2018320207 공인호 COSE361-02 Midterm

0. Before

보고서를 모두 작성한 후, result 파일을 스크린샷하기 위해 명령어를 작성하면서, 본 과제가 기본 맵에 대해서만 문제를 해결하면 된다는 사실을 깨달았습니다. 저는 RANDOM한 맵(-l RANDOM)을 기준으로 Agent들을 제작하였고, 이를 분석하여 보고서를 작성하였습니다. 참고해주시면 감사하겠습니다.

단, (1)의 스크린샷은 기본 맵으로 실행시켰습니다.

1. Screenshot

a. 명령어

```
C:₩Users₩Inho₩Desktop₩AI₩minicontest2>python capture.py -r 2018320207 -b baseline -n 10 -Q
```

b. CSV Screenshot

	your_best(red)	
	<Average Winning Rate>	
your_base	0.4	
your_base	0	
your_base	-0.4	
baseline	0.8	
Num_Win	2	
Avg_Winni	0.2	
	<Average Scores>	
your_base	-0.4	
your_base	0	
your_base	-1	
baesline	6.1	
Avg_Score	1.175	

c. Each Result

baseline1

baseline2

baseline3

baseline

2. SimpleBFSAttackAgent (your_baseline1)

a. Abstract

첫 번째로 소개할 Agent는 your_baseline1으로 채택된 SimpleBFSAttackAgent이다.
your_baseline1.py에서 Agent1과 Agent2 모두 SimpleBFSAttackAgent를 사용하였다.

SimpleBFSAttackAgent는 ReflexAgent로, 앞뒤 상황을 고려하지 않고 현재 경기 상황만을 가지고 다음 Action을 결정하며, "상대방 진영에서 죽지 말자"는 아이디어를 바탕으로 동작한다. Food를 먹기 위해 상대방 진영에 들어가기 전, 상대방의 가능한 모든 움직임을 예측하고, Food를 먹고 자신의 진영으로 반드시 돌아올 수 있을때에만 공격을 허용했다. 그 외의 경우에는 매우 간단한 greedy 알고리즘으로 아군 Food를 수비하도록 Agent를 구현하였다.

SimpleBFSAttackAgent의 자세한 작동 방식과, 각 메소드에 대한 설명은 아래와 같다. 단, 설명의 편의를 위하여 코드에서 등장하는 순서와 다르게 배치하였다

b. Agent Analysis

updateSafeArea

> 모든 (x,y)에 대해서, 내 Agent가 상대 Agent들보다 얼마나 빨리 해당 위치에 도달 가능한지 계산하여 *self.SafeArea*를 업데이트 해주는 메소드이다. 상대의 두 Agent들 중에서 먼저 도착하는 Agent를 기준으로 하였으며, 만약 상대 Agent가 scaredTime이 진행중이라면, 해당 Agent는 무한히 늦게 도착한다고 계산하였다.
>
> 이 행렬을 통해 여러가지 사실을 알 수 있다. 특히, 어떤 위치에서 행렬 값이 음수라는 것은, 그 위치에 아군 Agent가 상대방 Agent보다 먼저 도착할 수 있다는 것이고, 즉, 상대방 Agent를 마주치지 않고 이동할 수 있는 경로가 적어도 하나 존재한다는 의미를 가지고 있다. 이를 이용하여, 상대방 진영에서 아군 진영으로 돌아올 때, 어느 위치로 돌아와야 할지 쉽게 계산이 가능하다.

