Submission of the Final Project(2/2)

2017170307정승원[1]

Abstract

이 보고서는 Final Project(2/2)에 대한 보고서로,
구성은 다음과 같다.
1. Introduction
2. Methods
3. Results
4. Conclusion and Free discussion

1. Introduction

Final project(2/2)에서는 상대방과 겨루어 승자를 가린다. 게임 상에서 map은 내 진영과 상대 진영 두 부분으로 나누어져 있다. 각 팀은 두 개의 agent를 가지고 있다.

우리 팀 agent가 상대 진영으로 넘어가면 pacman으로 바뀐다 (이를 공격자라고 부르겠다). 공격자는 상대 진영에 있는 ghost를 피해 food를 먹고 내 진영으로 다시 가져와야 한다. 이때 가져온 food의 개수만큼 score는 증가한다. 만약 돌아오기 전에 ghost에게 잡히면 공격자는 자기 진영의 출발점으로 돌아오게 되고 가지고 있었던 food들은 ghost에게 먹힌 자리에 남겨진다. 상대 진영에 food 두 개를 남기고 먼저 18개를 내 진영으로 가져오거나 시간이 다 되었을 때 점수가 높으면 승리한다.

Baseline은 총 3개를 만들었다.

- **your baseline 1 :** 공격자와 수비자를 나누어 설계 Alpha-Beta Pruning을 이용한 Minimax Search를 사용하여 다음에 할 행동을 정하였다.
- **your baseline 2 :** 공격자와 수비자를 나누어 설계 Expectimax Search를 사용하여 다음에 할 행동을 정하였다.
- **your baseline 3 :** 공격자와 수비자를 따로 나누지 않고 두 가지 기능을 함께하는 agent를 설계 Alpha-Beta Pruning을 이용한 Minimax Search를 사용하여 다음에 할 행동을 정하였다.

[1]고려대학교, 건축사회환경공학부. Correspondence to: 정승원 <redlion0929@korea.ac.kr>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

2. Methods

이제부터 현재 탐색 대상인 agent를 나, 같은 팀의 다른 agent를 팀원, 상대팀의 ghost를 defender(ghost), 상대팀 pacman(공격자)을 invader라고 부를 것이다.

2.1. your baseline1

Alpha-Beta Pruning을 이용한 Minimax Search방식을 사용하였다. 공격을 담당하는 에이전트와 수비를 담당하는 에이전트를 따로 만들어 역할을 분담시켰다.

'현재 탐색 중인 agent(나) - 상대의 첫 번째 agent - 상대의 두 번째 agent' 이렇게 한 사이클을 진행할 때마다 depth는 1 증가한다. depth=2까지 탐색한 후 getFeatures에서 평가한 값을 바탕으로 각 상태에 해당하는 값을 구하고 다음 행동을 선택하였다.

2.1.1. 공격

만약 현재 남은 food가 2개 이하라면 출발점과 가장 가까운 곳으로 향하는 행동을 취해 돌아간다. 만약 남은 food가 2개보다 많다면 alpha-beta pruning을 적용한 minimax search를 사용하여 다음에 취할 행동을 선택하고 반환한다.

getFeatures에서는 12개의 features와 가중치를 이용하여 상태를 평가했다. 상대 defender의 수를 구해 features['numOfGhost'], 현재 나의 점수를 features['nextScore'], 현재 남은 음식의 개수를 features['successorScore'], 현재 남은 capsule의 개수를 features['leftCapsules']에 저장하였다.

나머지 features에 대한 정보는 아래와 같다.

1) 내가 pacman이 아닐 경우 상대 invader와의 최소 거리가 1보다 작거나 같다면 features['canCatchN']에 1을 저장해 주어 그 invader를 잡을 수 있게끔 코드를 작성했다.
2) food와의 최소 거리를 구해 features['distanceToFood']에 저장하였다.
3) ghost와의 최소 거리를 구해 features['distanceToGhost']에 저장하였다. 만약 ghost와의 최소 거리가 1이하일때는 아래와 같이 경우를 나누어 처리하였다.

