※ 본 아티클은 CMP MEDIA LLC와의 라이선스 계약에 의해 국문으로 제공됩니다

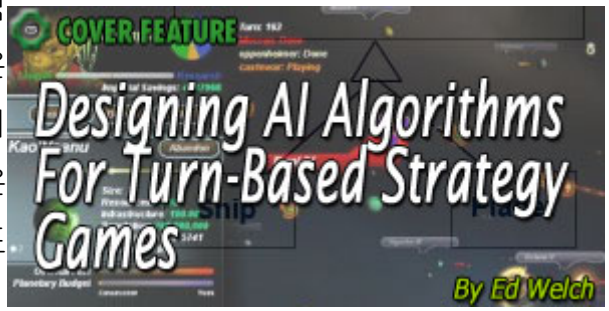
Gamasutra.com

턴 방식 전략 게임을 위한 AI 알고리즘 설계

Ed Welch

2007년 7월 27일

http://www.gamasutra.com/view/feature/1535/designing_ai_algorithms_for_.php



액션 게임에서 AI 상대들은 완벽한 정확성과 빛의 빠른 반사 같은 본래의 이점을 가지고 있다. 이 게임들의 AI를 설계한다는 것은 좀 더 인간과 같이 행동하게 만드는 놀라운 도전이다.

턴 방식 전략 게임에서 사정은 바뀌었다. 속도와 정확성은 더 이상 중요한 요소가 아니며, 플레이어의 재치와 직감은 어떤 AI 상대라도 쉽게 이겨버린다. 사실 숙련된 플레이어를 이길 수 있는 AI를 설계한다는 것은 불가능에 가깝지만, 어쨌던 그건 핵심이 아니다.

이 도전이란 것은 종국에 플레이어가 이길 수 있는 도전을 제공하지만 AI의 공격과 방어 전략을 영리하고 용의주도하게 함을 말한다. 플레이어가 AI의 전략에 익숙해지게 되면 게임은 급속도로 지루해지므로 많은 양의 예측 불가능한 요소가 요구된다.

복잡한 도전: 일반적인 전략게임 보기

AI 설계의 문제점은 실제 예로써 가장 쉽게 이해할 수 있다. 전쟁 게임 기반의 우주를 예로 들어 보겠다.

예로써 사용될 게임은 4X 게임(4X game)이라 불리며, 여러분은 은하를 확장, 정복해야 한다. 각각의 플레이어는 전함과 식민지 함선을 가지며 모 행성에서 시작해 거주 가능 행성을 식민지화할 수 있다.

AI 를 작성하는데 있어 첫 시도는 각각의 자원(행성이나 배)에 명령을 할당하는 간단한 알고리즘으로 가장 중요하다. 생산 행렬과 함께 행성을 방어하는 것은 가장 중요하기에 최고 우선 순위다.

2 번째로 우선 순위는 생산 행렬 없이 식민지를 방어하는 것이고 순서대로 적의 모 행성 공격, 거주 가능 행성 식민지화, 적 함선 공격, 손상된 함선 수리 순으로 내려간다. 마지막 가장 낮은 우선 순위는

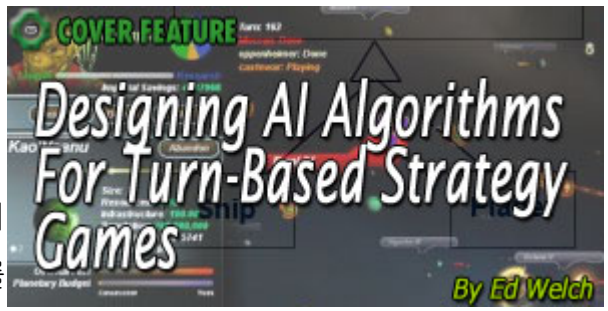
미지의 지역을 탐험하는 것이다. 그래서 우리는 최우선 순위 작업을 먼저 행한 다음 우리의 식민지 근처에 있는 적 함선을 확인하게 된다.