canEscape

> SimpleBFSAttackAgent의 정체성을 담고 있는 메소드이다. 목표 위치 pos(x,y)를 입력으로 받아, 해당 위치까지 안전하게 이동할 수 있는지, 또 해당 위치에서 아군 진영으로 안전하게 복귀할 수 있는지 알려준다. BFS를 이용해 상대방의 가능한 모든 움직임을 계산하며, 자세한 작동 방식은 아래와 같다.
>
> 0. 만약 상대방 Agent 중 하나가 현재 scaredTime 진행중이고, 남은 시간이 "목표 위치까지 거리 + 자기 진영까지 돌아가는 거리" 보다 긴 경우 해당 Agent는 BFS에서 제외한다.
>
> 1. 현재 위치부터 목표 위치(pos)까지의 거리를 계산하고, 그 거리만큼 상대방 Agent의 bfs를 진행한다.
> 2. 상대방과 아군 Agent를 번갈아가면서 BFS를 진행한다. 아군 Agent는 Pos에서 시작하며, 상대 Agent는 1에서 진행한 BFS상태를 그대로 이어받는다. 이때, 아군 Agent는 아군 진영에서 자유롭고, 상대 진영에서는 상대 Agent가 방문하지 않은 곳만 방문 가능하다. 또한 상대방 진영의 한 위치를 방문하려 할 때, 바로 전 시점에 상대 Agent가 같은 위치를 방문하려고 했다면, 아군 Agent의 방문을 취소한다. 상대방 Agent도 마찬가지로 작동한다.
> 3. 만약 아군 Agent가 아군 진영 아무 곳이나 방문을 한다면 True를 리턴하고, 방문 없이 queue가 모두 비어버리게 된다면 False를 리턴한다.

gotoPosSafe

> canEscape와 비슷한 알고리즘으로 작동하며, 경로를 추적하는 기능이 추가되어있다. 현재 아군 Agent의 위치에서 목표 지점까지 안전하게 갈 수 있는 경로를 찾고, 이를 위한 첫 번째 Action을 리턴해준다. 만약, 그런 경로가 없다면, gotoPos의 리턴값을 반환한다.
>
> canEscape 메소드 작동 방식 2번에서, 진영에 따라 각 Agent의 bfs에 제약을 준 것에 의해, 아군 진영, 상대 진영에 상관 없이 해당 메소드를 사용 가능하며, 아군 진영과 상대 진영을 넘나들며 이동하는 것도 자유롭다는 특징을 가진다.

gotoPos

> 입력받은 위치로 최단 경로로 이동한다. 현재 진영, 상대방의 Agent 등을 고려하지 않는다.

chooseAction

> 경기 상황을 체크하여 알맞는 명령을 수행한다. 작동 방식은 아래와 같다.
>
> 1. 만약 시간이 끝나간다면, 집으로 돌아온다.
> 2. Food들 중에서 안전하게 먹고 돌아올 수 있는 Food가 있는지 확인하고, 존재한다면 그 Food로 이동한다.

> 3. 그런 Food가 없다면 안전한 지역들 중에서 가장 가장 깊숙한 곳으로 이동한다.
> 4. 만약 이동할 안전한 지역조차 없다면, 상대방 영역이라면 comeBackHome, 아군 영역이라면 justGuard를 수행한다.
>
> 여기서 먹고 돌아올 수 있는지 또는 안전한 지역인지를 *self.SafeArea*와 *canEscape*를 이용하여 확인한다.

justGuard

> 우리 진영과 상대 진영을 오고갈 수 있는 width/2 부근만 방어를 잘 한다면, 상대방의 득점을 막을 수 있다고 생각하였다. 이에 해당 세로 라인 중에서 SafeArea 값이 가장 큰, 즉, 내 Agent보다 상대 Agent가 더 먼저 도착할 가능성이 있는 위치로 내 Agent를 이동 시킴으로써 방어 알고리즘을 설계하였다.

comeBackHome

> 우리 진영과 상대 진영의 경계에 대해서 SafeArea 값이 음수인, 즉 내 Agent가 먼저 도착 가능한 위치들 중에서 가장 가까운 위치로 이동(gotoPosSafe)시켰다. 만약 없는 경우, SafeArea값을 고려하지 않고 가장 가까운 위치로 이동시켰다.

timeBackHome

> *comeBackHome*과 동일하지만, 명령이 아닌 필요한 거리를 반환한다. 이를 이용하여 복귀에 걸리는 시간을 측정하여 게임 시간이 끝나기 전에 복귀를 완료한다.

c. Review

SimpleBFSAttackAgent는 간단해 보이지만, 꽤나 성능이 좋은 Agent이다. baseline과 100번의 게임을 시켜 보았을 때, 61승 38패 1무의 성적을 보여준다.

