3.주차 : 2주차 연속

교재 3장: 탐색을 통한 문제 해결

3.1 소개

우리는 검색 전략의 종류를 학습했다. 이전 장에서 다양한 미지탐색 전략을 살펴보았다. 미지 탐색 방법은 문제에 관련된 지식이 부족하기 때문에 많은 경우에 매우 비효율적이다.

문제의 특정 지식을 사용하면 극적으로 탐색 속도를 향상시킬 수 있다. 이 장에서는 문제의 특정 경험을 사용하는 미지 탐색을 다루고자 한다.

* 탐색 전략의 복습
  1. 블라인드 검색

a) 깊이 우선 탐색

b) 너비 우선 탐색

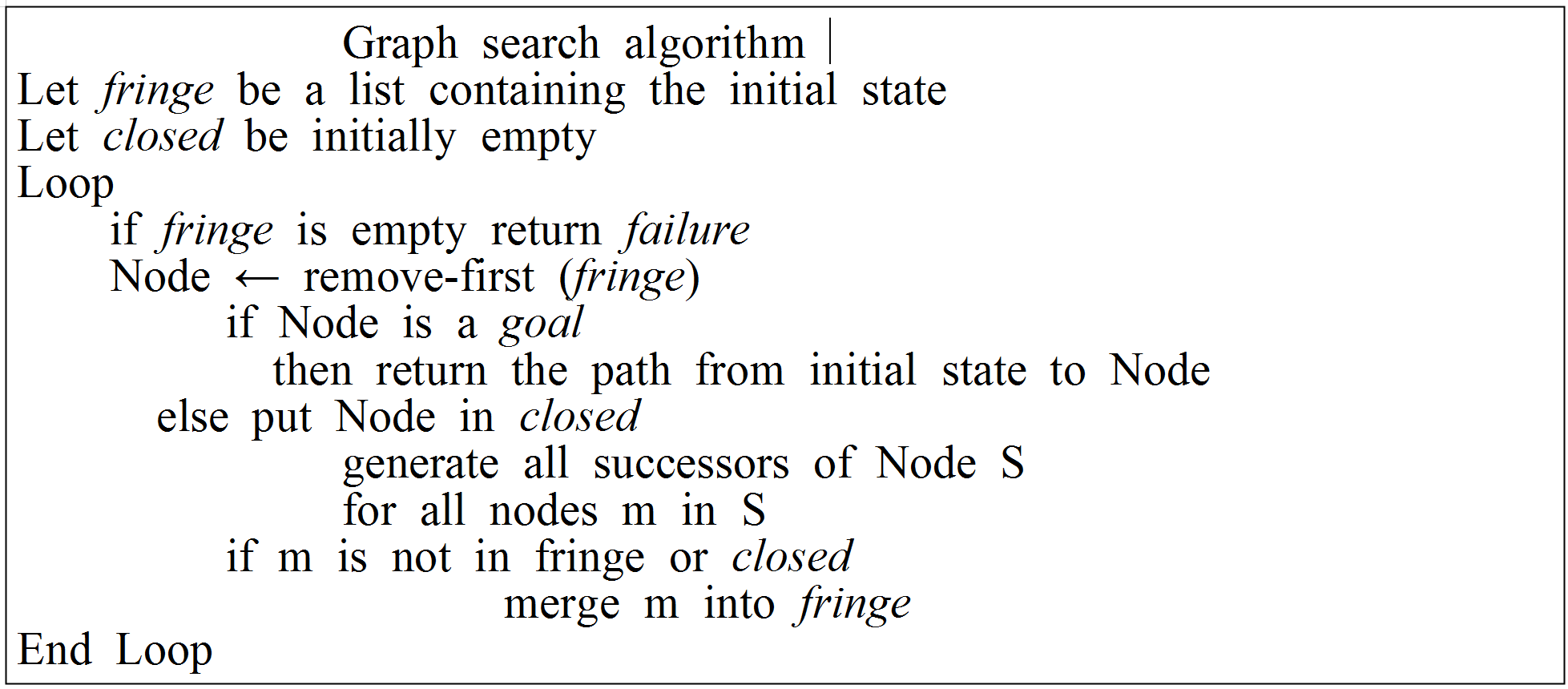
c) 반복 심화 검색

d) 양방향 검색

* 1. 정보에 근거한 검색

3.1.1 그래프 탐색 알고리즘

아래에 보인 알고리즘은 일반적인 그래프 탐색 알고리즘이다.



3.1.2 균일 비용 탐색 (UCS: Uniform Cost Search)의 복습

* 앞에서 학습한 너비 우선 탐색의 변형인 균일 비용 탐색을 다룬다.
* 너비 우선 탐색에 소요된 일정한 경로 비용을 고려하여 탐색이 이루어지면 균일 비용 탐색이 된다.
* 비용함수, g (n)을 시작 노드에서 현재 노드 n까지 경로 비용이라 하자.
* 알고리즘은 g의 값을 증가시키는 순서로 프린지에 노드를 정렬하고, 프린지의 최소 비용의 노드를 확장한다.
* 균일 비용 탐색의 특성

• 완전성

• 최적 / 허용 가능

• 지수적 시간과 공간의 복잡성, O (bd)

UCS(Uni focus?)가 알고리즘은 노드 확장의 순서를 선택하기 위해 g (n)의 값을 사용한다. 이제는 문제의 특정한 체험적 정보를 사용하여 탐색하는 정보 기반 탐색 또는 경험적 탐색을 소개하기로 한다. 경험 지식은 노드 확장의 순서를 선택하기 위해 사용된다.

3.1.3 지식 기반 탐색(정보 기반 탐색 또는 경험적 탐색)

3.2 경험적 탐색

체계적 탐색법은 탐색 대상인 상태 공간을 있는 그대로 모두 탐색해 나가는 방법이었다. 그 특정은 아래와 같이 나열할 수 있다.

* 해의 발견은 보장된다.
* 상태 공간이 커지면 목표 상태가 존재하는 위치에 따라 해를 발견할 때까지의 시간이 매우 길어진다.

이에 반해， 탐색하는 지점을 어떤 평가 기준으로 우선 순위를 붙여서 그 순서대로 조사해 나감으로써 탐색 효율을 향상시키는 방법으로 휴리스틱 탐색법 (heuristic search) 이 있다.

휴리스틱 탐색법이란 예를 들어 미로에서 탈출할 때 출구가 있는 방향으로 진행한다는 방침을 가지고 행동한다면 출구에 빨리 도착할 가능성이 높아진다는 등의 경험으로 얻을 수 있는 지식에 기반하여 탐색을 수행 하는 탐색 방법이다.

이와 같이 휴리스틱 탐색법은 경험적 지식을 이용하여 상태 공간의 탐색 효율을 향상시킨다. 특히 그 특정으로써 탐색에서의 경험적 지식， 즉 휴리스틱 함수 (heuristic function) 라고 하는 평가 함수 (evaluation function)를 이용한다. 이것은 탐색을 효율적으로 수행하기 위하여 현재 상태가 목표 상태로부터 어느 정도로 평가되는지 알아보기 위한 것이다. 대표적인 휴리스틱 탐색 방법들인 언덕 오르기 탐색 (hill-climbing search) 최적 탐색 (optimal search) 최고 우선 탐색 (best - first search), A\* 알고리즘 (A -star algorithm) 등이 있다.

3.1.3.1 경험(정보, 지식)

경험은 "엄지 손가락의 규칙"을 의미한다. 주디어 펄(Judea Pearl)의 말을 인용하면, "경험들은 기준, 방법이나 원칙 등으로 목표에 도달하기 위하여 가장 효율적인 탐색을 약속한다. 상태는 행동(연산자)에 의해 전환되므로 목표를 달성하기 위해 변형된 행동들의 과정(경로)들 중에서 가장 효율적인 경로의 선택은 경험에 의해 가능하다.

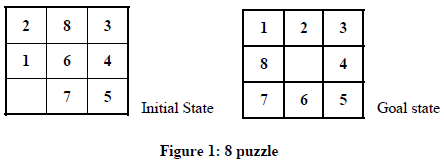
* 경험 함수의 예

노드 n에서 경험 함수, h(n)은 현재 노드에서 목표까지 최적의 비용을 추정한다.

* + 예 1: 서울에서 부산까지의 경로

서울에서 경험(지식)은 서울에서 부산까지의 유클리디안 거리(직선거리)이다. 이를 경험 함수로 표현하면

h(n)=유클리디안 거리(서울, 부산)

* + 예 2: 8 - 퍼즐: 잘못 배치 된 타일 지식은 제 자리가 아닌 타일의 수이다.

위 그림에서 좌측은 현재 상태 n 이고 우측은 목표 상태를 나타낸다.

타일 2, 8, 1, 6, 7(총 5개)이 목표 상태의 자리에서 벗어나 있기 때문에 경험함수

h (n) = 5

* + 예 3: 8 - 퍼즐 의 경험은 다르게 정의될 수도 있다. 즉 목표위치에서 벗어난 타일들의 거리이다. 이를 맨하튼 거리라고 부르기도 한다.

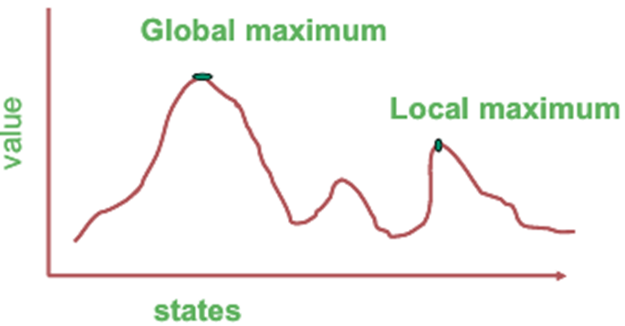
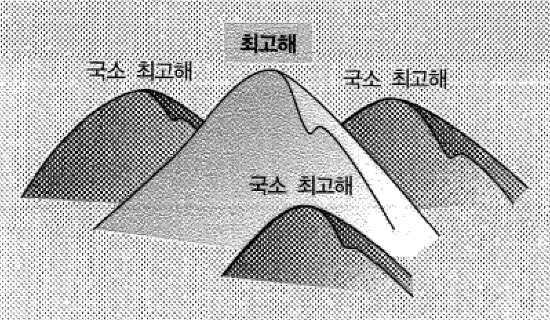
즉, 타일의 거리는 목표 상태에서 벗어난 타일의 x-위치와 y-위치의 차를 합한 거리로 측정(맨하튼 거리)된다.

따라서 타일 그림에 적용한 경험 함수 아래와 같다.

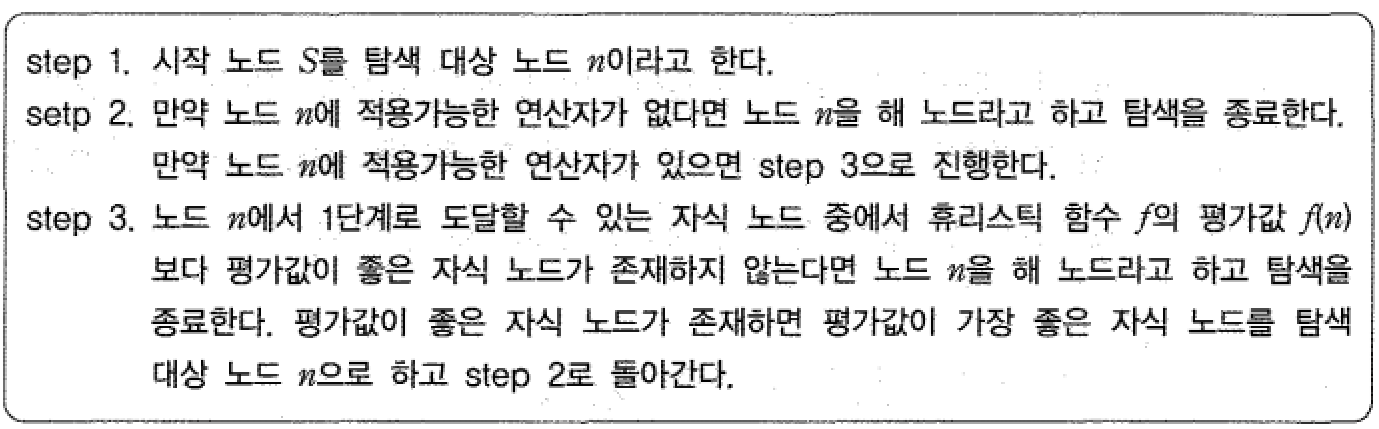
h (n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6

3.1.3.2 언덕 오르기 탐색(hill climbing search)

휴리스틱 함수를 이용하는 방법 중에서 가장 간단한 것이 언덕 오르기 탐색이 라는 탐색법이다. 언덕 오르기 탐색은 현재 상태로부터 휴리스틱 함수의 평가치가 높은 쪽으로 진행해 가는 방법이다. 이것은 현지 점보다 높은 장소로 이동해 가는， 산 정상을 향해서 등반하는 행위와 닮아 있다는 점에서 “언덕 오르기”라는 이름이 붙게 되었다 언덕 오르기 탐색은 알고리즘이 단순하고 현재 상태를 개선하는 것을 목표로 하는 것으로 자주 이용되고 있는 탐색법이다. 현재 상태에서 다음 상태로의 이동 조건에서 현재 상태보다 휴리스틱 함수의 평가값이 높은 경우에 다음 상태로 이동하기 때문에 산 정상에 도달하기 전에 봉우리에 도달하게 되면 그곳을 해라고 생각해버리게 된다. 그래서 이른바 국소(최고)해(Iocal best solution)로 수렴해버려서 최고해 (best solution)를 발견할 수 없는 경우가 있다는 단점을 가지고 있다. 그로 인하여 탐욕적 국소 탐색법(greedy local search)이라고도 한다.