1. 만약 내가 pacman이 아니고 내 진영에 있을 경우
상대 ghost가 나의 진영으로 넘어오는 순간에 ghost를 잡을 수 있다. 따라서 상대를 잡을 기회가 생기므로 features['canCatch']에 5000을 저장하였다. ghost와의 거리가 줄어들 때 더 많은 점수를 얻기 위해 fea-

Algorithm 1 Find Wall

Initialize $wallNum = 0$
$ar = [(-0.5, 0), (0.5, 0), (0, -0.5), (0, 0.5)]$.
if $gameState.getAgentState(self.index).isPacman$ **then**
  for i **in** ar **do**
    if $gameState.hasWall(pos[0]+i[0], pos[1]+i[1])$ **then**
      $wallNum = wallNum + 1$
    end if
  end for
end if

tures['distanceToGhost']에 10*(ghost와의 최소거리)를 저장해주었다.

2-1. 내가 pacman이고, 가장 가까운 ghost가 현재 scared 상태이며 scaredTime이 4보다 많이 남았을 경우
상대 ghost를 잡을 수 있는 상태이므로 features['canCatch']에 5000을 저장한다. ghost에게 잡힐 일이 없으므로 ghost와의 거리보다 food와의 거리에 더 가중치를 준다. 따라서 features['distanceToFood']에 2*(음식과의 최소거리)를 저장해주었다.

2-2. 내가 pacman이고, 가장 가까운 ghost가 현재 scared 상태이지만 scaredTimer가 4이하로 남았을 경우
ghost의 scared상태가 끝나기 전에 ghost를 잡아야하므로 features['canCatch']에 5000*(1-(ghost와의 최소 거리))를 저장해주었다. ghost와의 거리가 줄어들수록 이 값은 증가할 것이다.

3. 내가 pacman이고 ghost가 scared가 아닐 경우
이러한 상황이 되는 것을 피해야하므로 features['canCatch']에 –5000을 저장한다. 최대한 상대 ghost와의 거리가 멀어지는 것이 좋으므로 features['distanceToGhost']에 –7*(ghost와의 최소거리)를 저장하여 defender과 멀어지는 것에 중점을 두었다.

4) 만약 남은 food가 2개 이하로 남았을 경우 food를 더 먹지 않고나의 진영으로 가져가면 승리할 수 있으니, 가장 가까운 food와의 거리를 0으로 두어 food를 신경쓰지 않게 하였다.
5) capsule과의 최소 거리를 구하여 features['distanceToCapsule']에 저장하였다.
6) 'Algorithm 1 : Find Wall' 코드를 이용하여 주변에 있는 벽의 개수를 계산하였다. 만약 주변에 벽이 3개가 있다면 구석진 곳이다. 이 곳으로 갔을 경우 ghost로부터 도망갈 곳이 없기 때문에 잡힐 확률이 높아진다. 따라서 features['noWay']에 –1000을 저장해주어 상태를 낮게 평가한다. 만약 주변에 벽이 3개고 그 곳에 음식이 없다면 features['noWay']에 –1000000을 저장하여 이 곳의 점수를 아주 낮게 평가한다.

7) 현재 내가 가지고 있는 food의 개수를 havingFood라고 하자. 내 진영과 상대 진영 사이의 경계(중간 지점)와 내 위치 사이의 x좌표 차이를 minHDist라고 하자.

1. minHDist가 0이 아닐 경우// features['havingFood']에 havingFood/minHDist를 저장한다. 가지고 있는 음식의 개수가 늘어나면 좋지만, 경계선과의 거리가 멀어지면 좋지 않으므로 이를 반영하였다.

2. minHDist가 0일 경우
features['hvingFood']에 havingFood를 저장하였다.

8) 만약 havingFood가 2이상이면 features['goToHome']에 minHDist를 저장한다.