SimpleBFSAttackAgent는 상대방 진영에서 절대 죽지 않고 살아남아 점수를 쌓아간다는 강점을 가지고 있지만, 방어에 매우 취약하다는 치명적인 단점을 가지고 있다. baseline과의 게임에서 지는 경우를 분석해 보았을 때, 상대 agent가 capsure을 먹고 아군을 죽이는 경우, 또는 아군이 소극적으로 몇개의 Food를 가져오는 동안 상대 agent가 대부분의 Food를 먹고 돌아오는 경우가 대부분이었다. 또한, 두 Agent가 SimpleBFSAttackAgent로 동일하다보니, 두 Agent가 똑같이 붙어서 공격/수비하는 경향이 있었다.

3. StrategyAgent (your_best)

a. Abstract

두 번째로 소개할 Agent는 yout_best로 채택된 StrategyAgent이다. 2018320207.py에서 Agent1과 Agent2 모두 StrategyAgent를 사용하였다.

SimpleBFSAttackAgent에 존재하는 두 가지의 문제점을 개선하여 만든 Agent이다. 첫 번째는 쉽게 뚫리는 방어, 두 번째는 역할 분담이 이루어지지 않은 두 Agent이다. 이 두 가지 문제점을 해결하기 위해 Strategy라는 class를 생성하여 각 Agent는 'attack' 또는 'defence' 두 가지 상태 중 하나를 가지도록 하였고, 두 Agent가 서로 어떤 상태인지 공유할 수 있도록 class에 저장하였다. Agent의 상태에 따라 Agent가 취할 행동이 결정되며, 상황에 따라 'defence' 상태의 Agent가 'attack' 상태로 전환되거나 그 반대도 성립하도록 하였다. 단, 완벽한 방어를 위하여 두 Agent 중 적어도 한 Agent는 반드시 'defence' 상태를 유지하도록 알고리즘을 설계하였다.

게임에 들어가기 앞서 수비를 위한 전략적 요충지를 *self.savingPoint*에 저장하였다. 이는 우리 진영과 상대 진영의 경계에 해당되는 위치(그 중 아군 진영)와 아군 진영에 있는 capsure 위치이다. 만약 이 위치들에 대해 'defence' 상태의 아군 Agent가 상대 Agent들 보다 '항상' 먼저 도착할 수 있다면, 상대는 '절대로' 점수를 획득하지 못한다는 것을 쉽게 증명할 수 있다. 이에 'defence' 상태의 Agent는 savingPoint를 지키도록 하였고, 'attack' 상태의 Agent는 SimpleBFSAttackAgent가 작동하는 방식으로 공격하도록 하였으며, 적절한 조건을 주어 두 상태가 적절히 switch 되도록 하였다.

StrategyAgent의 자세한 작동 방식과 각 메소드에 대한 설명은 아래와 같다. 단, SimpleBFSAttackAgent와 중복되는 메소드에 대해서는 설명하지 않았다.

b. Agent Analysis

canKill

> Greedy적으로 생각해 보았을때, 가능한 모든 Action들 중에서 가장 우선순위가 높은 것은 상대방의 Agent를 죽이는 것이라고 생각했다(그 뒤로는, 아군의 savingPoint를 지키는 것, 상대방의 capsure를 먹는 것, 상대방의 Food를 먹는것이라고 생각한다). 이에 *canKill* 메소드는 한 번의 Action을 통해 상대 Agent를 죽일 수 있는지 확인하고, 죽일 수 있다면 해당 Action을, 없다면 None을 리턴해준다. 상대 Agent의 scaredTimer 또한 고려한다.

guardPoints

> 현재 gameState와 이전 gameState를 가져와, 상대의 두 Agent가 savingPoint들 중에서 어느 곳에 가까워지고 있는지 알려주는 메소드이다.

getDanger

> savingPoint들 중에서 위험하다고 판단되는 곳을 알려주는 메소드이다. 자기 자신과, 아군 진영에 있는 상대 Agent들에 대해 guardPoints까지 거리 차이가 2 이하로 나는 곳, guardPoints가 아닌 곳 중에서 거리 차이가 1 이하로 나는 곳, 두 아군 캡슐 중 거리 차이가 2 이하로 나는 곳을 중복 없이 수집하여 리턴한다.