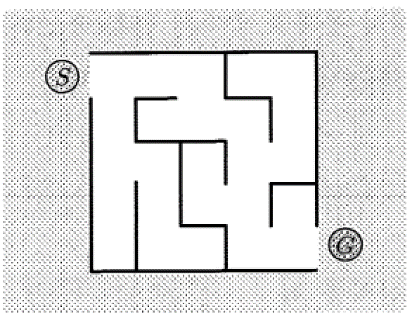
 

* 언덕 오르기 탐색의 알고리즘

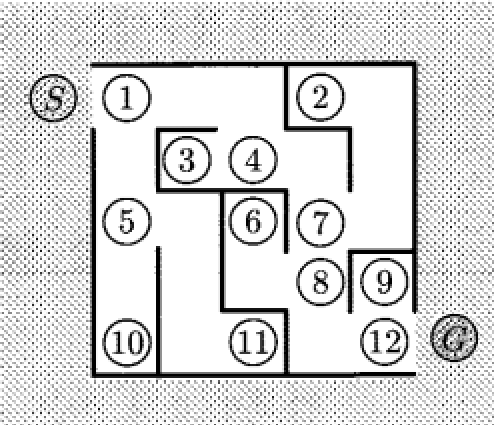


* 언덕 오르기 탐색 예: 미로 찾기

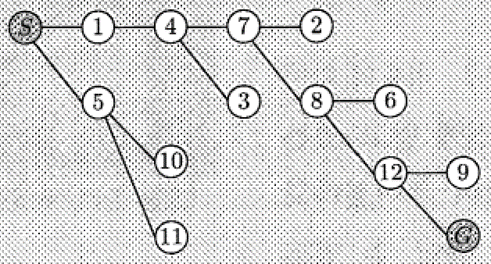
언덕 오르기 탐색을 그림에 나타낸 미로에 적용하자.



미로를 그림 과 같이 갈림길 혹은 막다른 지점을 노드로 번호를 부여한다.



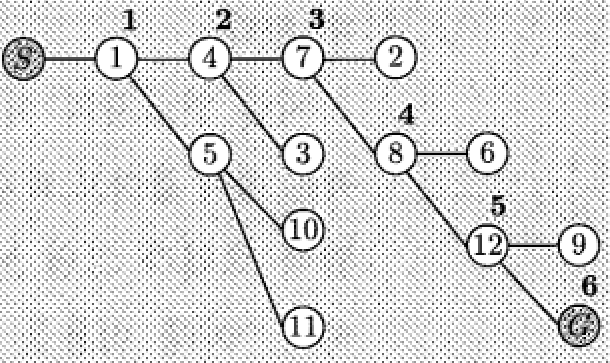
미로를 그래프로 표현한다.



미로 안에서 현재 지점의 좌표를 (x, y) 라 하고， 종착점의 좌표를 (a, b)라 하면, 평가 함수 f(n)은 예를 들어 다음과 같이 나타낼 수 있다.

f(n)= la -xl + Ib -yl

이 예의 경우 평가함수의 값이 작아지는 방향으로 진행하게 된다. 위에 나타낸 언덕 오르기 탐색의 알고리즘을 적용하여 탐색을 진행하면 목표점까지의 경로는 s →1→4→7→8→12→G가 된다. 탐색 순서는 다음과 같다.

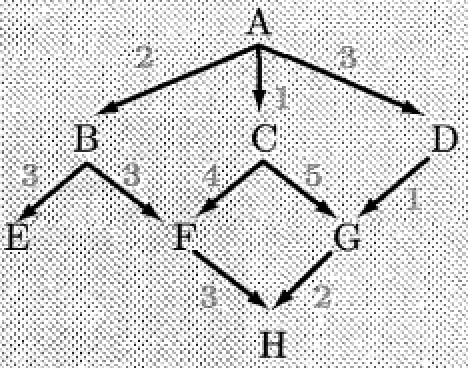


이 미로의 경우, 언덕 오르기 탐색의 평가 함수를 이용하여 목표까지 도착하는 미로였지만， 복잡한 미로의 경우에는 목표까지 도착하지 못하고 막다른 골목(국소(최고)해에 해당)에서 탈출하지 못하고 탐색이 끝나고 마는 경우가 자주 존재한다.

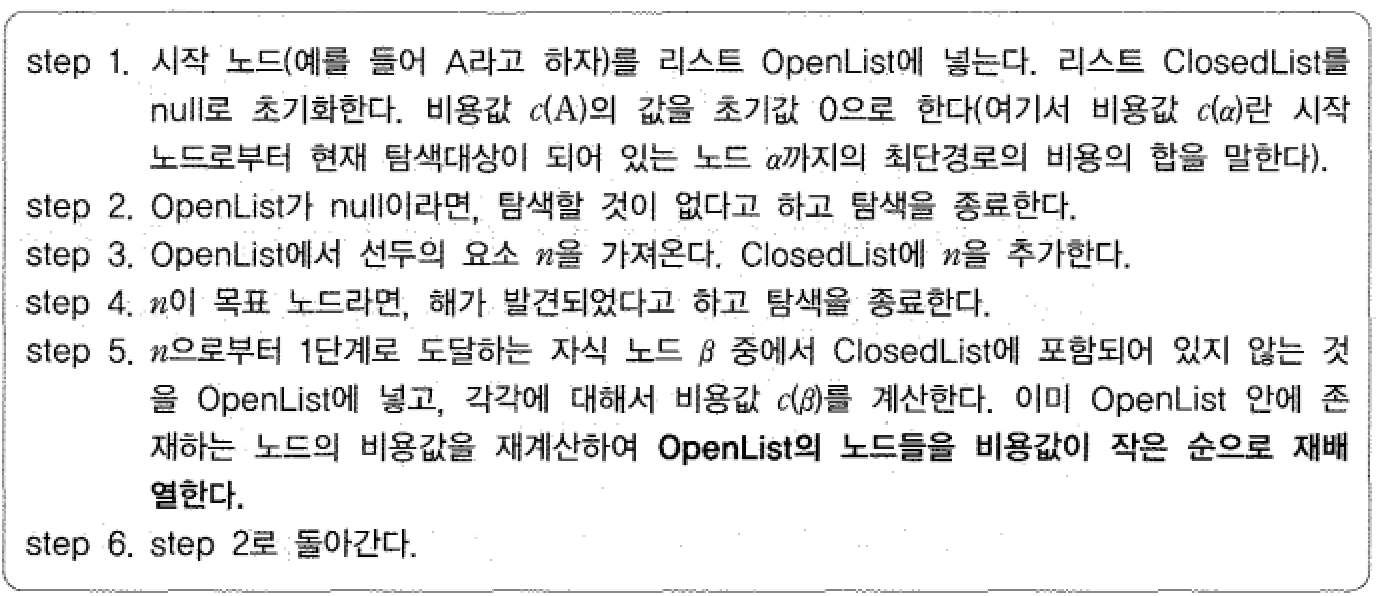
3.1.3.3 최적 탐색(optimal search)

탐색하고자 하는 문제에서 연산자를 적용했을 때의 비용을 사전에 알고 있다면， 그 비용을 고려하여 최적의 탐색을 수행할 수 있다.

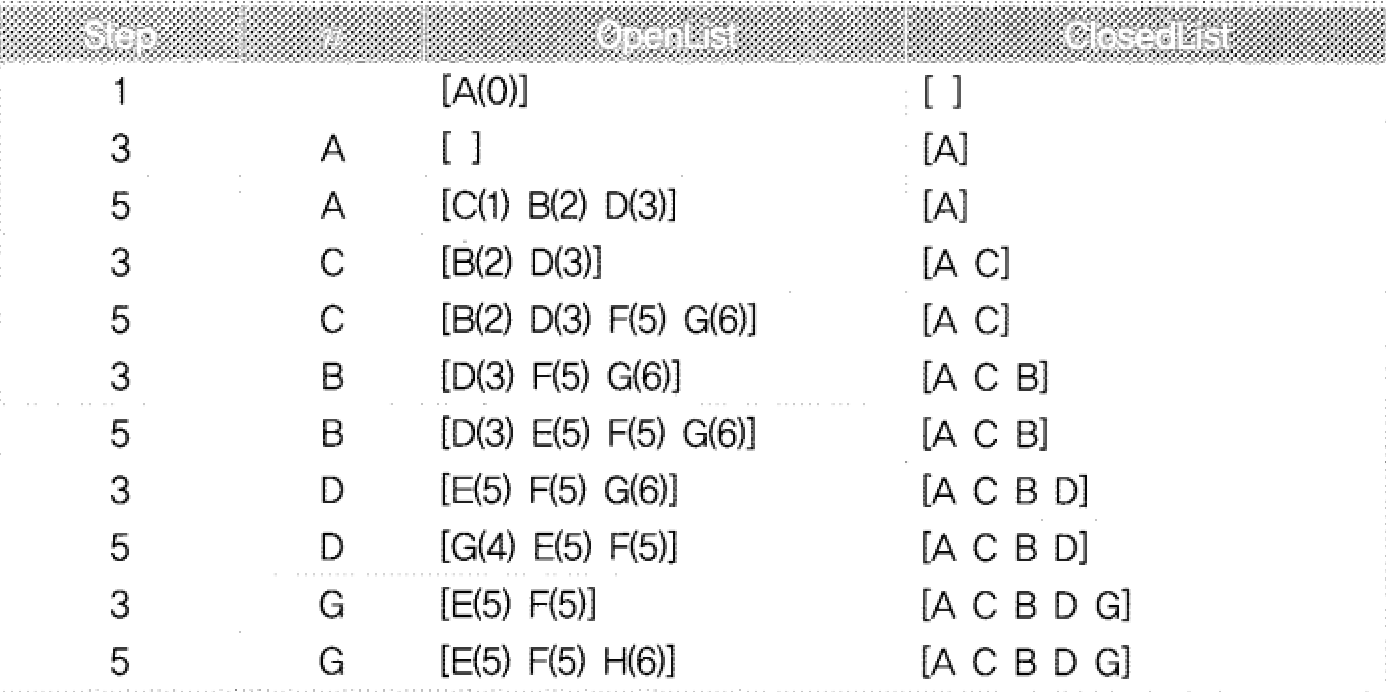
지금 연산자의 비용(이 경우， 비용의 값이 작은 쪽이 좋다고 하자)을 사전에 알고 있는 탐색 트리가 그림과 같이 주어졌다고 하자. 이때 비용을 가급적 줄일 수 있는 최적의 탐색을 수행하도록 한다.



아래에 최적해 (optimal solution) 의 탐색 알고리즘을 나타낸다.



아래에 나타낸 것처럼 최적 탐색에서는 시작 노드로부터 현재 탐색하고 있는 노드까지의 총비용을 계산하면서 탐색을 계속한다. 표에서 Openlist에 포함된 노드 옆의 ( )는 비용을 나타낸다. 같은 노드라도 탐색 경로에 따라 시작 노드로부터의 비용이 변하기 때문에 탐색 경로를 따라 노드 마다 비용의 재계산을 수행하여 최적의 경로를 발견한다. 예를 틀어, 표 에서 탐색 대상 노드가 D가 되었을 때 G로 가는 경로를 A → C → G로 계산한 비용 ‘6’에서 A → D → G로 재계산한 비용 ‘4’로 변경되는 것을 할 수 있다.