위의 그림에서 보는 바와 같이 적의 순양함 X & Y 가 AI 의 모 행성과 식민지를 위협하고 있다. 그래서 우리는 근접한 전함을 찾아 순양함을 공격하도록 할당을 내려야 한다. 여러분은 이 알고리즘에서 결함을 발견할 수 있을 것이다. 만약 먼저 순양함 Y 를 처리한다고 하면 구축함 A 가 가장 근접해 있기에 할당되게 된다. 그 다음 순양함 X 를 처리할 때 공격할 수 있는 유일한 함선인 구축함 X 는 너무 멀리 있어 순양함 X 는 모 행성에 대한 폭격을 계속 하게 된다. 구축함 B 가 순양함 Y 에 구축함 A 가 순양함 X 에 할당되는 것이 명확하다.

또한 이 간단한 알고리즘에는 다른 문제가 발생한다. 좀 더 복잡한 시나리오를 살펴보자.:

새 시나리오에서 우린 이전 공격으로 인해 심하게 손상된 구축함 A 를 보유하고 있다. 이 함선을 다시 전장으로 보내는 것은 의미 없는 희생이다. 수리를 위해 모 행성으로 돌려 보내는 것이 현명하다. 그리고 구축함 B 와 C 가 식민지를 방어한다. 하지만 구축함 C 는 순양함 Y 와는 너무 떨어져 있어 제 시간에 오지 못하며, 근처에 보이는(물론 연로 문제도 중요하다) 적 식민지를 폭격하는 것이 더 나을 것이다. 그러는 동안 AI 식민지 개척자는 무장을 하고 주요 식민지 임무에서 전환될 수 있을 것이다.

해결: 자원 할당 알고리즘



할당 점수

위에 적힌 문제를 해결하기 위해서 우선 우리는 점수 체계를 설계 해야 한다. 각각의 작업은 아래와 같이 일반적인 우선 순위가 할당된다.:

식민지 방어 : 1
적 식민지 공격 : 2
행성 식민지화 : 3
적 함성 공격 : 4
손상된 함성 수리 :5
미지의 지역 탐험 : 6

위에 적힌 문제를 각각의 직업은 우위 변경자를 가지는데 예를 들어 방어 작업은 식민지(생산 과정의 식민지는 높은 변경자를 가진다.)의 값에 대해 변경자를 가진다. 이왕 같이 수리 작업은 손상 정도에 따라 변경자를 가지고 식민지 작업은 행성의 “거주 가능성”에 따라 변경자를 가진다.

마지막으로 할당된 함선의 거리가 아래와 같이 계산된다.:

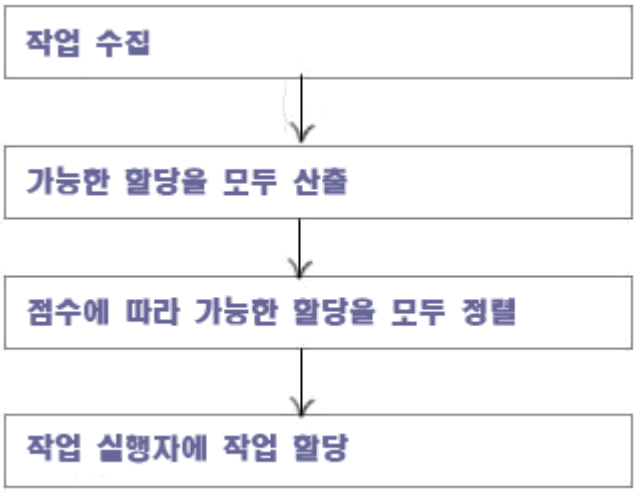
할당 점수 =(6 - 일반 우위 + 변경자) / 할당된 함선의 거리

그러므로 이전 시나리오에서 구축함 C는 방어 작업이 우위에 있더라도 적 행성에 근접해 있기 때문에 적 식민지 공격에 대해 높은 점수를 가진다.

또한 구축함 A에 대한 수리 작업의 우위 변경자는 함선이 크게 손상되었기에 꽤나 우위에 있다. 이는 수리 작업이 수리 행렬에 근접해 있기 때문에 방어 작업보다 우선 순위를 가지는 걸 동시에 의미한다.