letsGuard

> Agent가 'defence' 상태일 때, chooseAction이 호출하는 메소드이다. 작동 방식은 아래와 같다.
>
> 1. getDanger를 통해 위험한 위치를 수집한다.
> 2. 만약 위험한 위치가 없다면 (= 혼자서 모든 savingPoint를 지킬 수 있다면)
>   a. 자기 팀의 다른 Agent를 'attack' 상태로 바꾸어준다.
>   b. 만약 상대 Agent가 아군 진영 내부에 존재한다면, 해당 Agent에게 다가가는 Action을 리턴한다.
>   c. 그렇지 않다면, savingPoint들 중에서 두 상대 Agent와 가장 가까운 곳으로 이동하는 Action을 리턴한다.
> 3. 만약 위험한 위치가 있다면 자기 팀의 다른 Agent를 'defence' 상태로 바꾸어준다.
> 4. 적절한 Action을 취했을 때, 위험한 위치 모두에게 다가갈 수 있다면, 해당 Action을 리턴한다.

5. 자기 팀의 다른 Agent가 'defence'를 할 때 위험한 위치들을 getDanger를 이용하여 구하고, 1에서 구한 위치들과 교집합하여, 공통으로 해결해야하는 위치의 목록을 수집한다.
6. 이 목록에 대해 2-b,c과정을 반복한다.
7. 만약 위험한 위치에 capsure이 포함된다면, 상대방과 가장 가까운 capsure로 이동하는 Action을 리턴한다.
8. 가능한 모든 Action에 대해서 해당 Action을 취했을 때 가까워지는 위험한 위치 개수를 구하고 그 값이 가장 커지는 Action을 리턴한다.

letsAttack

> SimpleBFSAttackAgent와 똑같이 작동한다. 단, SimpleBFSAttackAgent에서는 공격할 위치가 존재하지 않을때 방어를 명령하였지만, StrategyAgent는 이를 Strategy를 이용해 관리하기 때문에 항상 *comeBackHome* 메소드를 호출한다.

chooseAction

> Agent의 상태에 따라 적절한 Action을 리턴해준다. 작동 방식은 아래와 같다.
>
> 1. SafeArea를 업데이트한다.
> 2. canKill 메소드로 상대 Agent를 죽일 수 있는지 확인한다. 죽일 수 있다면 죽인다.
> 3. 게임이 끝날때까지 시간이 얼마 남지 않았는데 상대 진영에 있다면, 집으로 돌아온다.
> 4. Agent의 상태(agentMode)에 따라 letsGuard 또는 letsAttack을 호출한다.

gotoPosKill

> *letsGuard*에서 2-b에 사용되는, 상대 Agent를 죽이러 가기 위한 Action을 리턴해주는 메소드이다.

gotoPosList

> (x, y) 위치로 가려고 할 때, 어떤 Action을 취해야 갈 수 있는지, 가능한 모든 Action을 List로 만들어 리턴한다.

etc

> 그 외의 다른 메소드는 SimpleBFSAttackAgent와 동일하다.

c. Review

StrategyAgent는 정말 많은 시행착오와 evaluation을 통해 만들어진 Agent다. 특히 *letsGuard* 메소드를 개선하기 위해, 정말 다양한 Feature를 도입해 조건을 만들어보고 이를 수정하는 작업을 거쳤다. baseline과 100번의 게임을 진행시켰을 때, 93승 2패 5무의 성적을 보여준다.

제작한 4 개의 Agent 중에서 가장 성능이 뛰어나다고 생각하여 your_best로 선정하였다. 그런 Agent를 두 번째로 소개한 이유는, 실제로 이 Agent를 두 번째로 제작하였기 때문이다. 단, 나머지 Agent를 제작하고 나서도, 새롭게 만든 Agent와 StrategyAgent를 계속 대결시키면서 StrategyAgent를 개선해시켰다.