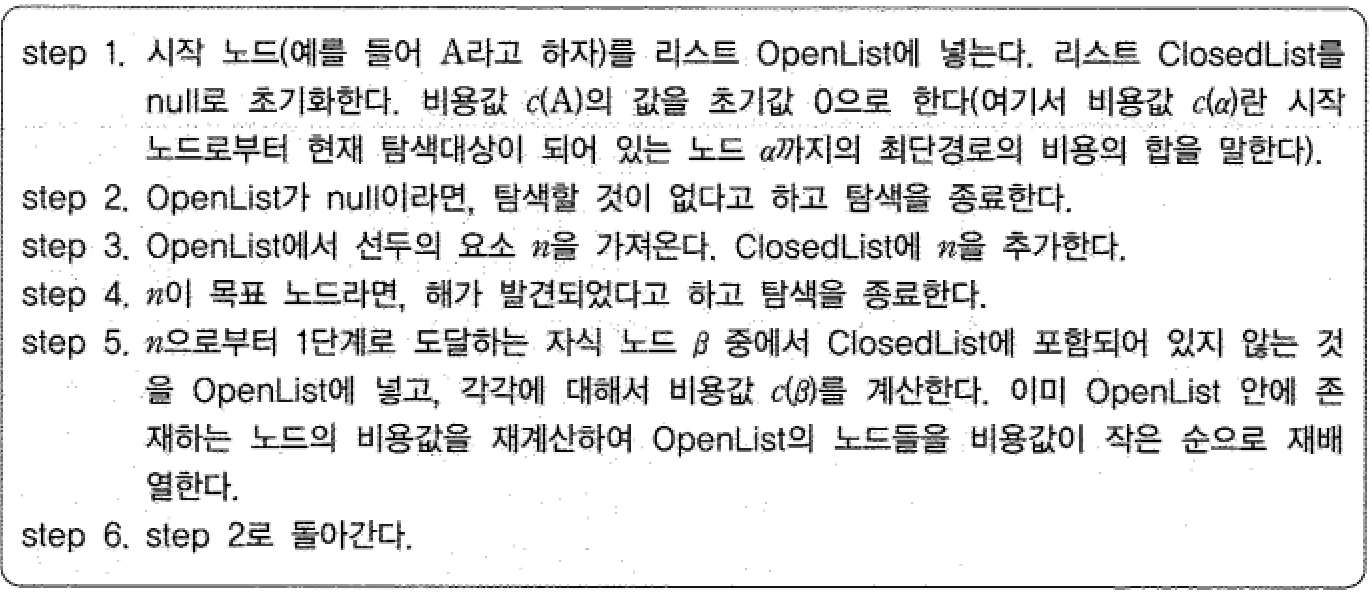
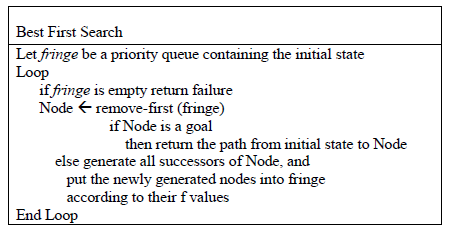
3.1.3.4 최고 우선 탐색

앞에서 언급한 탐색 방법은 연산자를 적용할 때의 탐색 비용을 사전에 알고 있다는 것을 전제하고 있었다. 그러나 실제 문제에서는 탐색 비용을 사전에 알고 있을 경우는 많지 않고, 문제에 따라서는 탐색에서 방문한 노드로부터 상황을 판단하여 목표에 도달할 때까지의 비용이 최소가 되도록 목표까지의 비용을 추정하면서 탐색을 수행하는 방법이 사용된다. 이와 같은 탐색 방법을 최고 우선 탐색이라고 한다.

최고 우선 탐색에서는 “최적”의 해는 보증할 수 없지만 문제 -해결을 유효한 시간 안에 수행하는 “최고”의 해가 되도록 탐색을 수행한다.

현재 방문하고 있는 노드로부터 목표까지의 추정이 가능하다면 탐색 도중에 다음에 나아갈 노드를 선택할 때 가능하면 목표까지의 추정 비용이 작은 노드를 선택하는 것이 가능하다.

아래에 최고 우선 탐색 알고리즘을 나타낸다.



균일 비용 탐색은 최고 우선 탐색 알고리즘의 특별한 경우이다. 이 알고리즘은 이전에 탐색된 노드들의 대기열을 갖고 있으며, 비용 함수 f(n)은 각 노드에 적용된다. 노드들은 자신의 f 값의 순서에 따라 OPEN에 배치된다. 적은 f(n)의 값을 갖는 노드가 먼저 확장된다.

비용 함수 f의 선택에 따라 경로 탐색이 달라질 수 있으므로 다양한 탐색 알고리즘을 살펴 보자.

3.1.3.5탐욕적 탐색

탐욕적 탐색은 목표에 도달하기 위하여 소요 추정 비용 중 가장 낮은 노드를 확장된다.

추정 함수 f(n) = h(n) 여기서 경험 함수 h(n)은 목표까지의 남은 거리이다.

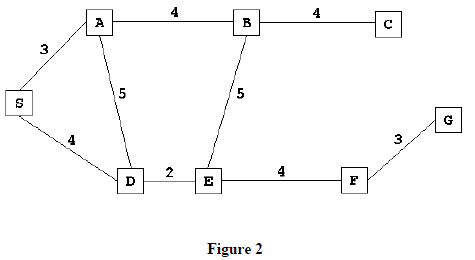
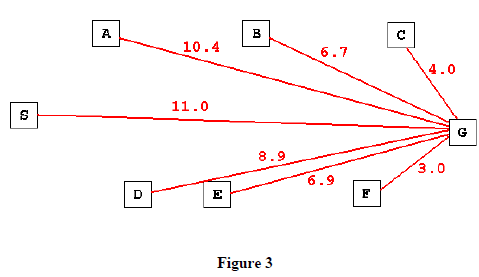
탐욕적 알고리즘은 때때로 목표에 도달하는 성능이 우수하여 항상 최적은 아니지만 양질의 해결책을 빨리 찾곤 한다.

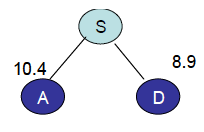
또한, 탐욕적 알고리즘은 최적도 아니며 불완전하여 존재하는 해결책의 탐색에 실패하는 경우도 있다. 이러한 경우를 다음의 예에서 볼 수 있다.

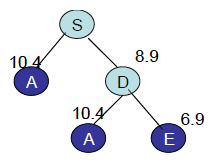
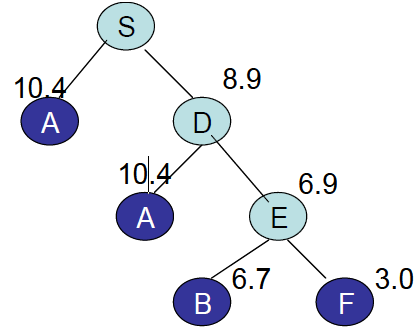
* 탐욕적 탐색

아래 좌측은 경로 탐색 문제의 예며 여기서는 탐욕적 탐색을 적용하기로 한다.

S는 시작 상태, G는 목표 상태이며 우측은 각 노드에서 목표까지의 직선거리이다. 경로 탐색 문제에서 도움이 되는 지식은 목표까지의 직선거리이다.

단계 1: 시작 상태인 S를 확장한다. 자식들은 A와 D이다.

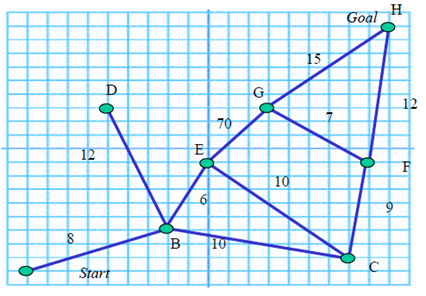
단계 2: 비용이 낮은 D를 확장한다.



3.1.3.6탐욕적 최고 우선 탐색

* 탐욕적 최고 우선 탐색

앞의 문제에 탐욕적 최고 우선 탐색 알고리즘을 적용하기로 한다.



경험 함수 h(n)은 n에서 목표까지 직선 거리로 한다. 노드의 확장은 A→B→E→G→H 순으로 된다. 획득 경로는 A-B-E-G-H이며 소요 비용은 99이다. 이는 최적의 경로는 아니다. 이 문제에서 경로 비용이 39인 A-B-C-F-H이 최적 경로임을 알 수 있다.

3.1.3.7 A\* 탐색

유명한 A\* 알고리즘을 다루기로 한다. 이 알고리즘은 1968 년 Hart, Nilsson & Rafael가 제안하였다.

A\*는 최고 우선 탐색 알고리즘이다.

평가 함수 f (n)은 경로 비용g(n)과 경험 함수h(n)의 합이며 여기서 g(n)은 시작에서 n까지의 소요 비용의 합계이며 h(n)은 n에서 목표까지의 추정된 최저 경로 비용이다.

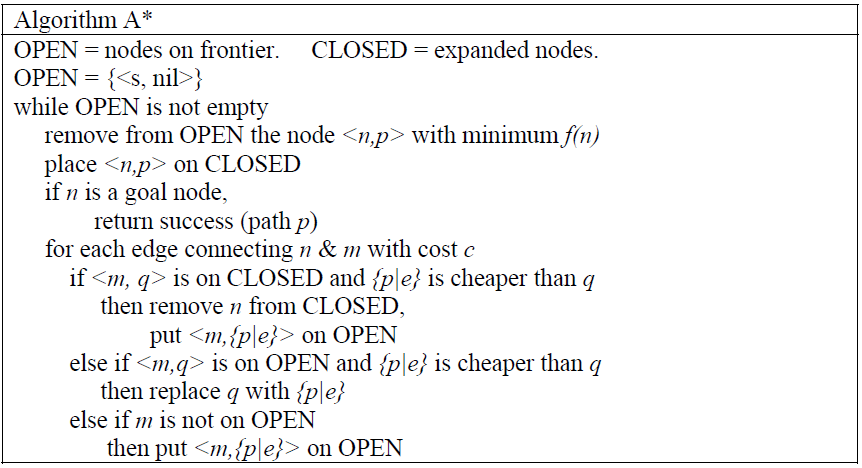
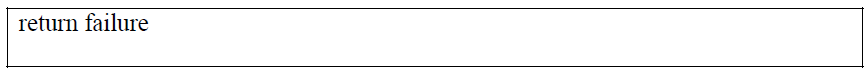
* 평가함수의 예

f (n) = 현재까지의 실제 거리 + 남은 추정 거리

h(n)은 n에서 목표에 도달 할 수 있는 해결책의 추정 비용이므로 임의의 해결책보다 낮게 추정되면 허용 가능이라고 한다. 만약 C\*(n)이 n에서 목표 노드까지 가장 저렴한 해결책의 경로 비용이고 경험 함수가 허용 가능이면, h(n) <= C \* (n).

h(n)이 허용 가능한 경우, 탐색이 최적의 솔루션을 찾는다는 것을 증명할 수 있다.

* A\* 알고리즘



3.1.3.8 .1A\* 알고리즘

A\* 알고리즘은 목표상태까지의 비용 추정값(일종의 휴리스틱 지식)을 이용하여 탐색의 효율화를 의도하는 탐색법이다.

