


작품 요약서

■ Basic Data

작품명	스웨덴 오목게임 펜타고 인공지능			지원분야	Software
개발인원	1 명	본인역할	설계 및 개발	개발기간	2009.06 ~ 2009.08
개발환경	Windows				
개발언어	C언어, Win32 API				
개발툴	Visual Studio 2008				
작품소개 (요약)	<p>스웨덴의 전통 오목 게임 펜타고의 인공지능을 미니맥스 알고리즘을 기반으로 개발 하였습니다. 이 게임은 네 부분으로 나뉘어진 6x6의 보드 위에서 진행되며 게임 방법은 다음과 같습니다.</p>  <p>구슬을 놓고 90도로 판을 회전시켜 5개의 구슬이 그림과 같이 나란히 일직선이 되면 승리한다.</p>				
작품내용	<p>1. 게임진행 부분 : 사용자와 인공지능으로부터 번갈아 가며 수를 입력받고, 진행상황을 출력하며, 종료시 어느 플레이어가 이겼는지 알려줍니다.</p> <p>2. 인공지능 부분 : 보드의 상태를 입력 값으로 넘겨주면, 일정시간 동안 인공지능 탐색을 하여 최적의 수를 찾습니다.</p>				
개발내용 (본인 구현부분)	<p>인공지능에 중점을 두어 개발하였습니다. 미니맥스 알고리즘을 기본으로 하여 알파베타 가지치기, 2중해싱을 이용한 중복제거, 휴리스틱한 우선순위 탐색, 시간제한을 둔 반복 심화 탐색 등을 적용하여 최적화된 알고리즘을 구현 했습니다.</p>				
기타	<p>그 동안 중점을 두어 공부했던 기초 알고리즘, 자료구조를 응용하여 문제에 맞는 새 알고리즘을 설계 하였습니다. 기초학문에 대한 이해도와 응용력을 보여줄 수 있는 프로그램이라 생각하며, 게임 인공지능 이라는 저의 관심분야에 대한 열정을 보여줄 수 있는 좋은 기회라는 생각이 듭니다.</p>				