StrategyAgent는 적절한 공격과 수비가 어울러지는 것이 특징이다. 모든 "savingPoint를 지키기만 하면 상대방이 절대 득점할 수 없다"는 greedy를 바탕으로 강력한 수비가 가능하며, guardPoints와 getDanger 메소드를 통해 지켜야 하는 위치들을 줄여줌으로써 이를 최적화시켰다.

baseline과의 게임에서 지는 케이스를 찾아본 결과, 게임이 시작하는 순간 아군 진영의 Food 중에 아군보다 상대 Agent에 더 가까우면서 상대 Agent가 해당 Food를 먹고 안전하게 돌아갈 수 있는 Food가 존재하고, StrategyAgent는 해당 Food를 수비하느라 상대편의 Food를 못 먹었지만 상대방은 먹은 상태에서, 게임이 끝나버리는 경우가 대부분이었다. 즉, 맵 자체가 완벽한 수비가 불가능한 상황에서 게임을 질 수 있다는 것이다. 하지만 이런 맵이 등장하는 경우가 그리 많지 않을 것이라 생각하였고, 오히려 이 상

황을 위한 Feature를 도입했다가 다른 맵에서 경기 결과가 안 좋게 작용할 수 있다고 생각하여 Agent를 수정하지 않았다.

4. IntelligentAgent(your_baseline2)

a. Abstract

세 번째로 소개할 Agent는 yout_baseline2로 채택된 IntelligentAgent이다. your_baseline2.py에서 Agent1과 Agent2 모두 IntelligentAgent를 사용하였다.

StrategyAgent와 baseline의 경기를 지켜보면서, StrategyAgent의 방어 체계에 허점이 존재함을 알 수 있었다. 예를 들어, Capsure이 2개가 있을때 두 번째 Capsure로 가는 척 하면서 첫 번째 Capsure을 방향을 전환하는 경우에, 이를 방어할 수 없었다. baseline이 가능한 모든 Food를 먹고 돌아가려는 욕심 때문에, Capsure에 큰 가중치를 두지 않았기 때문에 StrategyAgent가 baseline을 상대로 높은 승률을 보였던 것이지, 만약 조금 더 똑똑한 Agent였다면 방어 체계의 허술함이 쉽게 뚫리지 않을까 생각하였다.

이에 StrategyAgent에서 계산하고 있는, 아군과 상대 Agent의 각 savingPoint들까지의 거리 차이를 적절한 evaluation function으로 만들어 MiniMax algorithm과 Alpha-Beta pruning을 사용하여 방어하는 Action을 결정하도록 만들어 보았다.

IntelligentAgent의 자세한 작동 방식과 각 메소드에 대한 설명은 아래와 같다. 단, 위 Agent들과 중복되는 메소드에 대해서는 설명하지 않았다.

b. Agent Analysis

evaluate1

> 아군 Agent들 중 자신만 'defence' 상태일 때 실행되는 evaluate 함수이다. 모든 savingPoint들에 대해 "상대 Agent가 해당 위치까지 가는데 걸리는 시간 - self가 해당 위치까지 가는데 걸리는 시간"의 최소값을 계산하였다. 단 Capsure 같은 경우, 상대 Agent보다 한 발 빠르게 도착해야 그 의미가 있으므로 -1을 더해주었다.
>
> 위에서 계산한 값이 같아질 때를 대비하여, 모든 "savingPoint들에 대해 자신과 상대 Agent들의 distance 차이의 합"을 넣어주었다(위 최소값 식에서 min을 sum으로 바꾸면 된다). 또, 뒤에 나올 evaluate2를 위하여 같은 값을 중복해서 총 2개 넣어주었다. 즉, 차원이 3인 tuple을 리턴한다.

