

8. 게임 알고리즘

한국외국어대학교
고 석 훈

목차

8.1 Min-Max 알고리즘

8.2 Alpha-Beta 프루닝

8.3 A* 알고리즘

인공지능(AI, Artificial Intelligence) 개론

- 다양한 인공지능 응용분야
 - 컴퓨터 비전, 자연어 처리, 음성인식, 자율주행 자동차 등
- 게임은 가장 흥미로운 인공지능의 응용분야
 - 도전적으로 덤비는 적과 도움을 주는 아군 캐릭터
- 인간과 비슷한 수준의 인공지능을 만드는 것은 매우 어렵지만 '제한된 분야'에서는 매우 잘 동작함
 - 체스의 Deep Blue, 바둑의 AlphaGo 등

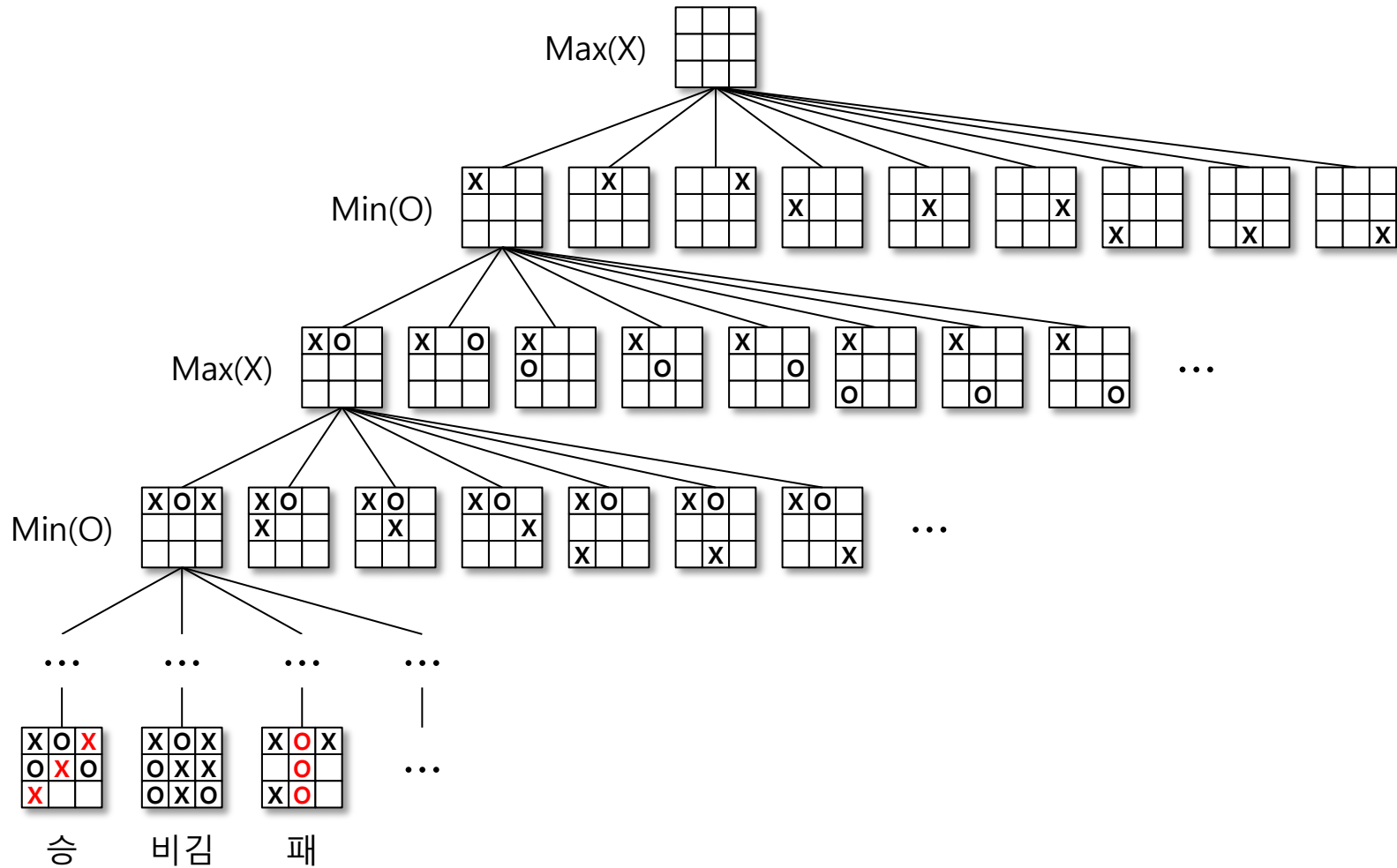
8.1 Minmax 알고리즘

- Minmax 알고리즘
 - 두명의 플레이어가 참여하는 턴방식 게임에서 가능한 모든 수에 대한 트리(Minmax Game Tree)를 만들어 질 확률이 가장 낮으면서 이길 확률이 가장 높은 수를 선택하는 방법
 - 거의 대부분의 컴퓨터 보드게임에서 사용
- Minmax 알고리즘을 적용할 수 있는 게임의 특징
 - 보드 위의 모든 정보를 확인할 수 있는 턴 방식 게임
 - ◆ 예) 틱-택-토, 오목, 오델로, 장기, 체스, 바둑 등
 - 모든 정보를 확인할 수 없는 게임에서는 사용할 수 없음
 - ◆ 예) 포커, 주사위 게임 등