가중치 설명
1. 상대 defender를 잡을 수 있는 기회가 오면 그 기회를 좋게 생각해야 하므로 canCatchN에 가중치 100을 주었다.
2. goToHome은 food를 2개 이상 있을 때 내 진영까지의 x축방향 직선 거리에 대한 features이다. food를 많이 가지고 있을 때 내 진영까지의 거리가 멀어지는 것은 좋지 않다. 따라서 거리가 커질수록 점수가 감소하게끔 –76을 가중치로 주었다. 가중치를 너무 작게 하면 food를 2개 가질 때마다 복귀하려고 할 것이고 너무 크게 하면 food를 많이 가졌음에도 탐험을 계속 진행할 것이다.
3. food를 먹고 내 진영과의 거리가 가까워지는 것은 좋게 평가할 수 있다. 따라서 havingFood에 가중치 190을 부과하였다.
4. food와의 최소거리가 작아지는 것은 부정적인 요소보다는 긍정적인 요소가 더 많다. 따라서 거리가 가까워지면 점수가 증가하게끔 하기 위해 가중치를 음수로 설정하였다.
5. 항상은 아니지만 ghost와의 거리가 멀어지는 것이 가까워지는 것보다 이롭다. 따라서 distanceToFood의 가중치를 양수로 설정하였다. 이때 distanceToFood의 가중치의 절댓값보다 작은 값을 주어 agent가 ghost보다는 food와의 거리를 우선으로 생각하게 하였다. ghost를 피하기만 하면 이기지 못하기 때문이다.
6. ghost의 수가 작을수록 좋으므로 numOfGhost의 가중치를 음수로 설정하였다.
7. nextScore의 가중치를 8000 주었다. 점수를 올릴 수 있는 상황에 점수를 올리지 않고 욕심을 부리면 경기에서 이기지 못하리라 생각하여 큰 가중치를 주었다.
8. successorScore은 현재 남은 food의 개수에 관한 정보이다. pacman이 food를 먹으면 이 값은 줄어든다. food를 먹는 것이 긍정적인 요소가 더 많으므로 가중치를 음수로 주었다. 큰 절댓값을 가중치로 주면 agent가 food를 먹는 것에만 집중하여 ghost를 피하지 않고, 작은 절댓값을 가중치로 주면 agent가 food에 관심을 두지 않는다. 시행착오 끝에 –3200을 가중치로 설정하였다.
9. pacman이 가장 자유로운 시기는 capsule을 먹어 ghost가 scared 상태가 되었을 때이다. capsule을 먹으면 남아있는 capsule의 개수가 줄어들게 된다. 따라서 leftCapsules에 가중치를 –3000을 주었다.
10. 9번에서 설명한 내용과 유사하다. capsule과의 최소거리가 가까우면 좋으므로 가중치를 음수를 준다. 이때 distanceToFood의 가중치가 –139이므로 절댓값이 139보다 큰 음수의 가중치를 주었다. 가중치의 절댓값이

너무 크면 capsule만을 향해 가기 때문에 비효율적이고 절댓값이 너무 작으면 capsule보다 food를 먹는것에 우선을 두기 때문에 비효율적이다. 시행착오 끝에 –218.5를 가중치로 주었다.

2.1.2. 수비

수비를 담당하는 agent는 공격과 수비의 역할을 둘 다 한다. invader가 없을 때는 공격을, invader가 있을 때는 수비를 하였다. 공격 시 agent의 활동은 2.1.1에서 설명한 agent와 같다.

getFeatures에서는 대해 9개의 features와 가중치를 이용하여 현재 수비중인 agent를 평가했다. invader의 개수를 features[‘invader’], 남은 capsule의 개수를 features[‘leftCapsules’], food와의 최소거리를 features[‘food’], capsule과의 최소거리를 features[‘distancToCapsule’]에 저장해주었다.

나머지 features에 대한 정보는 아래와 같다.

1) 내가 pacman이 아니라면 수비 중인 경우이므로 features[‘onDefense’]에 1을 저장하였다.
2) invader와의 최소 거리를 구해 features[‘distanceToinvader’]에 저장하였다. 만약 invader와의 최소 거리가 1이하일때는 경우 features[‘nextScore’]에 1000을 저장해주었다. 이후 경우를 나누어 상태를 평가하였다.