evaluate2

> 아군 Agent 모두 'defence' 상태일 때의 evaluate 함수이다. 모든 savingPoint들에 대해 "상대 Agent가 해당 위치까지 가는데 걸리는 시간 - (두 Agent의 해당 위치까지 가는데 걸리는 시간 중 짧은 값)"의 최소값을 계산하였다. 마찬가지로 Capsure의 경우, -1을 더해주었다.
>
> evaluate1과 마찬가지로, 위에서 계산한 값을 tuple의 1번째 값으로, "모든 savingPoint들에 대해 아군 Team과 상대 Agent들의 distance 차이의 합(위 min 식에서 min을 sum으로 바꾸어주면 된다)"을 tuple의 2번째 값으로, "모든 savingPoint들에 대해 자신과 상대 Agent들의 distance 차이의 합"을 tuple의 3번째 값으로 넣어주었다.

maxPlayer, minPlayer

> MiniMax Algorithm과 Alpha-Beta pruning을 구현하였다. 게임 순서는 0부터 시작한다면, [0, 1, 2, 3, 0, 1, ...]과 같이 하였다. 만약 아군 Agent 중에서 'defence' 모드인 Agent가 존재한다면, 그 순서를 minPlayer에게 넘겼다. maxPlayer는 찾아낸 evaluate 값과 그에 맞는 Action을 리턴한다.
>
> depth는 매번 한 Agent가 한 Action을 선택할 때마다 1씩 줄어들게 하였고 0이 되면, 아군 Agent들의 mode에 따라 적절한 evaluate 함수를 호출하였다.

letsGuard

> 위에서 구현한 메소드를 바탕으로 적절한 Action을 리턴한다. 아래와 같은 방식으로 작동한다.
>
> 1. 자신의 팀원 모드를 'attack'으로 바꾼다.
> 2. MiniMax 알고리즘을 수행시켜본다.
> 3. 만약 evaluate 값이 양수라면, evaluate 값과 함께 반환된 Action을 리턴한다.
> 4. 아니라면, 자신의 팀원 모드를 'defence'로 바꾼다.
> 5. 다시 한 번 MiniMax 알고리즘을 수행하고, 반환된 Action을 리턴한다.

c. Review

IntelligentAgent는 정말 완벽한 수비를 할 것이라고 생각하였지만, 시간 제한 문제가 발생하였다. MiniMax에서 depth를 6, 8로 설정했을때, 1초안에 다음 Action이 계산되지 않는 경우가 있었다(여기서 depth는 한 Agent가 한 Action을 취할때마다 1씩 줄어든다. 즉, 아군과 상대 Agent들이 모두 한 번씩 Action을 취하면 depth는 4만큼 줄어든다). 이를 개선하기 위해, gameState의 successor 함수가 느린 것이 원인이라고 판단하여, agent들의 위치만 담고 있는 새로운 class를 만들어 자체적으로 successor를 만들어보기도 하였으나 속도가 크게 개선되지 않아 결국 depth를 4로 설정하였다. 이에 baseline과 100번의 게임에서 89승 1패 10무의 성적을 거두었다.

depth가 4인 IntelligentAgent는 성능이 뛰어나진 않았다. baseline 상대로는 좋은 성적을 보여주지만, 타 Agent와 게임을 진행해보았을때 성적이 좋지 못했다. 자세한 내용은 아래 Analysis에서 설명하겠다.

만약 depth를 8 또는 12로 두었더라도, IntelligentAgent가 StrategyAgent보다 더 좋은 성능을 낼 것이라고 확신할 수 없다. StrategyAgent는 상대방의 다음 '행동'이 아닌, '목적지'를 예측하고 방어를 하는 모델이기 때문이다. 이를 확인하기 위해선 depth를 8로 두고 100번의 게임을 진행하면 되나, 한 게임에 걸리는 시간이 상상 이상으로 오래 걸려 결국 하지 못했다. 만약 depth를 충분히 크게 준다면, 그 땐 StrategyAgent보다 항상 우월할 것이다.

5. DefenceAgent (your_baseline3)

a. Abstract

마지막으로 소개할 Agent는 yout_baseline3로 채택된 DefenceAgent이다. your_baseline3.py에서 Agent1과 Agent2 모두 DefenceAgent를 사용하였다.