Minmax Game Tree

- Minmax Game Tree
 - 각 노드는 게임의 상태를 의미
 - 루트에서 시작하여, 한 수를 둔 상태에 대해 자식 노드를 생성한다.
 - 둘 수 있는 수의 종류만큼 자식 노드가 만들어진다.
 - 세대를 내려가면서 컴퓨터의 턴과 상대의 턴이 반복된다.
 - 게임의 승패가 결정되는 노드가 단말노드가 된다.

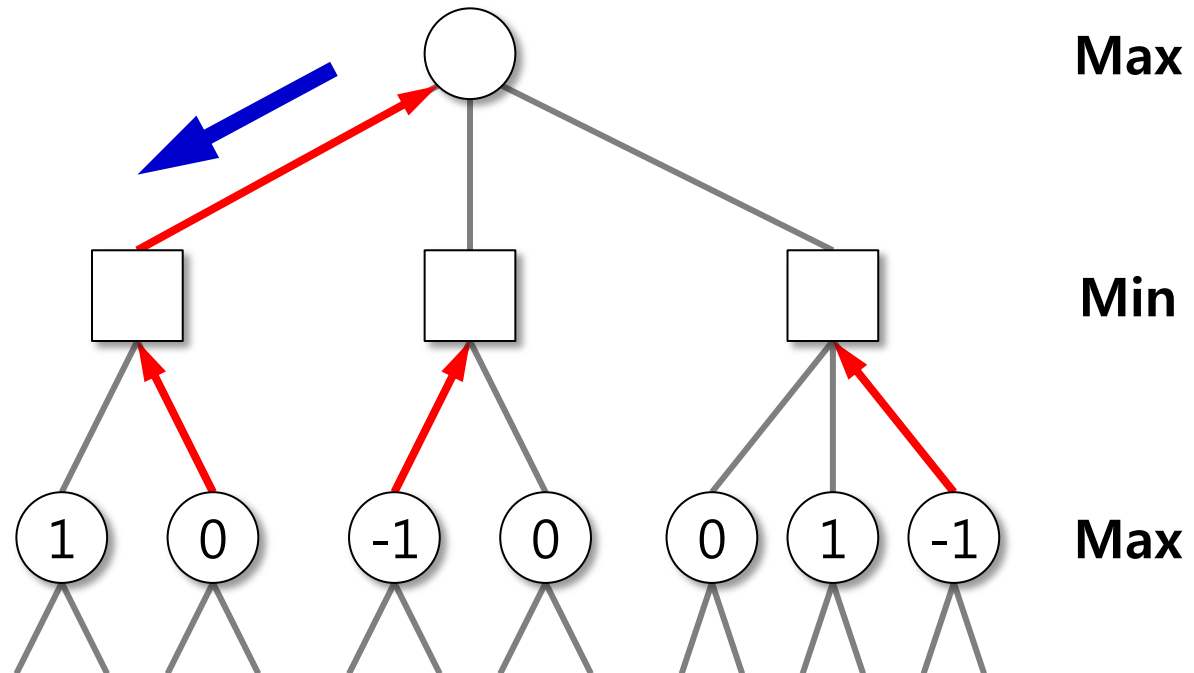
틱택토 게임의 Minmax Game Tree 예



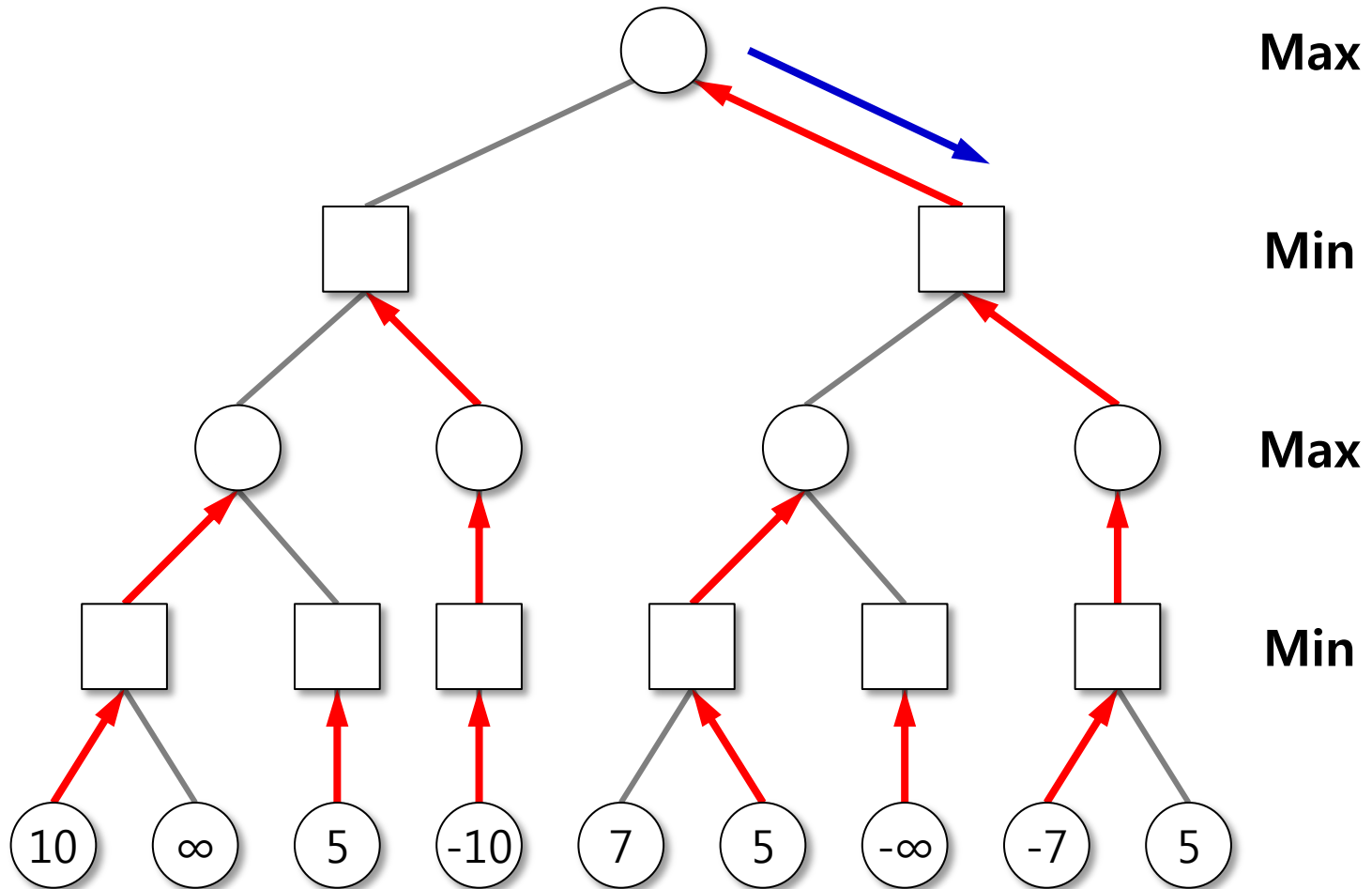
Minmax 알고리즘

- 알고리즘 이름이 Minmax인 이유는 알고리즘 실행과정이 min 값과 max 값의 선택을 반복하기 때문이다.
- 먼저, Minmax Game Tree의 단말 노드에 게임 결과 점수 부여
 - 틱택토의 예) X의 승리=1, O의 승리=-1
 - 체스의 예) 유/불리에 대한 여러 조건에 대한 가중치를 합한 평가 값
- Bottom-up으로 루트 방향으로 진행하면서
 - Max(X) 단계에서는 자식 노드의 점수 중 최대값을 선택하고,
Min(O) 단계에서는 자식 노드의 점수 중 최소값을 선택한다.
 - 두 플레이어가 모두 최선을 다한다고 가정한다.
 - 즉, X의 차례에서는 점수가 최대가 되는(X가 승리하는) 수를 선택하고,
O의 차례에서는 점수가 최소가 되는(O가 승리하는) 수를 선택한다.

틱택토의 실행 예



체스의 실행 예



Minmax와 체스

- Minmax 트리를 모두 검색하면 최선의 결과를 얻을 수 있다.
하지만, 체스와 같이 큰 게임은 모든 트리를 검색하기 어렵다.
 - 체스의 경우 한 턴에 약 35가지 경우의 수가 생긴다.
 - ◆ 1턴=35, 2턴= $35^2=1224$, ... 6턴= 35^6 =약20억, ... 10턴= 35^{10} =약2천조
 - ◆ 초보자는 4수, 고수는 8수 정도 보는데 Deep Blue는 12수를 본다.
- 이러한 큰 게임은 끝까지 검색할 수 없으므로, 검색을 멈추고 현재 상태의 점수를 예측하는 단계가 필요하다.
 - Evaluate() 함수에서 현재 보드의 상태를 보고 점수를 예측한다.
- 결국, 알고리즘의 성능은 다음 두 가지에 의해 결정된다.
 - 검색하는 트리의 깊이
 - Evaluate() 함수의 성능

MinMax 알고리즘

```
int MinMax(int depth) {  
    // White is Max, Black is Min  
    if (turn = WHITE)  
        return Max(depth);  
    else  
        return Min(depth);  
}
```

Then, call with:

```
value = MinMax(5); // 5수를 탐색
```

```

int Max(int depth) {
    int best =  $-\infty$ ;
    if (depth == 0)
        return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = Min(depth - 1);
        UnMakeMove();
        if (val > best)
            best = val;
    }
    return best;
}