A\* 알고리즘에서는 연산자를 적용해가면서 상태 공간 안을 탐색할 때 탐색 효율을 평가하기 위한 평가함수 f(n)을 도입한다. 여기서 f(n)은 아래와 같다.

f(n) = g(η) + h(n)

단, g(n): 시작 노드로부터 노드n까지 최적경로의 비용

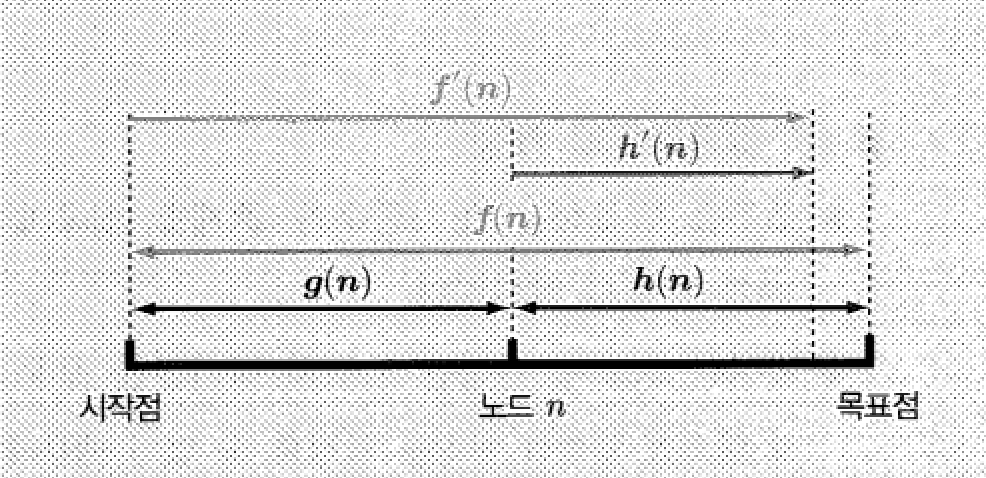
h(n): 노드n으로부터 목적지까지의 최적경로의 비용

탐색 도중에는 f(n) 의 값을 정확히 알 수 없기 때문에 f(n)을 추정한 값 f'(n)을 고려한다. 지금 노드 n까지 탐색했다고 하면, 시작 노드로부터 노드 n까지의 비용은 정확하게 알고 있기 때문에 아래와 같은 식이 성립한다.

f'(n): g(n) + h '(n)

단 h'(n): h(n) 의 추정값

평가 함수 f(n)과 f’(n)과의 관계는 다음과 같다.



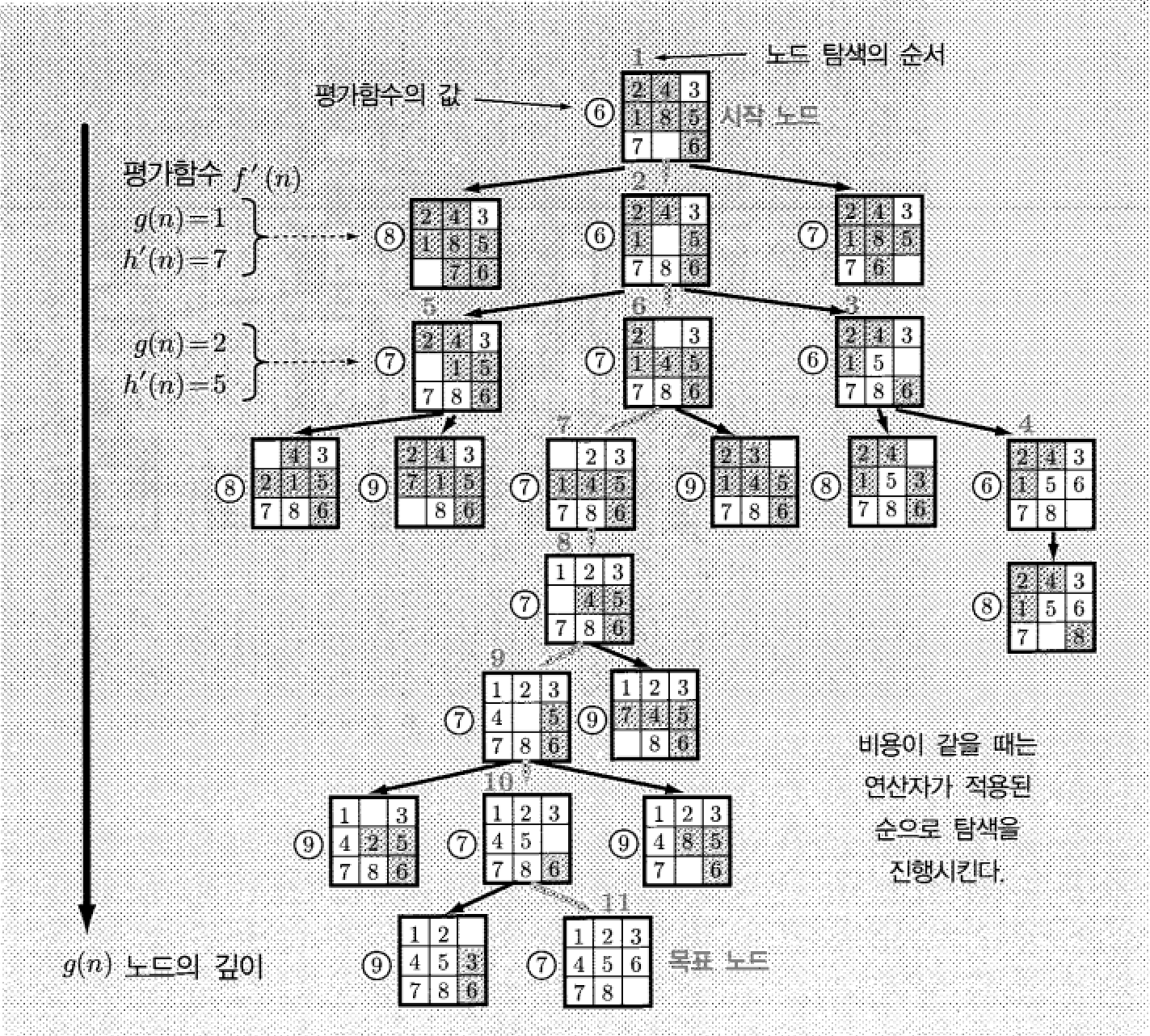
A\* 알고리즘에서는 h'(n) ≤ h(n) 의 관계가 성립할 때， 즉 노드 n으로부터 목적지까지의 비용 추정값 h'(n) 이 목적지로의 최적경로의 실제 비용 h(n) 보다도 작거나 같다면 반드시 최적해를 얻을 수 있음을 보증하고 있다. A\* 알고리즘의 절차는 최고 우선 탐색 알고리즘의 절차에서 추정 비용을 구할 때 평가함수를 조건 h'(n) ≤ h(n) 하에서 이용함으로써 나타낼 수 있다.

3.1.3.9 .2 A\* 탐색 예제: 8-퍼즐

8- 퍼즐에 A\* 알고리즘을 적용하기로 한다. 8- 퍼즐의 탐색공간을 트리 구조로 표현했을 때， 평가 함수 f (n)에 나타난 g(n)은 시작 노드에서 노드 n까지의 깊이가 된다. 지금 h'(n)을 “잘못된 위치에 있는 조각의 수”라고 하자. 이것은 잘못된 위치에 존재하는 조각과 최소한 같은 수의 조작을 거치지 않고서는 8- 퍼즐을 완성할 수 없다는 것을 의미하므로 h’(n) ≤ h(n) 을 만족한다. 이와 같이 평가함수 f'(n)을 설정한다.

이 평가함수를 이용하여 8-퍼즐을 탐색하는 예는 아래에 보였다.

원 안의 숫자는 평가함수 f'(n) 에서 얻어지는 값을 나타내고 있다. 또한 이 예의 경우 시작 노드에서 현재 노드까지의 비용 g(n)은 “탐색의 깊이”에 상당한다고 생각하고 있기 때문에 탐색 노드가 깊어짐에 따라 g(n) 의 값이 1씩 증가하고 있다. 추정 비용 h'(n)은 “잘못된 위치에 있는 조각(그림 에서 회색으로 표시되어 있는 조각)의 수”로 되어 있기 때문에 평가함수의 값은 “탐색의 깊이 +추정 비용”이 된다. 만약 추정값이 같은 경우에 연산자가 적용된 순으로 탐색을 진행한다고 가정하고 탐색을 수행한 결과를 나타내고 있다.



3.1.3.10 A\* 알고리즘의 특성

* 조건을 만족하는 평가 함수가 존재할 경우에는 최적 경로(최소 비용의 경로)가 반드시 얻어진다.
* 탐색한 노드에 인접한 노드의 평가값을 모두 조사하기 위해(평가값을 정하지 않으면 탐색의 방향을 결정할 수 없기 때문에) 대상이 되는 문제에 따라서는 그 나름의 탐색 비용이 들게 되는 경우가 있지만， 최적 경로를 발견할 수 있다는 것이 큰 이점이 된다.
* 미탐색 노드를 보존해 둘 필요가 있기 때문에 사용하는 메모리가 방대해져 버린다(아래 그림은 네 번째 노드를 탐색하고 있는 동안에 보존되어 있던 미탐색 노드를 나타내고 있다).



3.1.3.11 반복적 심화 A\* 알고리즘

반복적 심화 A\* 또는 IDA\*알고리즘은 깊이를 반복적 심화 깊이 우선 알고리즘과 유사하지만 다음과 같이 수정되었다.

깊이 제한이 f-limit로 수정되었다.

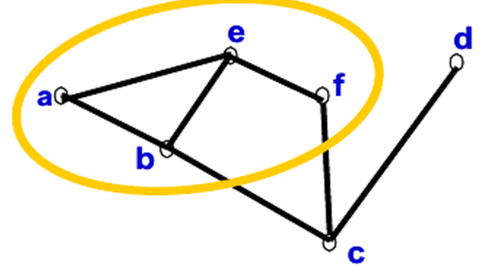
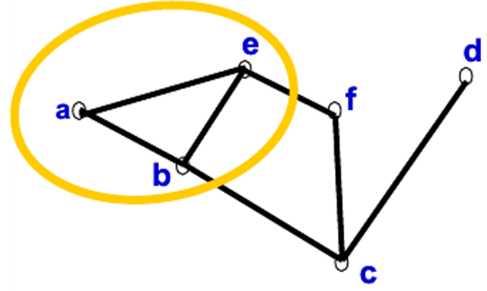
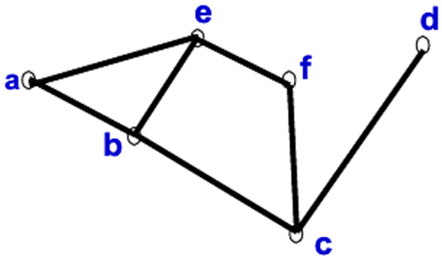
1. Start with limit = h(start)

2. Prune any node if f(node) > f-limit

3. Next f-limit=minimum cost of any node pruned

반복에서 확장 된 노드에 대한 컷오프는 노드의 f 값에 의해 결정된다. 그래프로 예를 들면 다음과 같다.

처음 반복은 노드a가 확장되고 자식 노드 b와 e가 생성된다. 평가 함수 f가 두 노드 다 15라면 다음의 반복은 f-제한은 15가 되어 a, b, e가 확장된다. 노드 e의 f값이21이라면 다음 반복은 f-제한이 21이 되어 노드 a, b, e, f 가 확장된다.



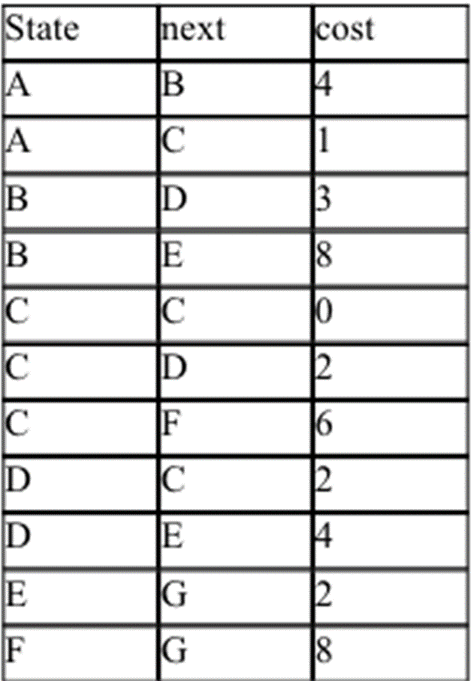
1회 반복(시작) 2회반복(f=15) 3회 반복(f=21)

3. 1. 3.12 문제

1. 시간과 공간의 복잡성과 관련하여 IDA \*와 A \*를 비교하라.

2. 언덕 등반이 n-여왕 문제의 해결책을 찾는 것을 보장하는가?

3. 다음과 같은 탐색 공간이 있다.



1. 주어진 탐색 공간의 상태 공간을 그려라
2. 초기 상태가 A이고 목표 상태가 G라고 가정합니다. 다음의 검색 전략 각각이 초기 상태에서 목표 상태까지의 경로를 찾기 위해 탐색 트리를 생성하여라.