1. 내가 scared 상태라면, features[‘canCaught’]에 –1000*(3-(invader과의 최소거리))를 저장해주어 invader과의 거리가 가까워질수록 점수가 크게 감소하도록 하였다.

2. 내가 scared 상태가 아니면 invader를 잡을 수 있는 상태이므로, features[‘canCaught’]에 1000*(3-(invader과의 최소거리))를 저장해주어 invader과의 거리가 가까워질수록 점수가 크게 증가하도록 하였다.

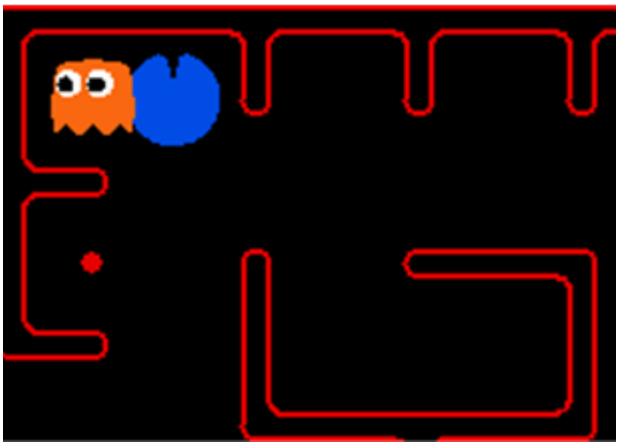
3. invodr이 없으면 상대 ghost와의 최소거리를 구해 features[‘distanceToinvader’]에 저장하였다. 이는 상대 pacman이 내 진영에 들어오기 전에 미리 거리를 줄여놓기 위함이다.

3)만약 이전 상태에서보다 현재 상태에서 상대 pacman의 숫자가 적다면, 내가 상대 pacman을 잡은 것이므로 features[‘catchP’]에 1000을 저장하여 이 상태를 높게 평가하였다.

가중치 설명
1. onDefense에 가중치 1000을 주어 agent가 pacman이 아니라 defender의 역할에 중점을 두도록 하였다.
2. 수비의 가장 중요한 역할은 상대 invader을 잡는 것이므로 canCaught에 가중치 100을 주었다.
3. invader은 food를 먹으러 올 것이므로 distanceToFood에 가중치 –150을 주었다. 음수를 준 이유는 최소 거리가 가까울수록 좋은 상태이기 때문이다. 같은 이유로 distanceToCapsule에는 –200을 주었다.
4. invader와의 최소거리는 작을수록 좋다. 따라서 distanceToinvader에 가중치 –500을 주었다. 이때 distanceToFood의 가중치보다 큰 절댓값을 사용하였는데 이는 agent가 food보다는 invader와의 거리에 더 중점을 두게 하기 위함이다.
5. 상대 pacman을 잡으면 invader의 수가 줄어들게 된다. invader의 수는 작을수록 좋기 때문에 가중치 –5000을 주었다. 위에서 설명했듯 수비의 역할은 invader을 잡는 것이기 때문에 절댓값이 큰 가중치를 주었다.
6. invader와의 거리가 가까우면 features[‘nextScore’]가 1000이된다. 따라서 nextScore에 양수의 가중치 500을 주었다.

2.2. your baseline2

상대 agent의 행동이 나의 agent가 정확히 예측한대로 되지 않을 경우를 고려하여 Expectimax Search를 사용하였다. 공격을 담당하는 에이전트와 수비를 담당하는 에이전트를 따로 만들어 역할을 분담시켰다. 공격과 수비를 담당하는 agent는 your baseline1에서 사용한 것과 같다.
탐색 과정에서 depth를 2까지로 지정하여 탐색을 진행하였고, getFeatures에서 평가한 값을 바탕으로 각 상태에 해당하는 값을 구하고 다음 행동을 선택하였다.