알고리즘 개요

전체 알고리즘은 4 부분으로 나뉘어진다.:



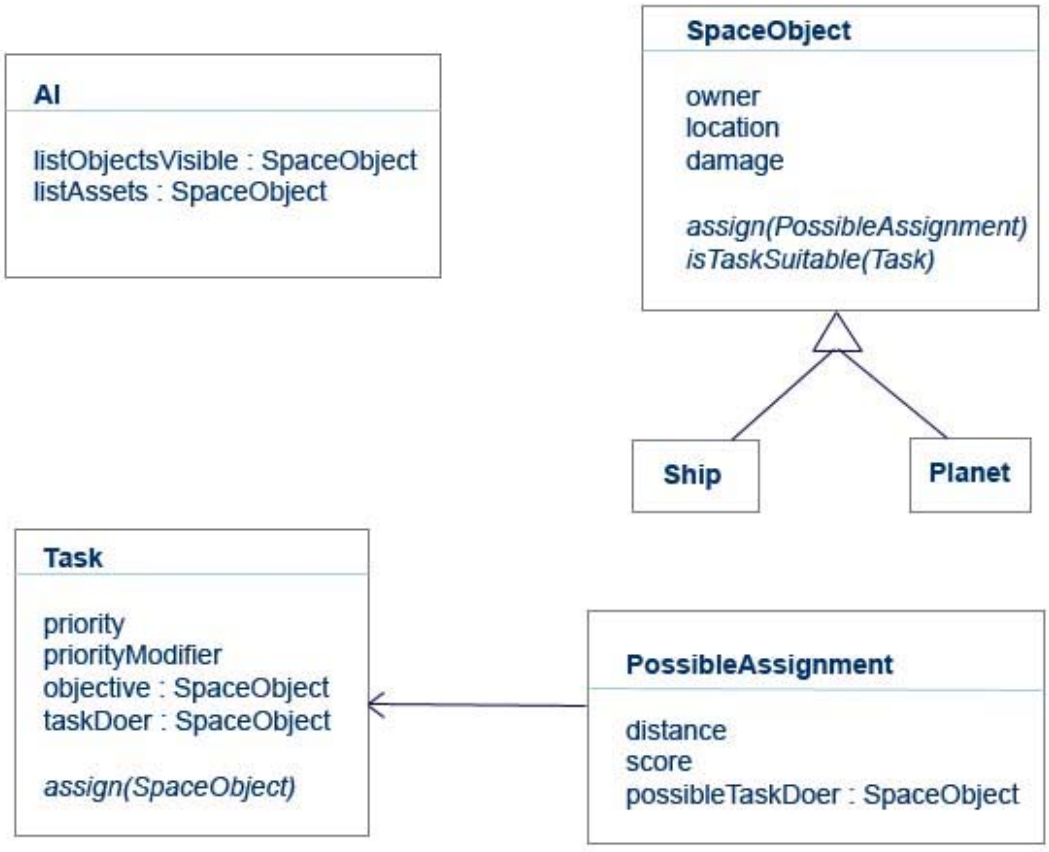
작업 집합

AI 는 감지 범위 내에 있는 적 함선과 행성 그리고 자신의 자산 목록을 가지고 있다. 행해야 할 작업은 아래와 같이 생성된다.:

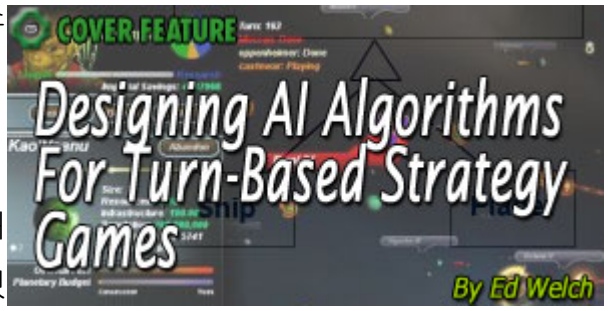
객체 등장	작업 생성
식민지 근처에 적 함선	식민지 방어 작업
적 함선	함선 공격 작업
적 식민지	행성 공격 작업
거주 가능한 행성	행성 식민지화 작업
손상된 함선	함선 수리 작업
미지의 지역	탐험 작업