균일 비용 탐색

탐욕적 검색

A\* 탐색

알고리즘의 각 단계에서 어떤 노드가 확장되고 있는지, 그리고 프린지의 내용을 보인다. 또한 각 알고리즘에서 발견되는 최종 해결책과 총 탐색 비용을 구한다.

3주차 2차시 교재 7장

학습 내용:

• 다중 에이전트 탐색의 형식화 및 2 에이전트 탐색을 상세하게 학습한다.

• 게임 트리를 이해한다.

• 2 에이전트 탐색 문제의 형식화를 학습한다.

• minimax 알고리즘을 학습한다.

• 경험적 스코어링 기능과 경험적 스코어 생성을 위한 표준 전략을 학습한다.

• 알파 베타 제거 알고리즘을 학습한다.

• 체스 프로그램을 분석하고 학습한다.

학습 목표:

• 주어진 문제를 분석하여 2 에이전트가 포함한 탐색 문제로 형식화 할 수 있다.

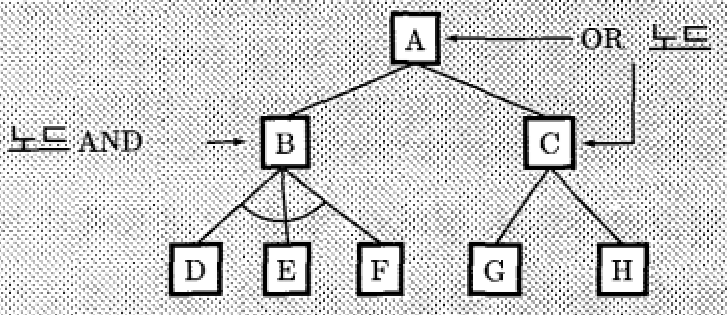
* 문제가 주어진 경우 2 에이전트가 포함된 탐색을 위한 가능한 전략을 적용하여 문제 해결 에이전트를 설계할 수 있다.

3. 2 적대적 탐색 (대항 탐색)

탐색 문제로서 2인 이상의 게임을 형식화하기 위한 프레임워크를 다루고 게임의 참가자들이 스코어링 함수를 최대 또는 최소화 위한 시도 또는 이동 등 행동을 하는 게임을 다루기로 한다. 문제를 간단화 하기 위하여 참가자들의 결탁은 없으며 항상 승자와 패자로 구분(제로 섬)되는 환경만을 고려한다.

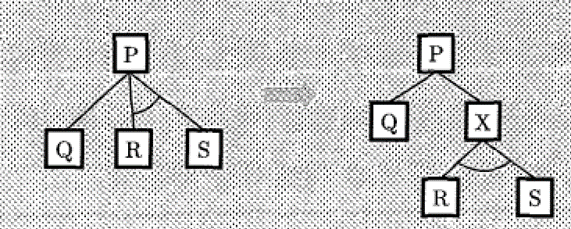
3.2.1 문제 분해

하나의 복잡한 문제를 해결하는 데에는 그 문제를 복수의 작은 문제로 분할하고, 그것을 하나씩 해결하여 최종적으로 원래의 복잡한 문제를 해결하는 것이 용이하다. 여기서 원래의 문제를 원문제 (original problem) 라고 하고， 원문제를 분할한 작은 문제를 하위 문제 (subproblem) 라고 한다. 원문제를 하위 문제로 분할함으로써 문제 전제가 해결 가능한가를 쉽게 판별할 수 있게 되는 동시에 해결하기 위한 전략을 발견하기 쉽다. 원문제는 하위 문제로 구성 되는 트리 구조에 의해 표현이 가능하다. 이 트리는 AND/OR 그래프(AND/OR graph) 라고 하여 다음과 같이 AND와 OR 노드로 구성된다.



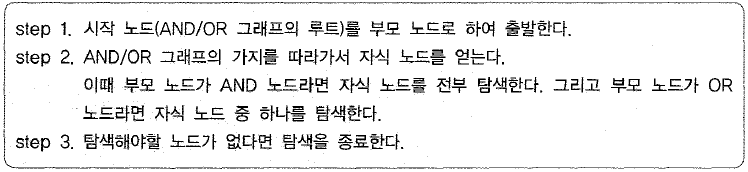
* AND 노드(AND node): AND로 연결되어 있는 복수의 하위 문제가 동시에 달성되지 않으면 원문제가 해결되지 않는다는 것을 나타내는 노드. AND를 가진 노드는 그 노드와 연결된 자식 노드를 원호로 묶어서 표시한다.
* OR 노드(OR node): OR로 연결되어 있는 복수의 하위 문제 중에서 어떤 하나가 달성되면 원 문제가 해결되는 것을 나타내는 노드.
* 터미널 노드(terminal node): AND/OR 그래프에서 그래프의 말단에 있는 노드는 터미널 노드라고 하여 해결 가능한 노드이다.

AND/OR 그래프에서 아래 그림의 왼쪽 그래프에 나타낸 것처럼 하나의 부모 노드에 대해 AND 노드와 OR 노드가 섞여있을 때는 오른쪽 그래프와 같이 보조 노드(그림에서는 노드 X)를 도입하여 그래프로 해석한다.

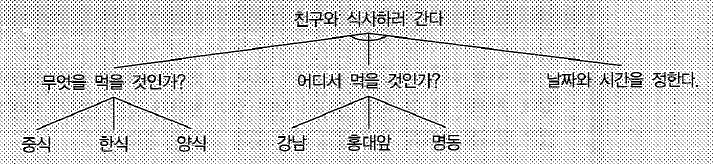


AND/OR 그래프로부터 실제로 문제-해결로써 선택된 하위 문제에 의해 구성된 그래프를 해 그래프(solution graph) 라고 한다.

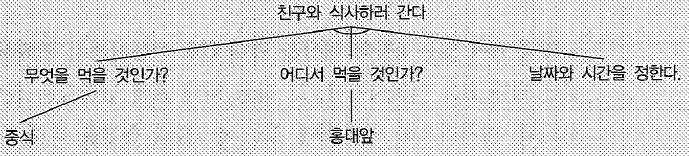
해 그래프는 다음과 같은 대략적인 방침에 의해 구할 수 있다.



이 알고리즘으로 얻어진 부분 그래프에서 터미널 노드가 모두 가능(해결가능)한 해라면, 얻어진 부분 그래프가 해 그래프가 된다. 예를 들어 “친구와 식사하러 간다”를 원 문제라고 한다면 이것은 아래에 나타낸 것과 같은 AND/OR그래프로 표현된다.



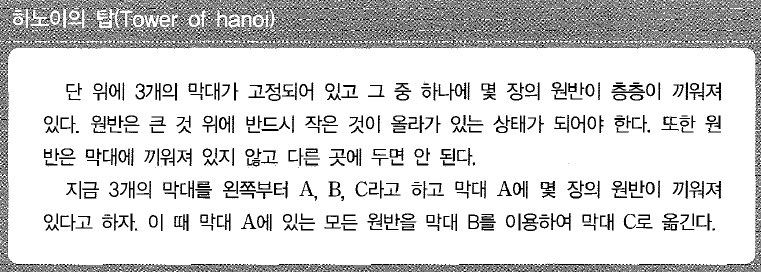
이 경우에 터미널 노드가 모두 가능한 해이기 때문에， 해 중의 하나로 아래와 같은 해 그래프를 얻을 수 있다.



해 그래프는 하나의 MU/OR 그래프로부터 복수 개가 얻어질 가능성이 있다. 원 문제의 해를 구하는 작업은 원 문제가 되는 노드로부터 출발하여, AND/OR그래프를 따라가서, 그래프의 분기점에서 AND 노드 및 OR 노드에 기반한 그래프의 추가에 의해 그래프를 성장시켜 나감으로써 해 그래프를 얻을 수 있게 되는 것이다.

3.2.2 문제의 재귀표현

문제를 분할했을 때 하위 문제가 재귀적인 요소를 포함하고 있는 경우가 존재한다. 이 경우, 소수의 하위 문제에 의해 원 문제가 해결될 가능성이 있다. 한 예로써， 1883년에 프랑스의 수학자 루카스(E. Lucas) 가 고안한 게임인 하노이의 탑(Tower of Hanoi) 이 있다.



막대 A에 있는 n장의 원반을 막대 C로 이동시키기 위해서는 n-1장의 원반을 일단 막대 B로 이동시킨다. 그 다음 막대 A에 남아있는 원반을 막대 C로 이동시키고， 마지막으로 막대 B에 있는 n-l장의 원반을 막대 C로 이동시키는 작업을 생각한다. 이 중에서 막대 A로부터 막대 B로 n-l장의 원반을 이동시키는 작업은 원반이 n-l장인 “하노이의 탑” 문제와 같으며 막대 B로부터 막대 C로 n-l장의 원반을 이동시키는 작업도 마찬가지로 원반이 n-l장인 “하노이의 탑”문제와 같다. 즉， 이 문제는 시간 순으로 아래와 같이 세 개의 하위 문제

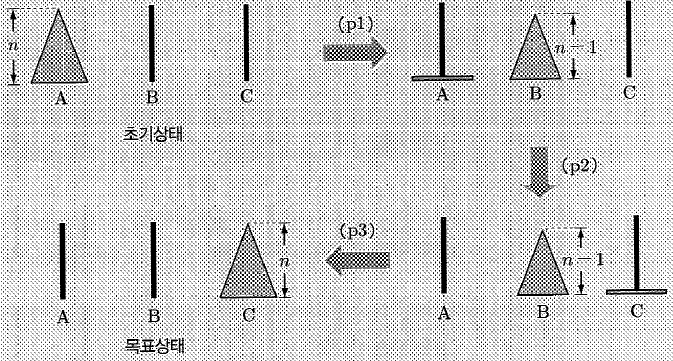
(p1)~(p3)로 분할되고， 그 중에서 두 개의 하위 문제 (p1)와 (p3)은 재귀문제로 되어 있다.

(p1) 막대 A로부터 막대 B로 n-l장의 원반을 막대 C를 이용하여 이동시킨다.

(p2) 막대 A로부터 막대 C로 막대 A에 있던 가장 큰 원반을 이동시킨다.

(p3) 막대 B로부터 막대 C로 n-l장의 원반을 막대 A를 이용하여 이동시킨다.

이를 기반으로 하는 문제 해결 흐름은 다음과 같다.



이를 가능하도록 2개의 연산자를 고안한다.

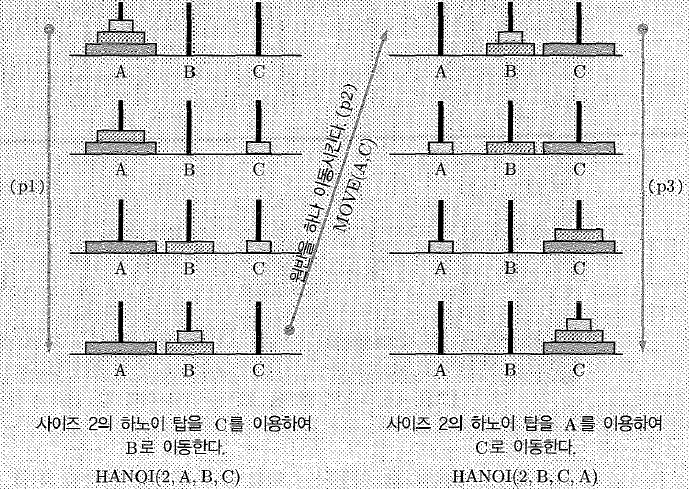
* HANOI(이동하는 원반의 수 n, 이동하는 원반이 있는 막대 x, 이동할 곳의 막대y, 이동할 때 이용하는 막대 z): n장의 원반을 막대 x로부터 z를 이용하여 막대 y로 이동하는 절차.
* MOVE(이동하는 원반이 있는 막대x, 이동할 곳의 막대y): 한 장의 원반을 막대 χ로부터 막대 y로 이동하는 절차.

이 두 개의 절차를 이용하여 n장의 원반에 대한 “하노이의 탑”문제를 하위 문제로 표현하면 다음과 같이 된다.

HANOI(n, A, C, B) = [HANOI(n-1, A, B, C) → MOVE(A， C) →HANOI (n-1, B, C, A)]

위의 표현에서 = (equal)의 좌변은 원 문제를 나타내고, 우변은 원 문제를 구성하는 하위 문제군을 나타내고 있으며 하위 문제는→(화살표)에 의해 적용되는 시간 순서와 함께 나타나 있다.