My team : red, Opponent :blue

Expectimax search를 사용했을 경우, 위의 사진처럼 invader를 잡아야만 하는 상황에서 내 agent와 상대 agent는 정지한다(baseline과 상대했을 경우). 저 상황에서 아무런 action도 취하지 않은 채 시간이 모두 흘러 게임은 끝이 난다. 저런 상황이 나왔을 때 탐색을 하지 않고 invader과 가장 가까운 방향으로 나의 agent가 이동하도록 아래의 ’Algorithm 2 : Action In Expectimax’ 코드를 넣어주었다. 이때 위와 같은 상황은 더 이상 발생하지 않았고 수비 상황에서 invader를 잡는 횟수가 더 늘어났다.

Algorithm 2 Action In Expectimax

Initialize $bestDist = infinity$.
if $minIDist == 1 and myState.scaredTimer <= 0$ **then**
  for $action$ **in** $gameState.getLegalActions(self.index)$ **do**
    $dist = distance$ **pos** $with$ **invaders[minIndex])**
    if $dist < bestDist$ **then**
      $nextAction = action$
      $bestDist = dist$
    end if
  end for
end if
$return$ **nextAction**

2.3. your baseline3

your baseline1과 같이 Alpha-Beta Pruning을 이용한 Minimax Search를 사용하였다. 하지만 agent는 공격과 수비를 모두 담당하는 all rounder agent를 사용하였다. 탐색 과정에서 depth를 2까지로 지정하여 탐색을 진행하였고, getFeatures에서 평가한 값을 바탕으로 각 상태에 해당하는 값을 구하고 다음 행동을 선택하였다.

all rounder agent는 4가지 경우에 따라 다른 행동을 취한다.

1) invader가 없거나, 내 진영에 남아있는 food가 14개보다 클 경우

이 agent는 공격에 참여한다. 전체적으로 보면 팀원 모두가 공격에 참여한다. your baseline1에 사용한 attack agent의 코드에 features 하나를 더 추가하였다. 팀원 간의 거리를 고려하지 않으면 두 agent가 같은 경로로 함께 움직일 것이다. 이는 매우 비효율적이기 때문에 팀원과의 거리를 구해 features['otherIndex']에 저장하고 가중치를 20을 주어 팀원과 멀어질수록 높은 점수를 받게 하였다.

2) invader가 1개이고 내 진영에 남아있는 food가 14개보다 작거나 같을 경우

나의 index가 우리 팀의 첫 번째 index와 같다면 수비에 참여하고 그렇지 않으면 공격을 계속한다. 전체적으로 보면 팀원 중 하나는 공격, 하나는 수비에 참여하게 된다. 공격과 수비 agent의 경우 your baseline1의 코드를 사용하였다.

3) 위의 두 경우가 아닐 때 (invader이 2개인 경우)

이때 이 agent는 수비에 참여한다. 전체적으로 보면 팀원 모두가 수비에 참여한다. your baseline1의 수비를 담당하는 코드에 features 하나를 더 추가하였다. 팀원 간의 거리를 고려하여 하나의 invader를 잡으러 두 agent가 함께 다니는 것보다는 흩어지는 것이 효율적이므로 같은 팀원과의 거리를 구해 features['otherIndex']에 저장하고 가중치를 20 주었다.

3. Results

	A	B	C	D
1		your_best(red)		
2		<Average Winning Rate>		
3	your_base1	1		
4	your_base2	0.8		
5	your_base3	-0.5		
6	baseline	1		
7	Num_Win	3		
8	Avg_Winni	0.575		
9		<Average Scores>		
10	your_base1	4.4		
11	your_base2	3		
12	your_base3	0		
13	baesline	11.4		
14	Avg_Score	4.7		

4. Conclusion and Free discussion

4.1. Conclusion

3개의 baseline중 가장 승률이 좋은 your baseline3를 best로 설정하였다.
all rounder agent는 상황에 맞게 행동을 선택하기 때문에 다양한 state에 대해 유연하게 대처할 수 있다. 경쟁에서 이겨야 하는 상황에서 이 agent는 좋은 성능을 보여 줄것이다. 따라서 your baseline3를 best로 정하였다.