■ System Architecture

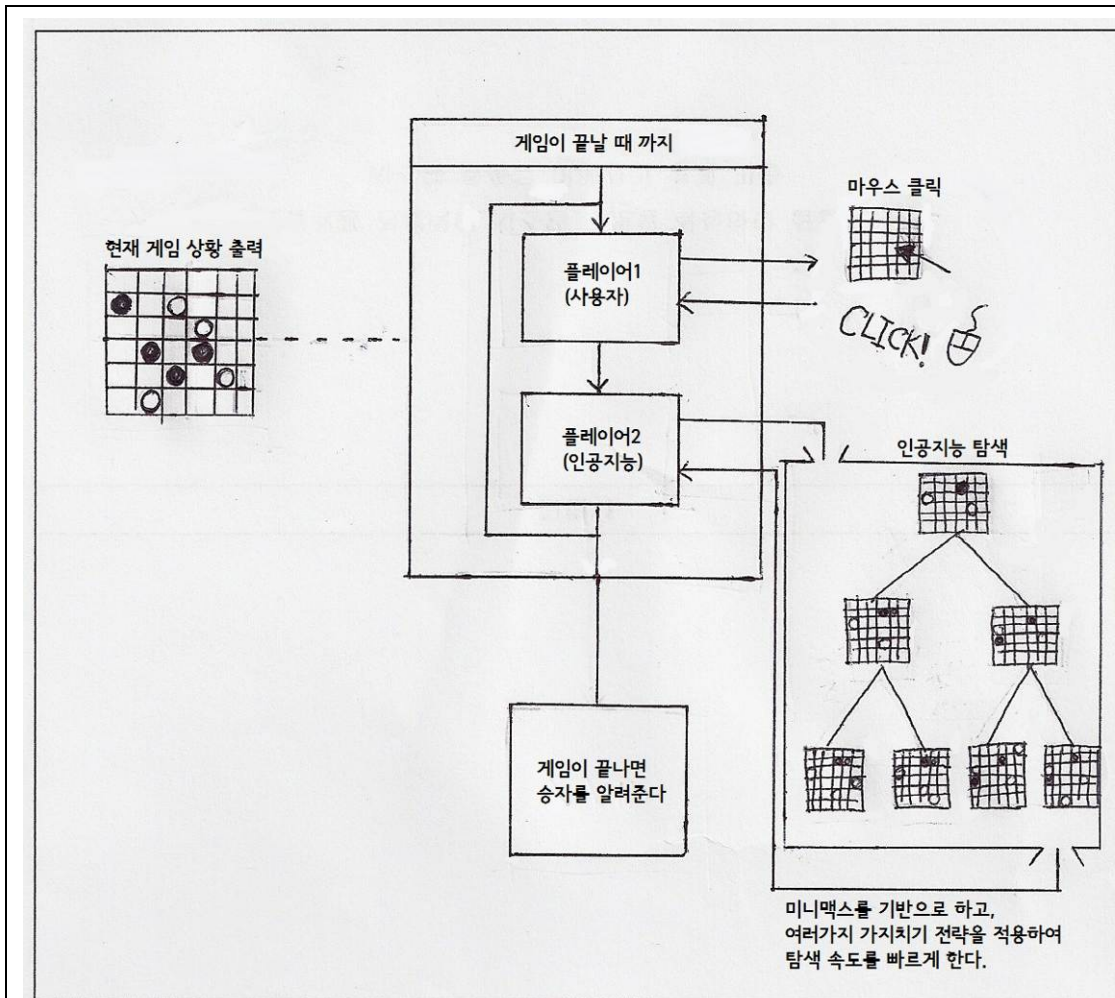


그림 1

둘 중 한 명이 5목이 되거나, 더 이상 둘 곳이 없어 게임이 끝날 때까지 게임이 진행됩니다. 플레이어1(사용자)과 플레이어2(AI)가 번갈아 가면서 두게 되는데, 플레이어1(사용자)은 마우스를 통해 입력 받고, 플레이어2(AI)는 현재 상태를 인공지능에 넘겨주었을 때 인공지능 탐색을 통해 구해진 최적의 좌표를 넘겨받습니다. 게임이 진행되는 동안 모니터에 실시간으로 게임 진행 상황이 출력 됩니다. 게임이 끝나면 승자가 누구인지 출력합니다.

인공지능 탐색은 양쪽 플레이어가 최선을 다한다는 가정하에 만들어진 상태공간 트리를 탐색하게 됩니다.(미니맥스 알고리즘) 이 때 빠른 탐색을 위해 알파베타 가지치기, 2중해싱을 이용한 중복제거, 휴리스틱한 우선순위 탐색, 시간제한을 둔 반복 심화 탐색 등을 적용하였습니다.

■ Software/Hardware Architecture

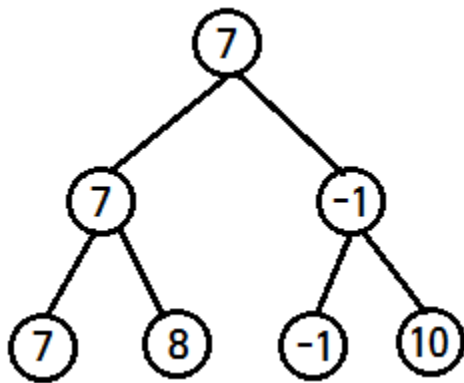


그림 2

1. 미니맥스 알고리즘

오목, 장기 등의 게임을 할 때 우리는 상대방의 행동을 예측하여 몇 수 앞을 내다봅니다. 내가 최선을 다하는 것처럼 상대방도 최선을 다해 플레이를 합니다.

어떠한 평가 기준에 근거, 상대방이 최대한 자신에게 불리한 수를 둘 거라는 가정하에 가장 유리한 수를 택하는 것. 이 것이 미니맥스 알고리즘의 기본 아이디어입니다.

간단한 예제를 그림2에 그려보았습니다. 깊이가 0(내턴)일 때는 자식노드 중 최대값, 깊이가 1(상대방턴)인 노드에서는 최소값을 취합니다.

2. 알파베타 가지치기

-1의 값을 가지는 단말노드를 탐색할 때 -1보다 큰 수는 선택되지 않음이 확실합니다. 상대방이 돌을 놓는 홀수 번째 노드에서는 최소값을 택하기 때문에 구해지는 값이 더 작아질 수는 있지만 커질 수는 없습니다.

그런데 최상위 노드에서는 지금까지 구한 7이라는 값보다 더 큰 값을 얻어야만 합니다. -1보다 작은 수는 7보다 더 큰 값이 될 수 없으므로 가지치기 됩니다.

3. 평가함수

노드의 가중치는 어떤 상태가 나에게 유리한 정도를 나타냅니다. 단말 노드에서는 다음과 같은 평가함수를 통해 이 것을 구합니다.