이전에 만든 Agent에다가 간단한 greedy를 첨가해 만들었다. 게임 룰에 따르면, 아군 Agent가 1개의 Food만 먹고 돌아오더라도, 방어를 완벽하게 하여 더이상 실점이 없다면, 해당 게임은 승리하게 된다. 이를 이용하여, StrategyAgent와 IntelligentAgent를, 게임을 이기고 있는 상황이면 오로지 방어만 하도록 코드를 수정하였다. 이에 총 2개의 새로운 DefenceAgent를 얻었으며, 그 중 StrategyAgent를 수정하여 제작한 Agent를 제출하였다.

기존 Agent를 DefenceAgent로 만드는 방법은 아래와 같다. 그 밖의 메소드는 Agent에 있는 그대로 사용한다.

b. Agent Analysis

chooseAction

```
if gameState.data.score * (1 if self.isRed else -1) > 0:
    gameStrategy.agentMode[self.index] == 'defence'
```

*chooseAction*의 mode별 메소드 선택을 하기 전, 위 조건문을 이용하여 자신의 Team이 이기고 있는지 확인하여 이기고 있다면 Agent가 'defence'를 하도록 만들어준다. StrategyAgent와 IntelligentAgent 모두 똑같이 적용 가능하다.

c. Review

각 Agent를 DefenceAgent로 만들었을때, baseline을 상대로 한 경기 결과는 다음과 같다.

Game Result (100 games)

	StrategyAgent	IntelligentAgent
Before	93승 2패 5무	89승 1패 10무
After	90승 0패 10무11.53	89승 1패 10무

예상과는 다르게, 바꾸기 이전의 승률과 큰 차이가 없었다. '승리를 조금 덜하는 대신, 패배하는 경기가 없다'라고 단정지을수도 없다. 위 StrategyAgent의 Review에서 서술했듯이, Map에 baseline과의 상성이 크게 좌우되기 때문이다. 대신 평균 득점은 11.54에서 14.77으로 증가하였다.

이러한 현상이 발생한 이유를 알아보기 위해 분석한 결과, 해당 StrategyAgent, IntelligentAgent는 baseline과의 경기에서 주로 이러한 순서로 경기가 흘러감을 알 수 있었다.

1. 게임이 시작되면 baseline Agent가 아군 진영으로 먼저 넘어온다.
2. 아군 Agent들이 baseline Agent를 견제한다.
3. 아군 Agent가 baseline을 잡아서 죽이게 되면
   a. 상대 Agent는 부활 위치에서 다시 움직이기 시작한다.
   b. 아군 Agent는 이 틈을 타서 상대 진영으로 공격을 한다.
   c. 점수를 획득한다.
4. 3에 실패하고 상대 Agent가 Capsure을 먹으면
   a. 상대 진영에 도달하여 점수를 잃는다
   b. 또는 나머지 Food를 먹다가 부활한 아군 Agent에게 죽는다. -> (3.)
5. 4명의 Agent가 모두 width/2 부근에 머물게 되며, 그 이후로는 진영의 경계에서 서로 공격하지 않는다(같은 위치만 반복).

즉, 두 팀중 한 팀이 득점을 하게 되면 매우 높은 확률로 5번 상태가 되어버려 더 이상의 득점이 일어나지 않는 것이다. 이러한 이유 때문에 DefenceAgent가 기존의 Agent와 성능이 비슷하게 나온다고 판단하였다. 게임의 맵을 모두 고정시킨 상태에서 두 Agent를 비교하면 더 좋은 분석 결과를 얻을 수 있었을 것 같다.

6. Result

a. Game Result (100 games)

baseline

	SimpleBFSAttackAgent	StrategyAgent	IntelligentAgent	DefenceAgent
# baseline	38승 61패 1무	2승 93패 5무	1승 89패 10무	90승 0패 10무

SimpleBFSAttackAgent

	baseline	StrategyAgent	IntelligentAgent	DefenceAgent
# SimpleBFSAttackAgent	61승 38패 1무	38승 31패 31무	27승 10패 63무	39승 29패 32무

StrategyAgent

	baseline	SimpleBFSAttackAgent	IntelligentAgent	DefenceAgent
# StrategyAgent	93승 2패 5무	31승 38패 31무	21승 0패 79무	26승 19패 55무