4.2. Free discussion

Q) 탐색 알고리즘 중 Minimax Search와 Expectimax Search 알고리즘을 선택한 이유는 무엇인가?

A) Q-learning을 이용하면 agent가 몇 번의 시행착오 이후 optimal하게 움직일 것이다. 하지만 정해진 시간 동안 점수를 많이 얻어야 하는 상황 속에서 learning을 하는 것은 비효율적이라고 생각했다. 따라서 learning보다 planning을 선택했다.
상대의 행동을 예측하지 않고 그저 food만을 먹기 위해 BFS나 DFS를 사용하는 것은 비효율적이라고 생각했다. food를 아무리 optimal하게 많이 먹어도 ghost와의 거리를 신경 쓰지 않는다면 결국 ghost에게 잡힐 것이고 점수는 오르지 않을 것이다. 따라서 다른 방법을 고안해보았다. Multi agent를 다루는 대표적인 탐색 알고리즘은 Minimax Search, Expectimax Search이다. 적당한 Evaluation Function을 사용한다면 좋은 결과를 얻을 수있다고 생각했다. 따라서 Minimax Search를 이용하여 Baseline 두 개, Expectimax Search를 이용하여 Baseline 한 개를 만들었다.

Q) 하나의 agent가 공격을, 나머지 하나의 agent가 수비를 담당하는 방식은 어떠한 상황에서 장점을 가지는가?

A) invader가 없다면 두 agent 모두 공격을 할 것이고, invader이 있다면 하나의 agent는 공격을 하고 다른 agent는 수비를 할 것이다. 이 방식은 invader가 있을때 내 진영에 항상 하나의 수비수는 존재하게 된다는 점에서 안정적이다. 상대 팀 모두가 내 진영에 들어올 때 상대 진영에는 defender가 없게 된다. 이 때 수비를 담당하는 agent가 어느 정도 상대 invader들을 견제하는 동안 공격을 담당하는 agent가 food를 모두 먹어 내 진영으로 가져오면 승리할 수 있다.

Q) all rounder agent는 어떠한 상황에서 장점을 가지는가?

A)두 agent가 역할을 따로 분담하지 않고 행동한다. 굳이 나누자면 첫 번째 index를 가진 agent는 좀 더 수비적이고 두 번째 index를 가진 agent는 더 공격적이다. 상황에 맞게 수비와 공격을 선택한다. all rounder agent는 아래의 상황들에서 우수한 성능을 보인다.
상대 팀 모두가 내 진영에 들어왔을 때 두 agent는 흩어져 invader들을 잡는다. all rounder agent는 빠른 공수전환이 가능하기 때문에 invader들을 모두 잡자마자 두 agent는 바로 공격을 하러 간다. 이때 상대 진영의 defender는 출발점에서 다시 시작하므로 food를 먹을 시간이 충분하여 많은 food를 먹을 수 있다.
추가로 상대 agent 중 하나가 공격, 하나가 수비를 담당하면 food가 많이 남아있을 때 나의 두 agent는 모두 공격에 참여한다. agent 간의 거리를 어느 정도 고려하였으니 둘 다 잡히지만 않는다면 점수를 획득할 수 있다.

Q) Expectimax Search를 사용한 baseline의 장점과 단점은 무엇인가?

A)Expectimax Search는 상대방 agent가 다음에 어떠한 행동을 취할지 확신할 수 없는 경우 사용한다. 게임에서 상대방이 어떠한 알고리즘을 선택할 지 모르기 때문에 상대방의 행동을 예측하여 움직이는 Minimax Search보다는 기댓값을 바탕으로 움직이는 것이 좋은 선택이라고 생각하여 Expectimax Search를 사용하였다. 하지만 Pruning을 할 수 없어 다음 행동을 계산하는 속도가 Alpha-Beta Pruning보다 느리다는 단점이 있었다.