여기서 원반의 수가 3장인 “하노이의 탑” 문제의 해결의 흐름도는 아래에 같다.



하노이의 탑을 비재귀 프로그램으로 풀려고 하면 너무 어렵게 되지만 재귀를 이용함으로써 매우 쉽게 된다.

3.2.2.1 하노이 탑의 이동 횟수

하노이의 탑에서 원반의 이동 횟수를 생각해보자. n장의 원반이 어느 한 막대에 존재할 때의 이동회수를 Hanoi(n)으로 표시한다. 이 때 아래의 조건이 성립한다.

* 원반이 1장인 경우: Hanoi(1) = 1
* 원반이n>1 장인 경우: Hanoi(n) = Hanoi(n-1) + 1 + Hanoi(n-1)

이것을 점화식으로 정리하면 아래와 같이 기술할 수 있다.

n=1의 경우, Hanoi(1) = 1

n>1의 경우, Hanoi(n) + 1 = 2(Hanoi(n-1) + 1)

이 점화식을 풀면,

Hanoi(n) =2n-1

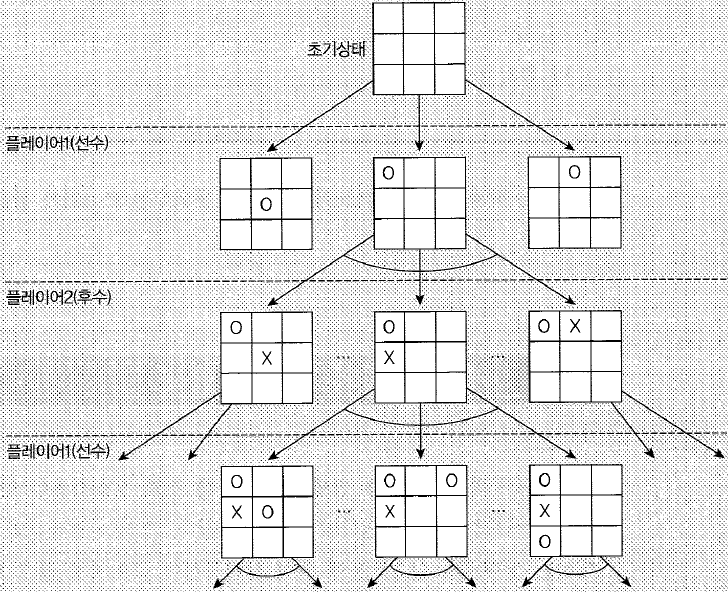
이 된다. 따라서 n장의 하노이의 탑의 이동회수는 2n-1 회가 됨을 알 수 있다.

3.2.3 게임 문제의 표현과 탐색

지금까지의 문제-해결 방법에서는 다른 것과 상호작용이 없는 상태에서 최적해를 탐색하는 방법에 관하여 논의하여 왔다. 그러나 컴퓨터와 대결하는 게임 등에서는 상대와의 상호작용에 의해 상태공간이 끊어지지 않고 변화하며, 그 속에서 상대의 행동을 예측하면서 최적의 해를 발견할 필요가 있다. 그와 같은 해를 탐색하기 위하여 게임 트리(game tree)라고 하는 AND/OR 그래프가 많이 사용된다.

게임 트리는 두 명의 플레이어가 교대로 플레이하여 게임을 진행해가는 천이 상태를 트리 구조로 표현한 것으로써 AND/OR 그래프로 구성된 탐색 트리가 된다. 선수 입장에 서면 선수 차례의 국면에서는 가능한 몇 가지의 수 중에 하나만을 실제로 두는 수로 선택하게 된다. 그러므로 선수 국변에서의 실제 두는 수는 OR 노드에 의한 가지가 갈라진다. 한편 후수의 국면에서도 몇 가지 수가 가능한데, 그 중에 어느 것이 실현되는지는 후수에 의해 결정되므로 선수는 모든 수에 대한 대응책을 생각해야 한다. 따라서 후수 차례의 국면은 AND노드에 의해 가지가 갈라진다.

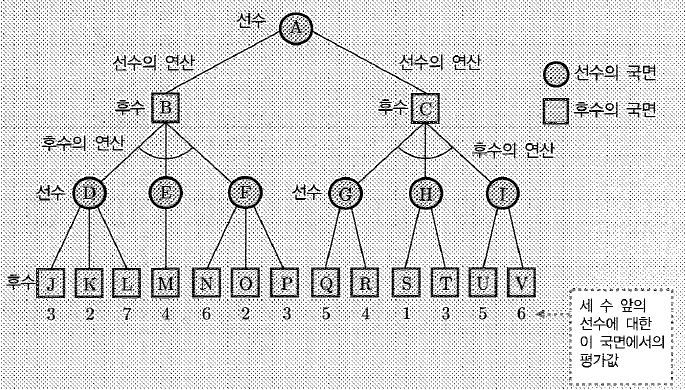
아래에 삼목 게임 (Tic-Tac-Toe) 의 게임 트리의 일부를 나타낸다.



3.3.1 최소-최대 알고리즘

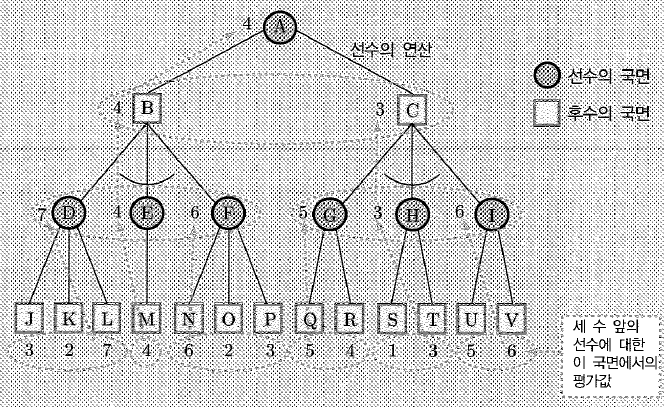
앞서 기술한 것처럼 게임 문제는 상대방과의 상호작용에 의해 상태공간이 변화되고, 상태공간은 일반적으로 AND/OR 그래프로 표현된다. 게임 대결에서 플레이어의 유리/불리를, 상태공간에서 하나의 노드가 되는 게임 형세의 상태에 의해 평가할 수 있다고 하자. 대결에서는 선수가 수를 둘 때는 선수 쪽이 유리한 형세의 상태가 되도록, 또한 후수가 수를 둘 때는 후수 쪽이 유리한 형세의 상태가 되도록 수를 두는 것이 쉽게 예측된다.

가령, 선수에 대한 세 수 앞의 형세에 대한 평가값이 다음 그림과 같이 같다고 하자. 그림에 나타난 세 수 앞의 형세의 평가값에 따르면, 평가값이 가장 큰 L에 도달하는 순서인 A → B → D → L과 같이 진행해 가면 선수에 대해 가장 유리한 전개가 된다(숫자가 클수록 유리). 그러나 앞에서 말한 것처럼 선수와 후수가 교대로 자신들에게 유리한 형세가 되도록 수를 두게 하면 반드시 선수나 후수 중 한쪽에 유리한 전개로는 되지 않는다. 세 수 앞의 평가값이 어떻게 선택되어 가는지는 평가값을 선수 순서의 상태 A까지 bottom-up으로 전파해감으로써 예상할 수 있다.



세 수 앞의 평가값은 선수 쪽에서 본 형세의 평가값이다. 예를 들어 형세 J, K, L에서 평가값이 3, 2, 7이 될 때 형세 D에서 선수는 자신이 유리하게 되는 형세 L이 되도록 수를 둔다. 그 때문에 형세 D의 평가값은 7이 된다.

마찬가지로 아래 그림에 나타낸 것처럼 형세 D, E, F, G, H, I에 대해 선수가 얻을 수 있는 평가값은 각각 7, 4, 6, 5, 3, 6이 된다. 그 다음 형세 B를 보면, 후수는 선수가 불리하게 되도록 형세를 변화시킨다고 생각할 수 있기 때문에 후수는 형세E가 되도록 수를 둔다고 예측할 수 있다. 같은 방식으로 후수가 형세 C의 상태에서 형세 H가 되도록 다음 한 수를 둔다고 예측할 수 있다. 선수는 형세 A에서 자신이 유리하게 되도록 수를 두게 되면 형세 B가 되도록 수를 둔다고 생각되기 때문에， 결국 선수와 후수가 서로 유리하게 되도록 수를 두었다고 할 때의 세 수 앞까지의 순서는 A → B → E → M예측 이 된다.