실행 가능한 할당

또 다른 문제는 우리가 틀린 순서로 작업을 할당하면 자원 사용 효율이 좋지 않다는 것이다. 이것은 구절에서 작업을 할당하는 것에 의해 해결할 수 있다. 우리는 작업을 도와줄 2 개의 특별한 계급을 사용한다: 실행 가능한 할당과 작업. 실행 가능한 할당은 잠재적인“작업 실행자”를 작업과 연결하고 “할당 점수”를 저장한다. 작업은 우위, 우위 변경자 및 객체를 저장한다.



이해를 명확히 하기 위해 클래스 계층도를 보자.:



우리는 작업에 "작업 실행자"라 불리는 각각의 조합에 대해 실행 가능한 할당 객체를 생성했다. 그러나 불가능한 조합은 제거한다. 예를 들어 무장하지 않은 함선은 공격 작업을 수행할 수 없으며 목적지에 도달할 연료가 충분하지 않으면 작업을 할 수 없다. 이것의 코드는 아래와 같다.:

```
// listAsset에는 사용 가능한 모든 자산의 목록이 포함되어 있음(함선과 같은)
for (n = 0; n < listTask.size(); n++)
```

```
{
for (f = 0; f < listAsset.size(); f++) {
if (listAsset[f].isTaskSuitable(listTask[n])) {
listPossAssignment.add(new PossibleAssignment(listTaskn]));
}
}
}
```

다음으로 우리는 각각의 실행 가능한 할당에 대해 할당 점수를 계산하고 목록에서 가장 높은 점수가 위로 가도록 순서대로 정렬해야 한다. 최종적으로 우리는 물리적으로 할당을 만든 것이다. 목록은 정렬되었기에 가장 효율적인 할당이 먼저 일어난다. 할당으로 인해 작업 실행자가 만들어지게 되면 혼잡 상태로 표시되고 또한 작업이 할당된 것으로 나타나 중복 할당을 방지한다.

이것은 코드의 일부다. :

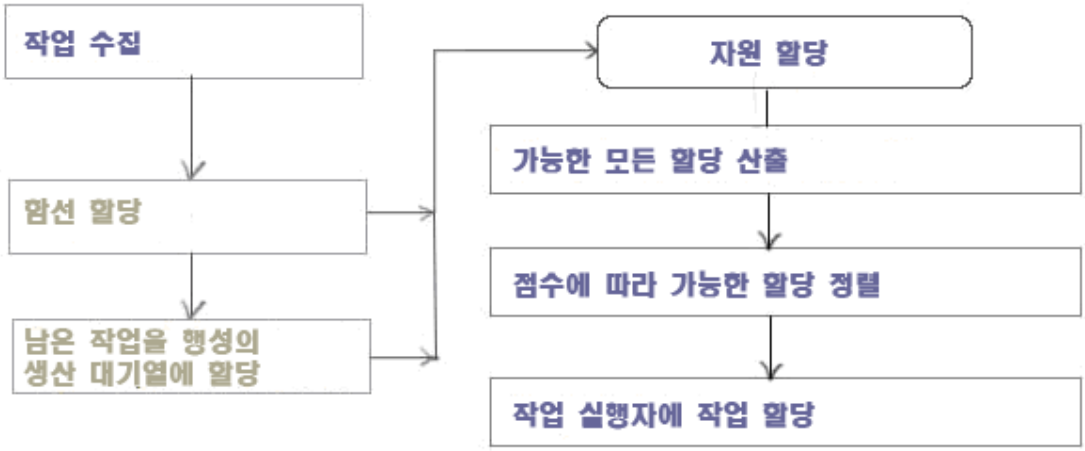
```
for (n = 0; n < listPossAssignment.size();n++)
{
listPossAssignment[n].assign();
}
public void PossibleAssignment::assign()
{
if (task.isAssigned()) return;
possibleTaskDoer.assign(this);
}
public void Ship::assign(PossibleAssignment possAssign) {
if (task != null) return;
task = possAssign.getTask();
possAssign.getTask().assign(this);
}
```

행성 생산 할당에 대한 알고리즘 재사용

AI는 존재하는 함대에 의해 처리할 수 없는 남은 작업이 있을 경우 새로운 우주선을 만들어야 한다. 예를 들어 우리가 적 함선을 찾았지만, 공격 가능한 전함이 없을 경우 새로운 전함을 만들 필요가 있다. 이와 동일하게 거주 가능한 행성과 사용할 수 있는 식민지 개척자 없을 경우 새로운 식민지 개척자를 만들 필요가 있다.