- 1) 내가 승리하였을 경우 : $+\infty$
- 2) 상대방이 승리하였을 경우 : $-\infty$
- 3) 비겼을 경우 : 0
- 4) 그 외의 경우 : 내가 오목을 만들 수 있는 줄의 수 - 상대방이 오목을 만들 수 있는 줄의 수

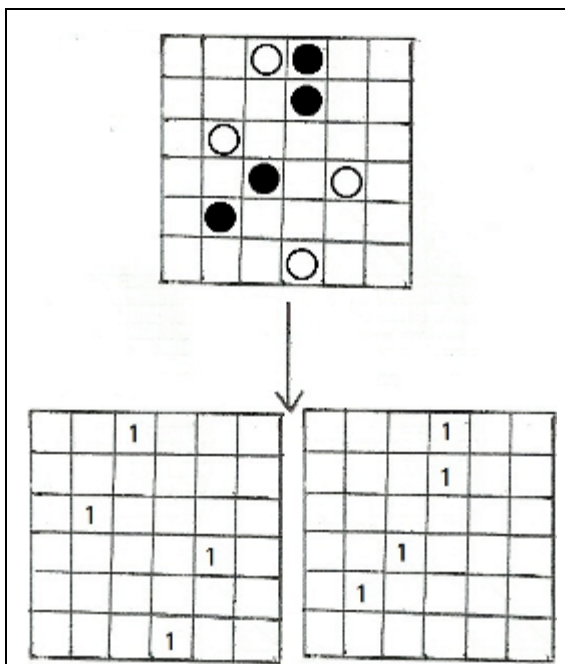


그림 3

4. 상태를 64비트 정수 2개로 표현

상태를 6x6의 배열로 표현 할 수도 있지만 비트연산을 통해 2개의 정수로 표현하여 메모리를 절약하고, 속도가 빠른 비트연산을 사용할 수 있습니다.

이러한 상태표현법 덕분에 중복 가지 제거를 위한 해시 테이블을 사용할 수 있게 되었습니다. 메모리를 적게 쓰고, 해시함수를 구현하기 쉽기 때문입니다.

그림5에서 아래의 두 사각형이 각각의 정수를 나타내는데 왼쪽은 흰 돌, 오른쪽은 검은 돌의 정보를 담고 있습니다. 왼쪽 위부터 오른쪽 아래까지 0부터 35번째 비트입니다.

■ Software/Hardware Architecture

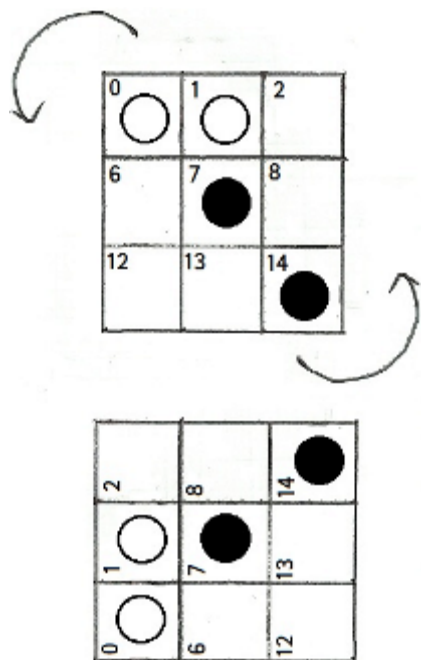


그림 4

5. 비트연산을 통한 속도향상

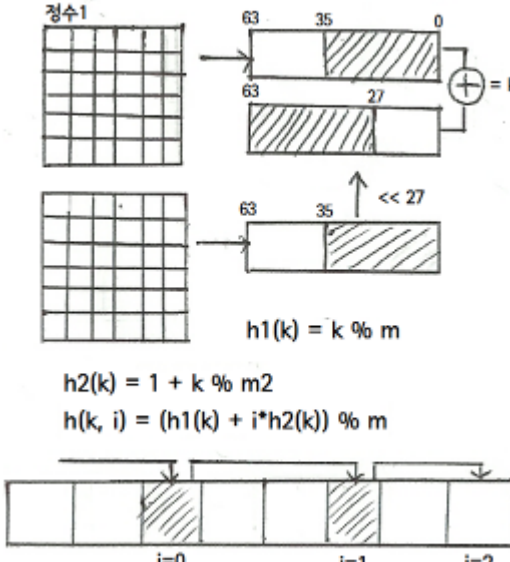
다음은 왼쪽 위에있는 3x3의 보드를 반시계 방향으로 회전하는 코드입니다.

```
s1 |= ((t1 & ((__int64)1 << 2)) >> 2);
s1 |= ((t1 & ((__int64)1 << 8)) >> 7);
s1 |= ((t1 & ((__int64)1 << 14)) >> 12);
s1 |= ((t1 & ((__int64)1 << 13)) >> 5);
s1 |= ((t1 & ((__int64)1 << 12)) << 2);
s1 |= ((t1 & ((__int64)1 << 6)) << 7);
s1 |= ((t1 & ((__int64)1 << 0)) << 12);
s1 |= ((t1 & ((__int64)1 << 1)) << 5);
s1 |= (t1 & ((__int64)1 << 7));
```

이와 같이 프로그램 곳곳에 비트연산을 활용하여 속도를 빠르게 하였습니다.