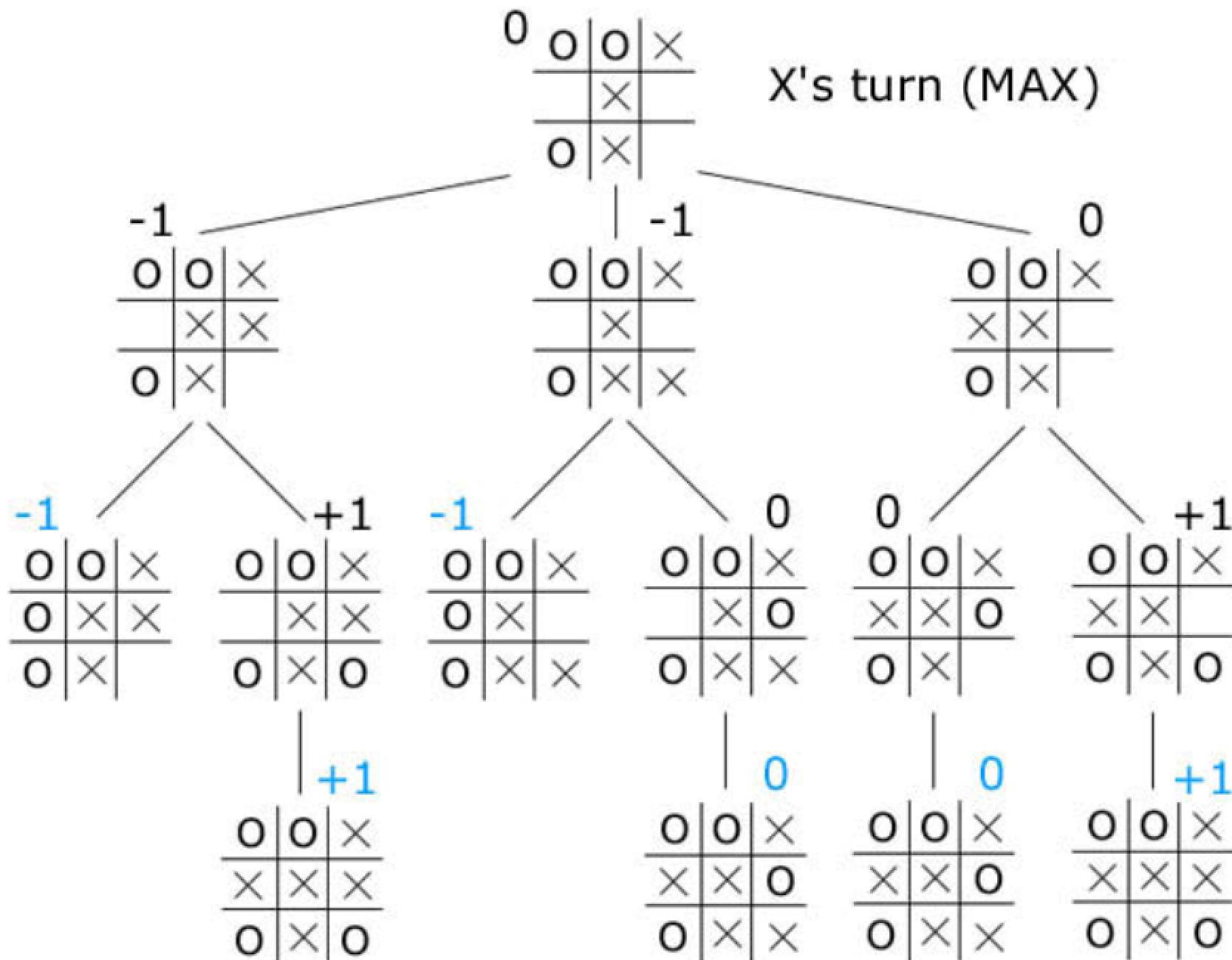


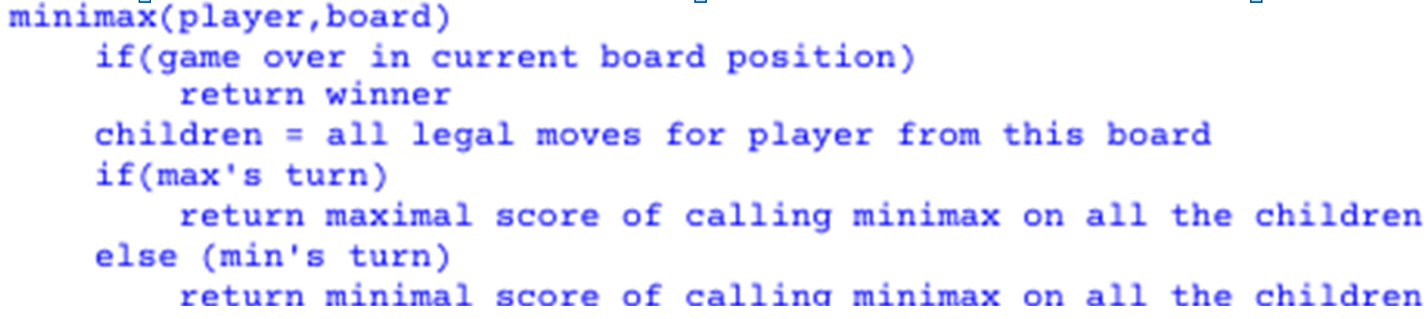
이와 같이 게임 트리에서 몇 수 앞의 평가값이 명확할 때는 자신의 차례일 경우에는 평가값이 최대가 되도록 bottom-up으로 전파시키고， 상대방의 차례에서는 자신에 대한 평가값이 최소가 되도록 평가값을 bottom-up으로 전파시킴으로써 자신과 상대방의 상호 득실을 고려한 게임 진행 방법을 실현한다. 이 방법은 형세의 평가값의 최대값과 최소값이 교대로 취해지기 때문에 mini-max법이라고한다.

일반적으로 평가값의 결정에 대해서는 각 게임의 특성이 고려된 다음에 이루어진다. 장기는 각 말의 자유도나 가진 말의 수 왕의 안전도 등을 고려하여 평가값이 정의되며, 바둑에 대해서는 살아있는 돌의 개수나 확정된 집의 크기와 두께 등이 고려되어 평가값이 정의된다.

또한， 주의해 둘 것은 mini-max법에서는 미리 몇 수 앞까지 계산하는가를 결정한 다음 선수에게 유리한 게임 진행 방법을 탐색한다는 것이다. 이로 인해 몇 수 앞까지 계산하는가에 의해 선수의 게임 진행 방법도 당연히 바뀌게 된다. 일반적으로 수읽기가 가능한 수의 양에 의해 게임 능력이 정해지지만, 그러기 위해서는 게임 트리의 깊이에 따라 탐색에 걸리는 시간과 메모리가 필요하게 된다. 그래서 짧은 시간과 적은 메모리로 탐색을 수행하는 여러 가지 효율적인 게임 트리 탐색 방법이 고안되고 있다. 그 중 대표적인 것이 다음에 설명하는 α-ß자르기이다.

3.3.1.1 삼목게임(Tic-Tac-Toe): 최소-최대 알고리즘의 예



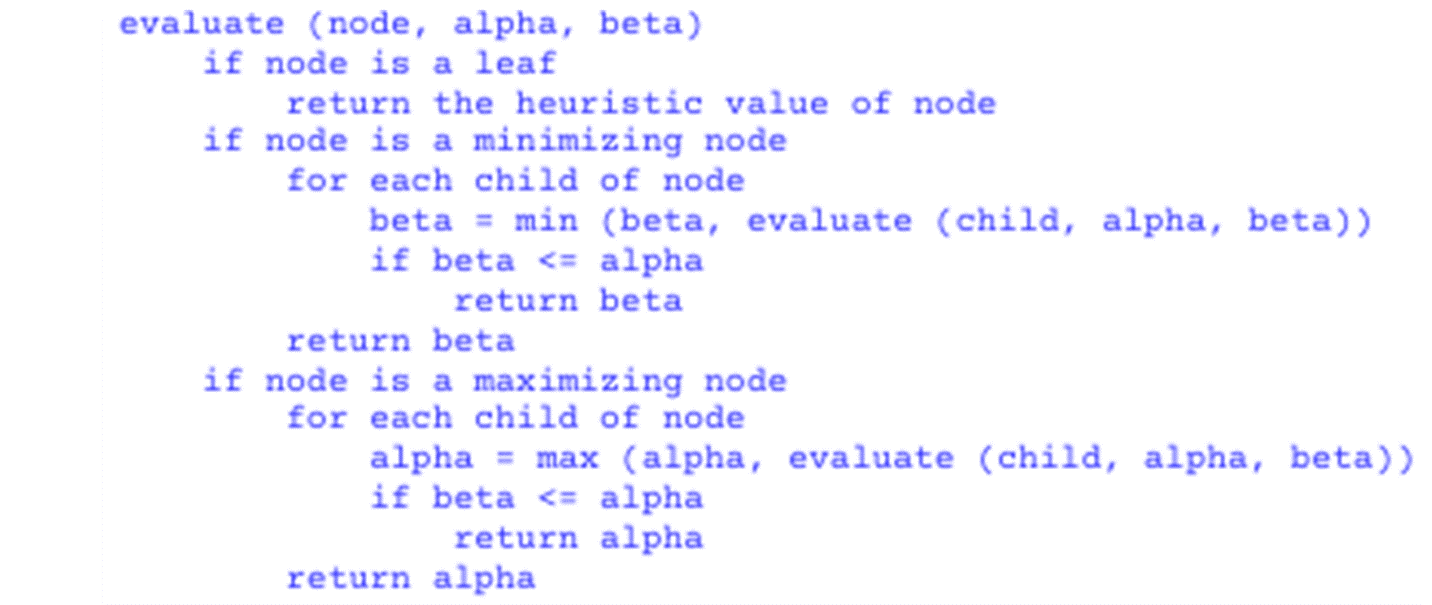


위의 게임 범주는 노드가 게임의 현재 상태를 나타내고 호가 동작을 나타내는 트리로 표시 되였다. 게임 트리는 루트에서 시작하는 현재 플레이어의 가능한 모든 이동과 이러한 노드의 하위 노드 인 다음 플레이어의 가능한 모든 이동 등으로 구성되어 있다. 게임 트리의 리프은 게임의 결과가 명확한 (승, 패, 무승부) 터미널 위치를 나타낸다. 각 터미널 위치에는 점수가 있다. 높은 점수는 MAX 플레이어라고 불리는 선수에게 좋다. MIN 플레이어라고 불리는 후수는 점수를 최소화하려고 시도한다. 예를 들어 1을 승리로, 0을 무승부로, -1을 선수에 대해 손실로 연결할 수 있다.

3.3.2 α-ß자르기

mini-max법에서는 형세의 국변을 미리 읽을 수 있으면 있을수록 좋은 수를 선택하여 게임을 유리하게 진행시켜 나갈 수 있다. 그러나 그 반면 수읽기를 너무 많이 하면 게임 트리의 탐색 공간이 방대해져서 평가값을 구하는 계산의 비용이 커지게 되는 문제점이 있다. 그래서 평가값의 최대값과 최소값을 교대로 취하는 min-max법의 특정을 살려서 게임 트리의 탐색에서 불필요한 탐색을 피함으로써 탐색의 효율화를 꾀하는 방법이 제안되었다. 그것을 α-ß 자르기 (α–ß pruning) 이라고 한다.

α -ß 자르기에 의해 게임 트리를 탐색할 때는 통상 종형 탐색이 수행된다. 종형탐색 방법 하에서 mini-max법에 의해 최대값 최소값이 교대로 취해지는 특성을 살려서 게임 트리의 탐색공간을 제한한다. 이것은 일반적으로 가지치기 (pruning)라고 한다. α -ß 자르기 알고리즘은 다음과 같다.



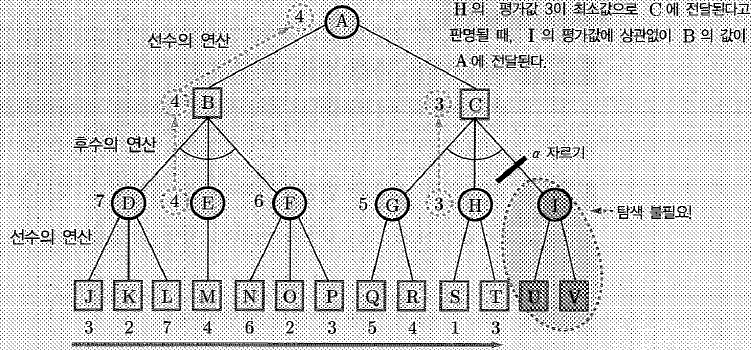
3.3.2.1 ß자르기(ß pruning)

ß 자르기는 후수의 수 두기 에 의한 평가값의 최소화 국면에서의 가지치기를 말한다. 아래 그림에서 종형탐색으로 좌측 끝부터 탐색 된다고 하면, 선수의 두 번째 수에 대해서는 형세 J, K, L, M, N의 순서로 각각 평가값이 조사되어 형세 D, K, F에 그 평가값이 전파되어 간다. 형세 B에서 후수의 수에 의한 평가값을 최소화하는 국면에서는 형세D, E, F의 평가값 중에서 최소값이 그 평가값이 된다. 여기서 그림을 보면 형세 M까지 탐색했을 때 형세 D와 형세 E의 평가값을 비교하여 형세 E의 평가값 4가 형세 B에 전파되는 값의 후보가 된다는 것을 알 수 있다. 다음으로 형세 N을 탐색했을 때 평가값이 6으로 판명되어 형세 F에는 형세 N, O , P의 평가값 중 최대값이 전파된다는 것으로부터 그 시점에서 최종적인 형세 F의 평가값은 6 이상이라는 것이 확정된다. 형세 B에는 형세 D, E, F 중 최소 평가값이 전파되므로 형세 O, P에 대한 탐색을 하지 않아도 형세 B로의 평가값 전파는 형세 E로부터의 평가값인 4가 된다는 것을 알 수 있다. 이 때문에 형세 O. P로의 무의미한 탐색을 수행하는 것을 피하여(즉， 가지치기를 행하여) 탐색의 효율화를 꾀한다.



3.3.2.2 α자르기(α pruning)

α 자르기는 선수가 두는 수에 의한 평가값의 최대화 국면에서의 가지치기를 말한다. 아래 그림에서 선수의 두 번째 수에서는 형세 J, K, L, M, N, Q, R, S, T의 차례로 각각 평가값이 조사되어(형세 O, P는 앞서 기술한 ß자르가에 의해 가지치기됨 것으로 간주한다) 형세D, E, F, G, H에 그 평가값이 전파되어 간다. 형세 A에서 선수의 수에 의한 평가값을 최대화하는 국면에서는 형세 B, C의 평가값의 최대값이 그 평가값이 된다. 여기서 그림을 보면 형세 T까지 탐색했을 때 형세 G와 형세 H의 평가값을 비교하여 형세 H의 평가값 3 이 형세 C에 전파되는 값의 후보가 되는 것을 알 수 있다. 여기에서 형세 U, V를 탐색하여 형세 I의 평가값을 구했다고 하더라도 형세 C에는 값이 작은 쪽이 전파되기 때문에 그 시점에서 최종적인 형세 C의 평가값은 3 이하의 값으로 된다는 것이 확정되어 있음을 알 수 있다. 형세 A에는 형세 B와 형세 C의 평가값 중 큰 쪽이 그 평가값으로 되기 때문에 형세 B의 평가값보다 형세 C의 평가값이 작다는 것이 판명된 시점에서 그 이상의 탐색은 불필요하다. 그래서 형세U, V로의 탐색은 수행하지 않고, 이어서 형세 I의 평가값도 구할 필요가 없게 되어 형세 U, V, I로의 탐색을 수행하지 않음으로써 탐색의 효율화를 꾀한다.



