS를 이용하여 찾는다. 해당 Dot의 좌표를 (x, y)라고 할 때, (Dist[x][y])의 값과 (현재 agent로부터 해당 Dot까지의 거리)를 비교한다.
3. Dist 배열에 저장된 값이 더 작은 경우, 해당 Dot에 대해 더 빠르게 도달할 수 있는 agent가 이미 있다는 뜻이므로 무시하고 BFS를 계속 이어나간다.
4. Dist 배열에 저장된 값이 더 큰 경우, 해당 Dot으로 향하는 action을 취하고, Dist 배열의 값을 업데이트한다. 이때 해당 Dot으로부터 일정 범위만큼 BFS를 다시 실행하여 주변 Dot들의 Dist 값도 같은 방식으로 업데이트한다.
5. 모든 agent에 대해 순차적으로 2~4를 반복한다.

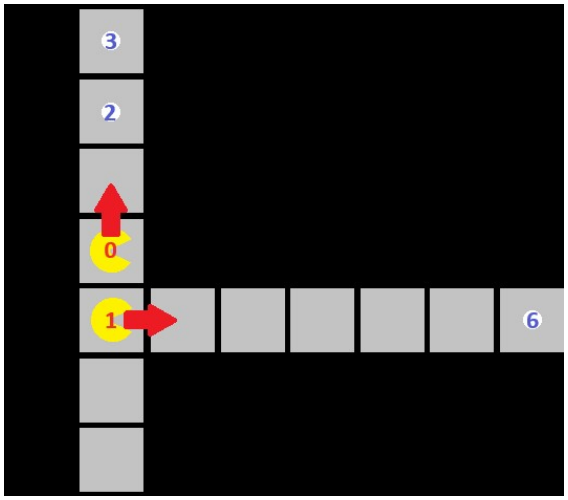
그림으로 간단한 예시를 들면 다음과 같다. 팩맨에 써 있는 숫자는 agent index를 의미한다.



위의 경우, 0번 팩맨이 getAction을 취한 후 Dist 배열의 상황과 팩맨의 움직임은 다음과 같다. 파란색 숫자는 해당 Dot의 Dist 값, 빨간색 화살표는 0번 팩맨의 이동 방향이다.



다음 1번 팩맨이 getAction을 취할 때, 위쪽 두 개의 Dot은 Dist값이 자신과의 거리보다 작으므로 무시하고, 그 다음으로 가까운 오른쪽 Dot을 향하게 된다.



c-1. Discuss cases where the agent implemented by yourself is better than the baseline.

위에서 그림으로 든 예시가 baseline(ClosestDotAgent)보다 MyAgent가 효율적인 경우이다. 위의 상황에서 ClosestDotAgent라면 두 팩맨이 모두 위쪽을 향해 이동하다가 0번 팩맨이 두 개의 Dot을 먹고 난 후에야 두 팩맨이 함께 오른쪽 Dot을 향해 움직일 것이다. 그러나 MyAgent의 경우 0번 팩맨은 위로, 1번 팩맨은 오른쪽으로 동시에 이동하므로 동선 낭비를 줄일 수 있다.

c-2. Discuss cases where the agent implemented by yourself is worse than the baseline.

위의 예시에서 두 팩맨의 인덱스가 서로 바뀌면 MyAgent는 ClosestDotAgent보다 비효율적이게 된다. 위 그림의 1번 팩맨이 먼저 행동을 취하면, 위쪽 두 개의 Dot의 Dist 값은 각각 3, 4가 되고, 팩맨은 위쪽으로 향한다. 그러고 나서 0번 팩맨이 행동을 취하면 위쪽 두 개의 Dot의 Dist 값 3, 4는 자신과의 거리 2, 3보다 크므로 Dist 값을 업데이트하고 0번 팩맨 또한 위쪽을 향한다. 즉, ClosestDotAgent와 동일한 동선을 따르는 것이다. 그런데 연산량이 더 많으니 결과적으로 MyAgent가 더 비효율적이다.

c-3. Ask & Answer your own question about the above discussion.

Q. c-2에서 언급된 문제점을 개선할 수 있는 방안은 어떤 것이 있을까?

A. 가장 쉬운 방법은 BFS 탐색 및 dist 배열 업데이트 연산을 반복적으로 수행하는 것이다. 위의 문제점이 발생하는 이유는 각 agent가 행동을 취할 때 자신보다 높은 인덱스의 agent가 가진 정보를 이용하지 못하기 때문이다. 즉, 1번 팩맨은 0번 팩맨이 업데이트 해 놓은 Dist 배열 정보를 이용하여 행동을 정할 수 있지만, 0번 팩맨은 1번 팩맨이 어느 Dot으로 향할 것인지 알 수 없는 상태로 행동을 결정한다. 이 문제를 해결하기 위해선 Dist 값의 업데이트가 반복적으로 이루어져 높은 인덱스의 agent가 가진 정보를 낮은 인덱스의 agent도 이용할 수 있도록 해야 한다. 예를 들어, agent의 인덱스가 0~N까지 존재한다 하자. 이때 현재 MyAgent의 작동 순서는 다음과 같다. '계산'은 어느 Dot을 향할지 결정하는 단계, '행동'은 실제로 움직이는 단계를 의미한다.

> 0번 계산 -> 0번 행동 -> 1번 계산 -> 1번 행동 -> ... -> N번 계산 -> N번 행동

이것이 다음과 같이 바뀌어야 한다.

> 0번 계산 -> 1번 계산 -> 2번 계산 -> ... -> N번 계산 -> 0번 계산 -> 1번 계산 -> ... -> N-1번 계산 -> 0번 계산 -> ... -> N-2번 계산 -> 0번 계산 -> ... -> 0번 행동 -> 1번 행동 -> ... N번 행동

또한, 더 효율적인 경로를 선택하는 것은 좋으나 연산 시간의 증가로 인해 baseline에 비해 극적인 성능 향상을 보이지 못한 점이 아쉽다.