```

```

int Min(int depth) {
    int best =  $\infty$ ;
    if (depth == 0)
        return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = Max(depth - 1);
        UnMakeMove();
        if (val < best)
            best = val;
    }
    return best;
}

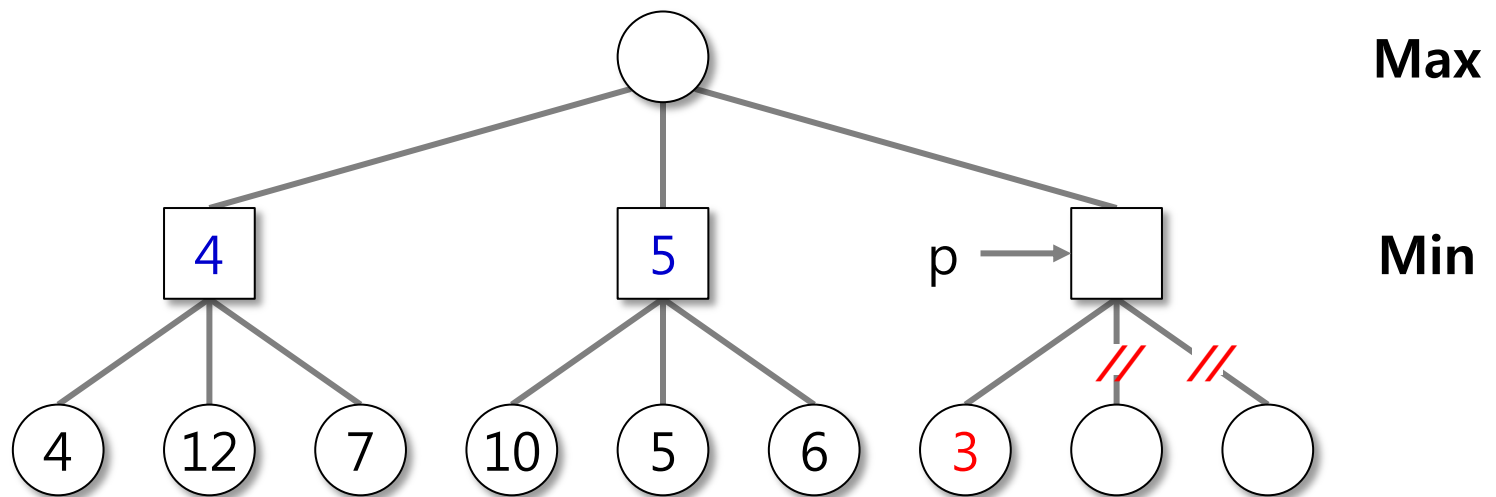
```

8.2 $\alpha - \beta$ Pruning

- Minmax 알고리즘은 무조건 모든 경우를 검색하는데, 검색할 필요가 없는 경우가 생긴다.
- 다음과 같은 경우 자식 노드에 대한 평가를 중지하여 (가지치기 하여) 불필요한 연산을 줄일 수 있다.
 - 자신이 그 경우를 택하면 자신이 불리해지는 것이 확정된 경우
 - 어떠한 경우가 자신에게 유리한 것이 확정되어 상대가 그것을 택하지 않을 확률이 높은 경우

가지치기 아이디어

- 노드 p의 값을 구하기 위해 첫 번째 자식 노드 3을 검색했다.
 - 노드 p는 Min값을 구하는 단계이므로 나머지 자식 노드를 검색해도 노드 p의 최종 값은 3보다 작거나 같을 것이다.
 - 노드 p의 부모 노드에서는 Max값을 구하게 되는데, 이미 다른 자식 노드에서 4, 5가 구해졌으므로 p의 값은 무조건 제외됨을 알 수 있다.
 - 이런 경우, 노드 p의 나머지 자식 노드는 검사할 필요가 없다.