3.4 최고 수준의 게임 플레이 프로그램들

3.4.1 체스

IBM의 Deep Blue 체스 프로그램은 널리 홍보된 시범 경기에서 세계 챔피언 개리 카스파로프(Garry Kasparov)를 이긴 것으로 유명하다. Deep Blue는 30개의 IBM RS/6000 프로세서를 갖춘 병렬 컴퓨터에서 알파 베타 검색을 수행했다. 독특한 부분은, 480개의 커스텀 VLSI 체스 프로세서들로 구성된 회로이다. 이 프로세서들은 트리의 마지막 몇 수준의 수들의 생성과 순서 결정, 그리고 잎 노드들의 평가에 쓰였다. Deep Blue는 한 수당 최대 300억 개의 국면을 검색했다. 일상적으로 14 수준의 검색을 수행한 것이다. 성공의 핵심 요인은 충분히 흥미로운 강제하는/강제된 수들의 경로에 대해 깊이 한계를 넘는 특이 도구들을 생성해 내는 능력이었다. 평가 함수에는 8000개가 넘는 자질들이 포함되었으며 시스템은 약 4000개의 국면을 담은 ‘시작 수 모음집(opening book)’을 사용했으며, 70만 개의 그랜드 마스터 급 시합 자료를 담은 데이터베이스를 참조했다. 이 데이터베이스는 유효 검색 깊이를 크게 확장하는 효과를 냈으며, 그 덕분에 Deep Blue는 체크 메이트까지 여러 수가 필요한 몇몇 상황에서도 완벽하게 플레이할 수 있었다. 2008, 2009 세계 컴퓨터 체스 챔피언십 우승자인 Rybka는 현재 가장 강한 컴퓨터 플레이어로 간주된다. 이 시스템은 소비자용 8코어 3.2GHz Intel Xeon 프로세서를 사용한다. 그러나 프로그램의 설계에 대해서는 알려진 바가 거의 없다. Rybka의 주된 장점은 평가 함수인 것으로 보이는데, 그 평가 함수는 주 개발자인 국제 체스 마스터 바식 라일릭Vasik Rajlich과 적어도 세 명의 다른 그랜드 마스터들이 조율했다. 가장 최근 시합을 보면 최상급 체스 프로그램들이 모든 인간 경쟁자보다 훨씬 앞서 있는 것으로 보인다.

3.4.2 체커

조너선 섀퍼Jonathan Schaeffer와 동료들이 개발한 Chinook는 보통의 PC에서 실행되며, 알파 베타 검색을 사용한다. Chinook는 오랫동안 챔피언 자리를 유지한 인간 플레이어를 1990년의 한 약식 경기에서 이겼다. 그리고 2007년부터 Chinook는 알파 베타 검색과 39조 개의 끝내기 국면들을 담은 데이터베이스의 조합을 이용해서 완벽한 플레이를 해 왔다. 오델로Othello(또는 오셀로)는 보드 게임보다는 컴퓨터 게임으로 더 유명할 것이다. 이 게임의 검색 공간은 체스의 것보다 작다. 적법한 수는 보통 5~15개이다. 그러나 평가에 관한 전문 지식을 무에서 개발해야 했다. 1997년 Logistello 프로그램(Buro, 2002)은 인간 세계 챔피언 다케시 무라카미Takeshi Murakami를 6전 전승으로 물리쳤다. 오델로에서는 인간이 컴퓨터를 이길 수 없음을 인정하는 사람들이 많다.

3.4.3 백개먼

주사위 굴림에 의한 불확실성 때문에 깊은 검색이 값비싼 사치가 되는 이유를 설명했다. 백개먼에 대한 대부분의 연구는 평가 함수의 개선에 초점을 두었다. 테사우로는 강화 학습과 신경망을 결합해서 놀랄 만큼 정확한 평가 함수를 개발했으며, 그것을 이용해서 깊이가 2 또는 3인 검색을 수행했다(Tesauro, 1992). 자신을 상대로 한 백만 회의 연습 시합을 마친 테사우로의 프로그램 TD-Gammon은 최상급 인간 플레이어와 경쟁할 수 있는 수준이다.

3.4.5 알파고(바둑)

바둑판이 19×19이고 (거의) 모든 빈칸에 돌을 놓을 수 있으므로 분기 계수는 361에서 시작한다. 이는 보통의 알파베타 검색 방법으로는 감당할 수 없는 크기이다. 또한 평가 함수를 작성하기 힘든데, 이는 끝내기 단계에 도달하기 전에는 영역의 지배권을 예측하기가 아주 어렵기 때문이다. 그래서MoGo 같은 최상급 프로그램들은 알파베타 검색을 피하고 대신 몬테카를로 롤아웃 기법을 사용한다. 여기서 관건은 롤아웃 과정에서 어떤 수를 둘지 결정하는 것이다. 공격적인 가지치기는 없다. 모든 수가 가능하다. UTC(upper confidence bounds on tree; 트리에 대한 확신도 상계) 방법은 처음 몇 반복에서 무작위로 수를 두고 시간이 지남에 따라 이전 표본들에서 승리로 이어지는 수들을 선호하도록 표본 선택 과정을 조정하는 식으로 진행된다. 여기에 지식 기반 규칙(주어진 패턴이 검출될 때마다 특정 수들을 제시하는)이나 제한 국소 검색(전술적 질문에 대한 답을 얻기 위한) 같은 요령들을 곁들인다. 끝내기(endgame) 분석을 위해 조합적 게임 이론(combinatorial game theory)의 특별한 기법들을 동원하는 프로그램들도 있다. 그런 기법들은 하나의 국면을 개별적으로 분석해서 다시 결합할 수 있는 부분 국면들로 분해한다(Berlekamp 및 Wolfe, 1994; Müller, 2003). 이런 방식으로 얻은 최적해들은 자신이 지금까지 최적으로 플레이했다고 생각한 여러 프로 기사를 놀라게 했다. 2007년 모고는 9X13 축소 바둑판 경기에서 유럽 순위 300등을 기록하여 컴퓨터가 바둑게임에서 사람을 이길 수 있다는 가능성을 열었다. 모고의 바둑실력 핵심은 몬테카를로 방법이라는 통계적 샘플링 알고리즘이다. 이는 1940년대 원자폭탄을 만들기 위한 맨하튼 프로젝트에서 개발된 알고리즘이다. 2013년 이후 몬테카를로알고리즘을 적용한 ‘크레이지 스톤’, ‘젠’, ‘파치’ 등의 바둑 프로그램이 개발되었고 최근의 알파고가 현존하는 가장 진화되고 지능적인 프로그램이다. 이는 몬테카를로 알고리즘과 딥러닝 알고리즘을 추가해 사람처럼 스스로 바둑을 학습하는 능력이 있다. 학습을 위하여 인공 신경망이 필요하며 알파고는 다음 수를 두기 위한 정책망이 후보 수들을 제시하고 가치망이 최적의 수를 결정하는 2개의 인공신경망을 갖는다. 또한, 정책망은 강화 학습을 하여 스스로 똑똑해지는 것이 특징이다.

3.5 요약

이번 장에서는 최적의 플레이가 뜻하는 바가 무엇이고 실제 게임을 잘 플레이하는 방법이 무엇인지 이해하기 위해 다양한 게임들을 살펴보았다. 가장 중요한 개념들은 다음과 같다.

* 하나의 게임을 초기 상태(게임판의 구성), 각 상태에서의 적법한 동작들, 각 동작의 결과, 말단 판정(게임이 끝났음을 말해 주는), 그리고 말단 상태들에 적용되는 효용 함수로 정의할 수 있다.
* 완전한 정보가 제공되는 2인용 영합 게임에서 최소 최대 알고리즘은 게임 트리의 깊이 우선 열거를 통해서 최적의 수들을 선택할 수 있다.
* 알파 베타 검색 알고리즘은 최소 최대와 동일한 최적의 수들을 계산하나, 결과와 무관함이 확실한 부분 트리들을 제거하기 때문에 훨씬 더 효율적이다.
* 일반적으로 게임 트리 전체를 고려하는 것은 비현실적이다(알파베타에서도). 따라서 특정 지점에서 검색을 차단하고 상태의 효용을 평가하는 경험적 평가 함수를 적용할 필요가 있다.
* 여러 게임 플레이 프로그램들은 게임 개시 및 끝내기에서의 최선의 수들을 담은 표를 미리 계산해 두고, 검색 대신 표 참조로 수를 찾는다.
* 우연이 관여하는 게임은 우연 노드를 평가하도록 최소 최대 알고리즘을 확장해서 처리할 수 있다. 우연 노드는 모든 자식 노드 효용들의 가중 평균(각 자식의 확률을 가중치로 적용)을 구해서 평가한다.
* 크리그슈필이나 브리지처럼 불완전한 정보를 가진 게임의 최적 플레이를 위해서는 각 플레이어의 현재와 미래의 믿음 상태를 추론할 필요가 있다. 간단한 근사 방법은, 누락된 정보의 모든 가능한 구성 각각에 대한 동작의 값의 평균을 구하는 것이다.
* 체스나 체커, 오델로 같은 게임에서는 컴퓨터 프로그램이 사람을, 그것도 세계 챔피언을 이긴 바 있다. 포커나 브리지, 크리그슈필처럼 정보가 불완전한 게임이나 바둑처럼 분기 계수가 아주 크고 좋은 경험적 지식이 거의 알려지지 않은 게임에서는 아직 사람이 우월하다.

3.6 문제

1. Tic-tac-toe 게임을 한다. x가 MAX 플레이어(선수)라고 가정합니다. X의 승리 유용 함수값(utility function)을 10, X의 패배 유용함수값은 -10, 그리고 무승부는 0이라 한다.

a) 아래에 있는 게임 보드 1에서 다음에 플레이 할 X의 차례가 주어지면 전체 게임 트리를 표시하라. 각 터미널 상태의 유틸리티를 표시하고 minimax 알고리즘을 사용하여 최적의 이동을 계산하라.

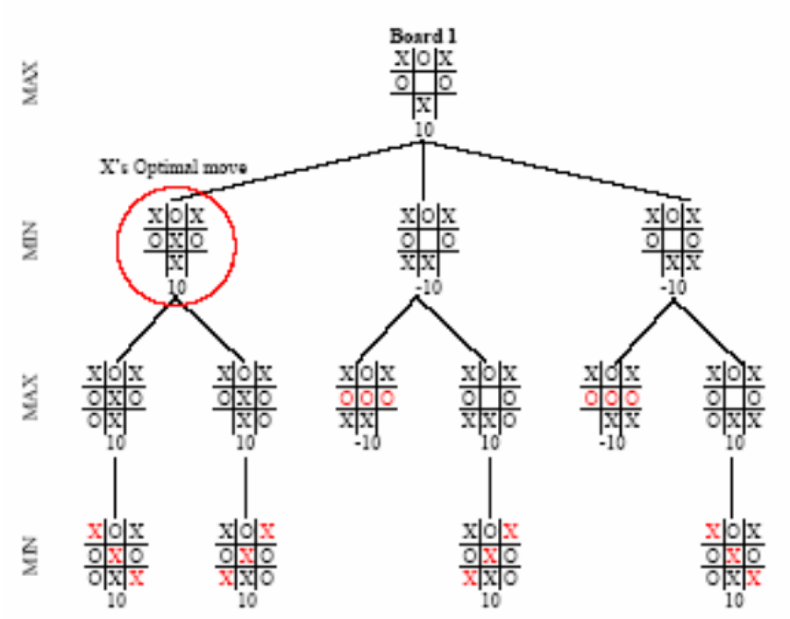
b) 아래에 있는 게임 보드 2에서 X가 다음에 플레이 할 차례 인 경우, 게임 트리의 컷오프는 2ply(선수 후수 한번씩: 즉, 각 플레이어가 한 번 이동 한 후 중지한다). 모든 리프 노드에 대해 다음 평가 함수를 사용하라.

Eval(s) = 10X3(s) + 3X2(s) + X1(s) – (10O3(s) + 3O2(s) + O1(s))

여기서 Xn(s)은 s 상태에서 O가 없고 X가 n개 있을 때 행, 열, 또는 대각선의 번호로 정의하고 동일하게 On(s)을 상태 s에서 X가 없고 O가 n개 있을 때 행, 열, 또는 대각선의 번호로 정의한다. X의 형세가 가장 유리하도록 결정하기 위하여 최소 최대 알고리즘을 사용하라.

3.6.1 풀이

1. a) 보드 1의 게임 트리는 아래와 같다.



1. b) 보드 2의 풀이는 아래와 같다.