IntelligentAgent

	baseline	SimpleBFSAttackAgent	StrategyAgent	DefenceAgent
# IntelligentAgent	89승 1패 10무	10승 27패 63무	0승 21패 79무	0승 23패 77무

DefenceAgent

	baseline	SimpleBFSAttackAgent	StrategyAgent	IntelligentAgent
# DefenceAgent	90승 0패 10무	29승 39패 32무	19승 26패 55무	23승 0패 77무

각 Agent끼리 랜덤 맵에 대해 총 100번의 게임을 진행시킨 결과이다.

b. Analysis

위 결과를 통해 baseline을 상대로 했을때, StrategyAgent, DefenceAgent, IntelligentAgent, SimpleBFSAttackAgent 순으로 성능이 좋음을 알 수 있다. 하지만 앞서 논의했듯이 게임 승패에 있어 Map의 영향이 크기 때문에 이것만으로는 StrategyAgent, DefenceAgent, IntelligentAgent의 우열을 가릴 수 없다. 이에 각 Agent들끼리 경기를 진행시켜 본 결과, SimpleBFSAttackAgent, StrategyAgent, DefenceAgent, IntelligentAgent 순으로 성능이 좋음을 알 수 있었다.

여기서 SimpleBFSAttackAgent가 다른 Agent에게 매우 강한 모습을 보여주는 이유는, 해당 Agent는 수비를 신경쓰지 않고 공격을 하는 반면에, 다른 Agent들은 상대 Agent가 자신의 진영에 들어와 있다면 높은 확률로 두 Agent 모두 수비를 하는데 사용하기 때문이다. 뿐만 아니라 SimpleBFSAttackAgent는 죽지도 않기 때문에, 다른 Agent들은 공격할 타이밍을 잡지 못하게 된다.

SimpleBFSAttackAgent 외의 Agent 끼리의 경기에서 수많은 무승부 경기가 나오는 것도 마찬가지이다. 서로가, 수비적인 자세를 취하다가 상대방이 실수를 하는 순간에 득점을 취하는 전략을 가지고 있기 때문에, 결국 서로 수비적인 자세만 취하다 그 누구도 죽지 않고 득점 없이 게임이 끝나는 것이다.

IntelligentAgent는 depth를 크게 설정하지 못하는 탓에 그 성능이 많이 좋지 않게 나왔다. 타 제작한 Agent들은 공격시, 목표 지점을 설정하고 절대 죽지 않는 경로로 이동하는 반면, IntelligentAgent는 딱 한 수 앞만을 바라보면서 수비를 하기 때문이다. 즉, 한 수 앞만 보고 잘못된 판단을 할 수 있고 이로인해 공격을 허용하는 순간 득점과 바로 연결되기 되는 것이다. 뿐만 아니라, 상대 Agent가 아군 진영에 넘어오기 전에도 계속 경계하며 수비적인 자세를 취하기 때문에 공격을 잘 하지 않는다.

c. your_best

4개의 Agent의 특징을 아래와 같이 정리할 수 있다.

- SimpleBFSAttackAgent: 매우 강력하면서도 안정적인 공격을 하는 Agent이다. 비록 baseline을 상대로 큰 활약을 하지 못하지만, 나머지 수비적인 Agent들에게 매우 강한 모습을 보여준다.
- StrategyAgent는 baseline을 상대로 가장 높은 승률을 보여주며, 과하지 않은 공격, 과하지 않은 수비의 밸런스가 특징이다.
- IntelligentAgent는 baseline에게는 강한 모습을 보여주지만, 타 Agent들에게 매우 약한 모습을 보여준다. depth를 충분히 키울 수 있다면 매우 좋은 성능을 낼 수 있다고 기대된다.
- DefenceAgent는 StrategyAgent를 개선시키고자 하였으나, 경기 진행 흐름의 특성상 큰 변화가 있지 않았다. 승률에서 이득을 볼 가능성이 있지만, 평균 점수 많이 떨어질 위험이 있다.

본 과제에서는 이기는 횟수 뿐만 아니라 함께 점수의 평균 또한 중요하게 평가되기 때문에, StrategyAgent를 your_best로 채택하게 되었다.