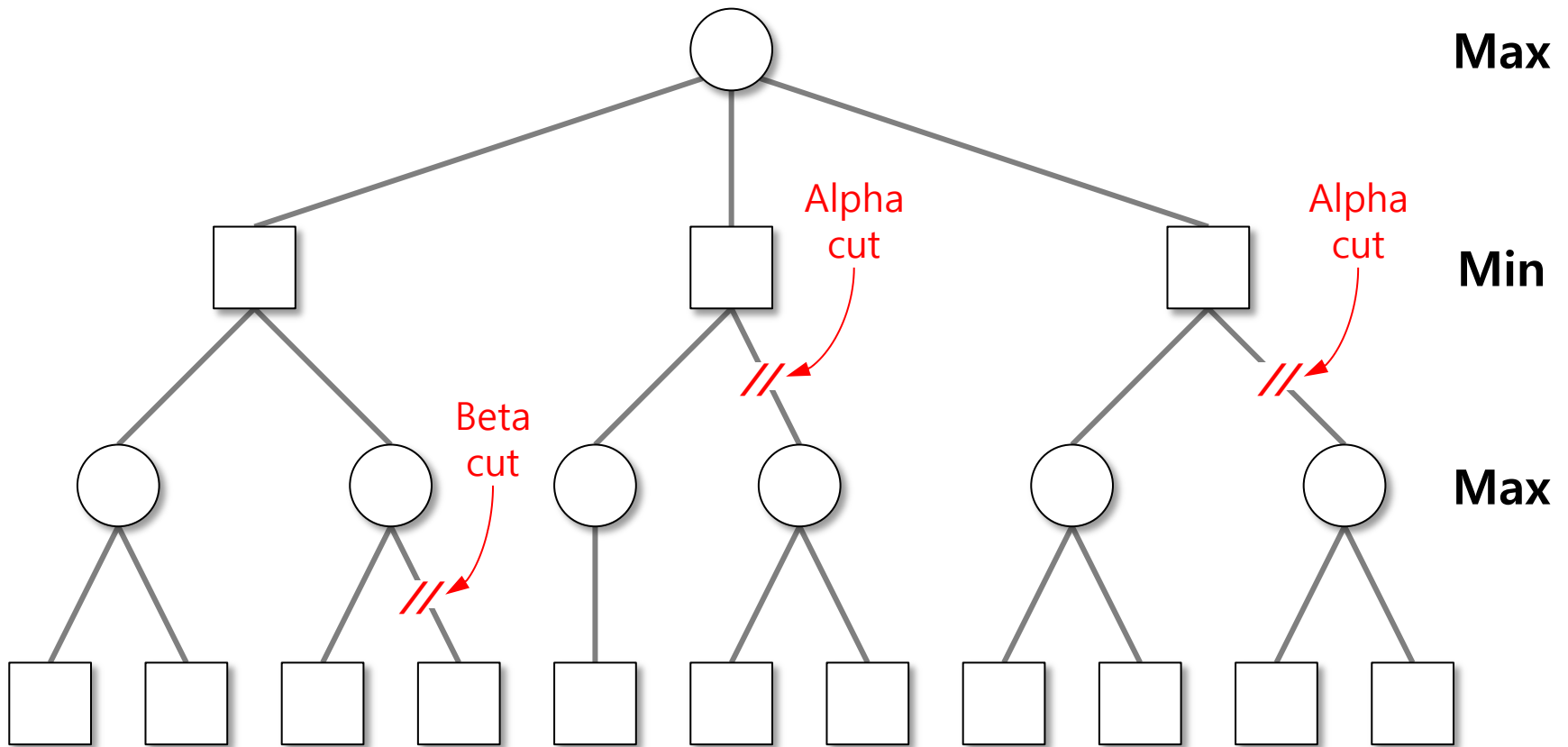
$\alpha - \beta$ Pruning 아이디어

- 검색 중에 두 개의 점수 α, β 를 사용
- α
 - Max 노드가 현재까지 검색한 가장 큰 값
 - 현재의 α 값보다 작은 값은 의미가 없으므로 가지치기의 대상이 됨
- β
 - Min 노드가 현재까지 검색한 가장 작은 값
 - 현재의 β 값보다 큰 값은 의미가 없으므로 가지치기의 대상이 됨

$\alpha - \beta$ Pruning 알고리즘

```
int AlphaBeta(int depth, int alpha, int beta) {
    if (depth <= 0)
        return Evaluate();
    GenerateLegalMoves();
    while (MovesLeft()) {
        MakeNextMove();
        val = -1 * AlphaBeta(depth-1, -beta, -alpha);
        UnMakeMove();
        if (val >= beta)
            return val;
        if (val > alpha)
            alpha = val;
    }
    return alpha;
}
```


$\alpha - \beta$ Pruning 예



- $\alpha - \beta$ Pruning의 성능은 검색 순서에 영향을 받음
 - 만일, 계속해서 worst case 순서로 검색되면 전혀 가지치기가 발생하지 않는다.
 - 만일, 계속해서 best case가 먼저 검색되면 가지의 개수는 sqrt 수준으로 줄어든다.
 - 체스의 경우, best case로 검색되면 35에서 6개 정도로 범위가 줄어들어 동일 시간에 2배 정도 깊이 검색할 수 있다.

Evaluate()

- 일반적으로 가중 함수(weight function) 사용
 - 여러 가지 상태 변수에 가중치를 적용
 - $c1 * \text{material} + c2 * \text{mobility} + c3 * \text{king safety} + \dots$
 - material 점수: pawn 1, knight 3, bishop 3, castle 3, queen 9