※ 개발작품을 최대한 부각 시킬 수 있도록 작성 (그림의 개수 제한 없음)

■ Software/Hardware Architecture

 <p>정수1</p> <p>63 35 0</p> <p>63 27</p> <p>63 35</p> <p>$h1(k) = k \% m$</p> <p>$h2(k) = 1 + k \% m2$</p> <p>$h(k, i) = (h1(k) + i * h2(k)) \% m$</p> <p>i=0 i=1 i=2</p>	<p>6. 해시 테이블을 이용한 중복제거</p> <p>탐색을 하다 보면 다른 경로를 통해 같은 상태에 다다른 경우가 있습니다. 이럴 때는 가지치기가 가능합니다. 같은 상태에서 뿔어나간다면 다시 탐색을 해도 동일한 상태들을 탐색하게 되기 때문입니다.</p> <p>한 번 탐색한 상태는 체크를 해두어 다시 지나지 않도록 한다면 속도향상을 기대할 수 있을 것입니다. 그런데 곧바로 배열에 체크하기에는 상태의 경우의 수가 3^{36}으로 너무 많습니다. 그래서 해시 테이블을 사용하였습니다.</p> <p>7. 이중 해싱을 이용하여 해시값 구하기</p> <p>먼저 상태를 나타내는 두 정수를 XOR연산을 통하여 하나로 합쳤습니다. 이 때 상태의 정보를 최대한 반영하기 위하여 검은돌의 상태를 담고 있는 정수를 왼쪽으로 27칸 시프트 연산 하였습니다.</p> <p>그렇게 해서 얻은 k값을 해시테이블의 크기인 m으로 나누어 해시값을 얻었습니다. 그런데 해시충돌이 일어날 경우를 대비해서 또 다른 보조함수 $h2(k)=1+k\%m2$를 사용했습니다.</p> <p>이렇게 하면 해시충돌이 일어날 경우 빈 공간을 찾을 때 까지 반복문을 돌며 이중 해싱을 하였기에 조금 더 충돌이 적게 일어납니다.</p>
<p>그림 5</p>	

8. 우선순위 탐색

자식 노드들에 대한 평가함수값을 구하여 정렬을 한 뒤 탐색을 진행하면, 가중치가 최적 값에 가까운 노드를 먼저 탐색 하기 때문에 가지치기가 더 빨리 일어나게 됩니다.

자식노드를 구한 뒤 바로 함수 호출을 하지 않고, 일단 리스트에 넣어두고 정렬하여 마지막에 일괄적으로 함수 호출을 하는 방법으로 구현했습니다.

9. 시간제한을 둔 반복 심화 탐색

모든 경우를 다 탐색하기엔 시간이 부족하기 때문에 탐색할 깊이를 미리 정해두는 방법이 있습니다. 그런데 초반엔 탐색 시간이 오래 걸려 깊은 곳 까지 탐색할 수 없지만 뒤로 갈수록 더 깊은 곳 까지 탐색 할 수 있습니다. 이러한 차이 때문에 깊이를 미리 정해두는 것은 효율적이지 않습니다.

대신 시간 제한을 두고, 깊이를 늘리면서 탐색 하였습니다. 이런 방법을 쓰면 게임의 초반에는 낮은 깊이에서 시간이 초과 되지만 후반에는 더 깊은 곳 까지 탐색하게 되어 효율적인 탐색이 가능 합니다.

■ Function Explanation (Applicant)

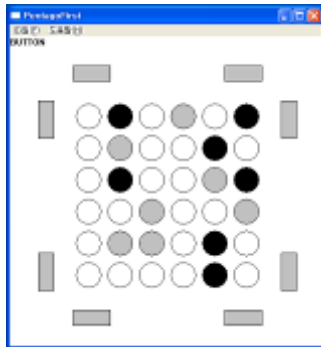


그림 6

1. 게임 진행 부분

인공지능에 중점을 두어 개발하였고, 게임 플레이는 인공지능이 잘 돌아가는지 테스트 하는 것에 초점을 맞추어 개발 하였습니다.

가운데 보드에 돌을 놓고, 네 귀퉁이에 있는 네모칸을 클릭하여 3x3의 작은 판을 회전 시킵니다.

사용자의 입력이 끝나면 바로 인공지능이 작동하여 최적의 값을 찾습니다. 게임이 끝날 때 까지 돌아가며, 끝났을 경우에는 누가 이겼는지 보여줍니다.

2. 인공지능 추론 부분

사용자와 게임을 할 수도 있지만 인공지능끼리 맞대결을 할 수도 있게 만들었습니다.