사실상 생산 행렬에서의 생산 우위는 우주선에 대한 작업 우위와 완전히 동일하다. 계급 다이어그램에서 보다시피 함선과 행성 양 계급은 SpaceObject에서 기인한다. 그래서 이 둘은 같은 알고리즘에서 적은 변경만으로 사용될 수 있는 것이다. 이것이 바로 객체 지향 설계에서 코드 재사용의 좋은 예이다.

아래의 다이어그램은 이것이 작동함을 보여준다. :



간소화하기: 이전 작업 포기

이것은 턴 방식의 게임이기에 지난 턴의 모든 작업은 각각 새로운 턴에서 유효하지 않다. 예를 들어 여러분의 구축함이 공격하려고 하는 적의 순양함이 후퇴했을 수 있고 당황스럽게도 여러분이 지배하려고 하는 식민지를 적이 이미 차지했을 수도 있다.

가장 쉽게 할 수 있는 일은 바로 모든 작업을 포기하고 각각 턴의 시작에서 자원 할당 루틴을 부르는 것이다. 이것은 모든 작업이 갱신될 필요는 없기 때문에 비효율적으로 보일 수 있다. 하지만 이전 턴에서의 작업을 유지할 필요가 없기 때문에 AI 가 복잡하지 않게 할 수 있다.

알고리즘은 매우 빠른 속도로 복잡하게 되며 디버깅과 유지가 어렵게 되는 경향이 있기 때문에 코드를 복잡하지 않게 유지하는 것이 AI 알고리즘에서 매우 중요하다. 또한 모든 최적화 작업은 알고리즘이 완료되고 첫 실행위치에서 알고리즘이 느려지지 않는다는 확실한 증거를 가진 후에 최종 단계에서 진행되어야 한다

턴 중간의 변수

증거를 우리의 턴이 진행되는 동안 아군 함선 중 하나가 새로운 적 행성이나 함선을 발견할 것이다. 우리는 바로 공격 작업을 할당해야 하지만 이 함선이 이미 중요한 작업을 가지고 있을 경우 문제가 발생한다. 간단하며 실수를 막는 최고의 방법은 단지 자원 할당 루틴을 다시 구동하는 것이다. 이것은 가장 최적화된 자원 할당을 보장하기 때문이다.

결론: 알고리즘 작업은 실제 어떠한가?

이 AI 알고리즘은 4X 전략 게임(예로써 추측했겠지만)의 개발 과정에서 설계되었다. 연습에서 적 함대의 조종에서 일정수준의 실제 지능이 있다는 인상을 받았다.

함선은 예상치 못한 상황에서 전략을 바꿔야 한다. 적 함선이 탄약을 소진했다면 바로 전장에서 물러나 기지로 돌아가 재무장을 하고 미지의 지역을 탐험할 수도 있다(유일하게 남은 유용한 작업일 경우). 조선소에서 새로운 함선이 만들어지면 전체 함대에 대한 명령이 바뀔

수 있다. 일부 함선은 수리를 위해 귀환하고 새로운 전함에게 공격을 맡긴다.

기본적으로 알고리즘은 시간과 노력을 투자할만한 비율로 작동과 디버깅이 하기 쉬운 꽤 복잡하지 않은 알고리즘을 제공하지만 아직 도전할만한 AI 상대를 보여주지는 못한다.

알고리즘은 게임에서 하나의 특화된 유형으로 설계되었다 하더라도, 쉽게 다른 유형의 전략 게임에도 적용될 수 있어야 한다.