8.3 A* 알고리즘

- A* 알고리즘

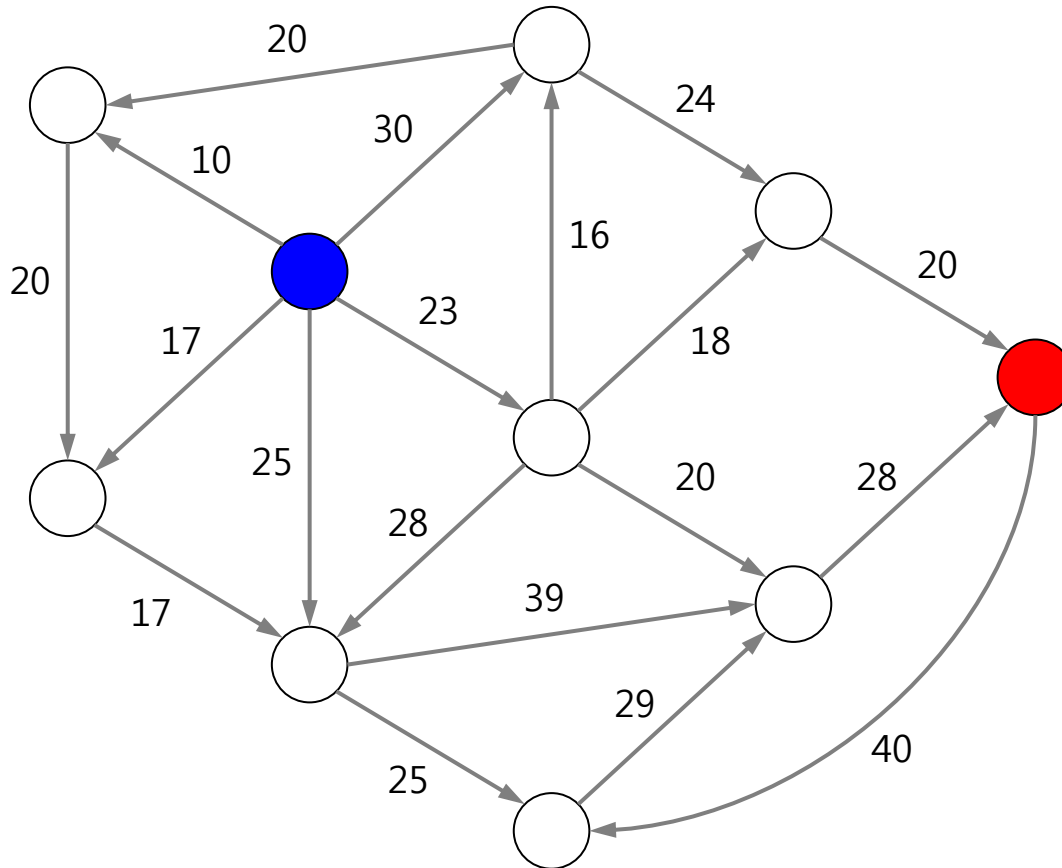
- 형식적인 방법(formal method)과 휴리스틱 방법(heuristic method)를 결합하여 효과적으로 최단거리를 찾는 그래프 탐색 알고리즘
- 정점 x 의 평가함수 $f(x) = g(x) + h(x)$ 이 가장 작은 경로로 이동
 - ◆ $g(x)$: 출발점에서 정점 x 까지의 거리
 - ◆ $h(x)$: 정점 x 에서 도착점까지의 잔여 추정 거리
 - ◆ 잔여거리의 추정치 $h(x)$ 는 절대로 실제 잔여거리보다 크면 안 된다.
- Dijkstra 알고리즘과 비교
 - ◆ Dijkstra 알고리즘은 다른 모든 정점으로 가는 최단거리를 구한다.
 - ◆ A* 알고리즘은 하나의 도착점으로 가는 최단거리를 구한다.
 - ◆ 만일 $h(x) = 0$ 이면 Dijkstra 알고리즘과 같아진다.

A* 알고리즘

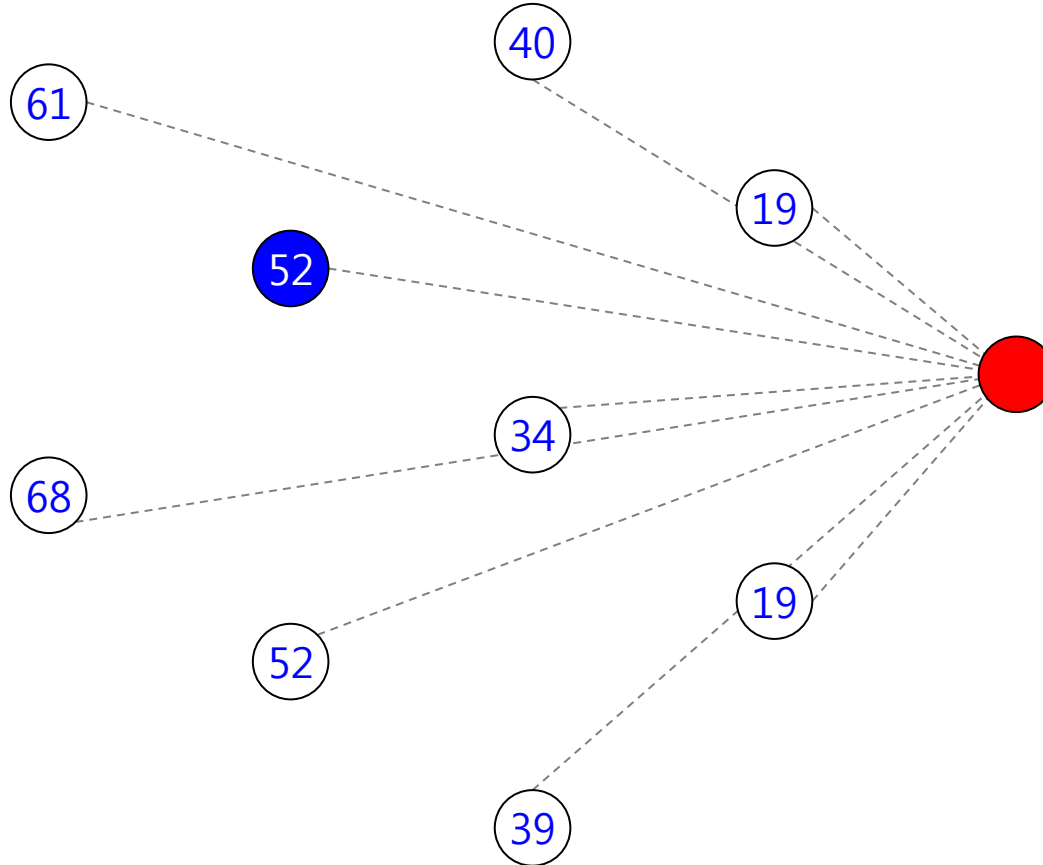
```
A-Star(start, goal) {  
    open list = { };           // 탐색중인 노드 집합  
    closed list = { start };   // 탐색이 끝난 노드 집합  
    current = start;  
    while (current  $\neq$  goal) {  
        for (current의 모든 인접 노드 p)  
            if (p  $\in$  closed list) continue;  
            if (p  $\in$  open list)  
                p의 평가값을 갱신;           // 새로운 경로의 평가값과 비교  
            else {  
                p의 평가값 = g(p)+h(p);       // 평가값 계산  
                p를 open list에 넣는다.  
            }  
        }  
        if (open list = 공집합) error 경로가 존재하지 않음;  
        open list에서 평가값이 가장 작은 노드를 current로 선택하고,  
        open list에서 제거하고 closed list에 넣는다.  
    }  
}
```

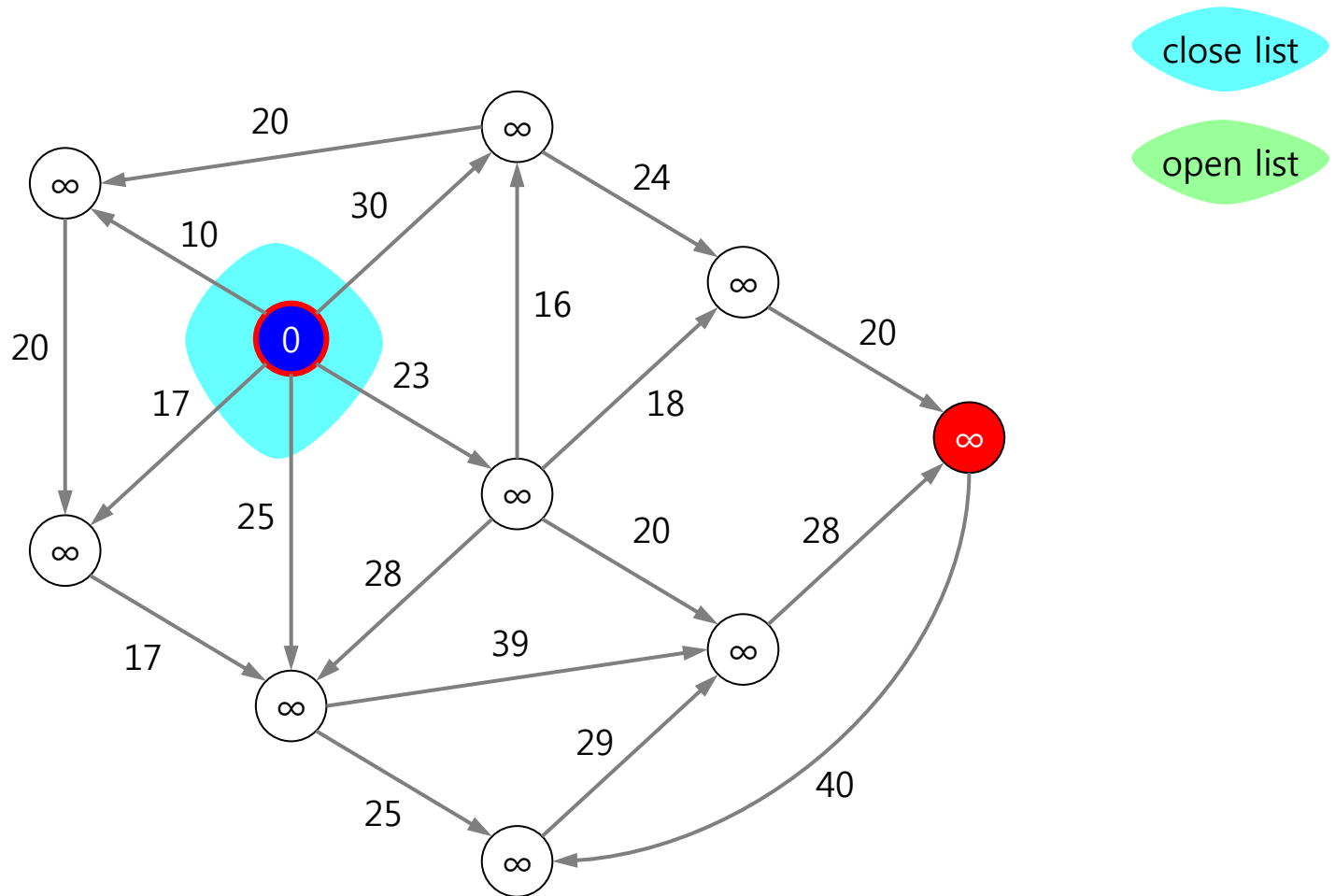
A* 알고리즘 예

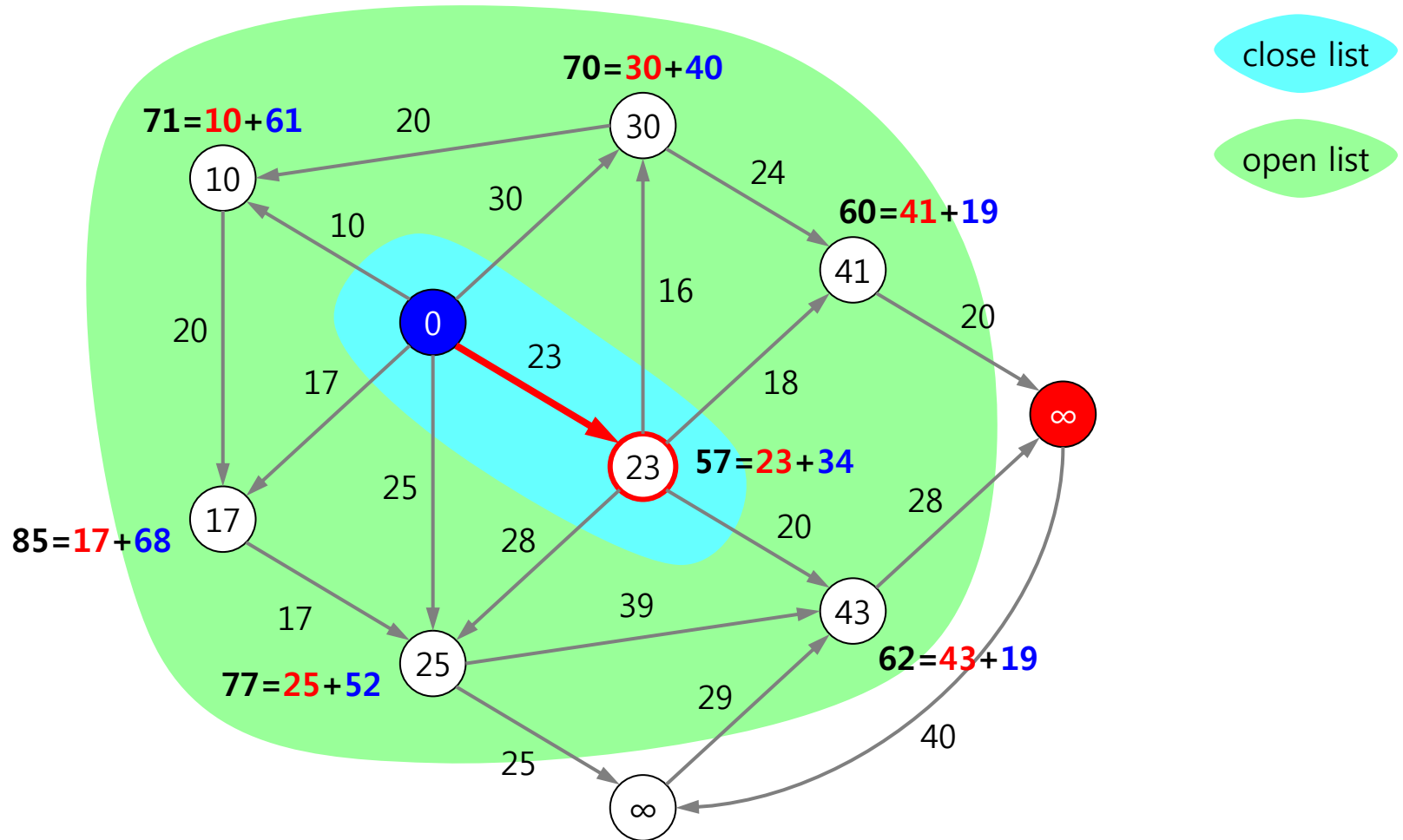
- 정점 간의 거리

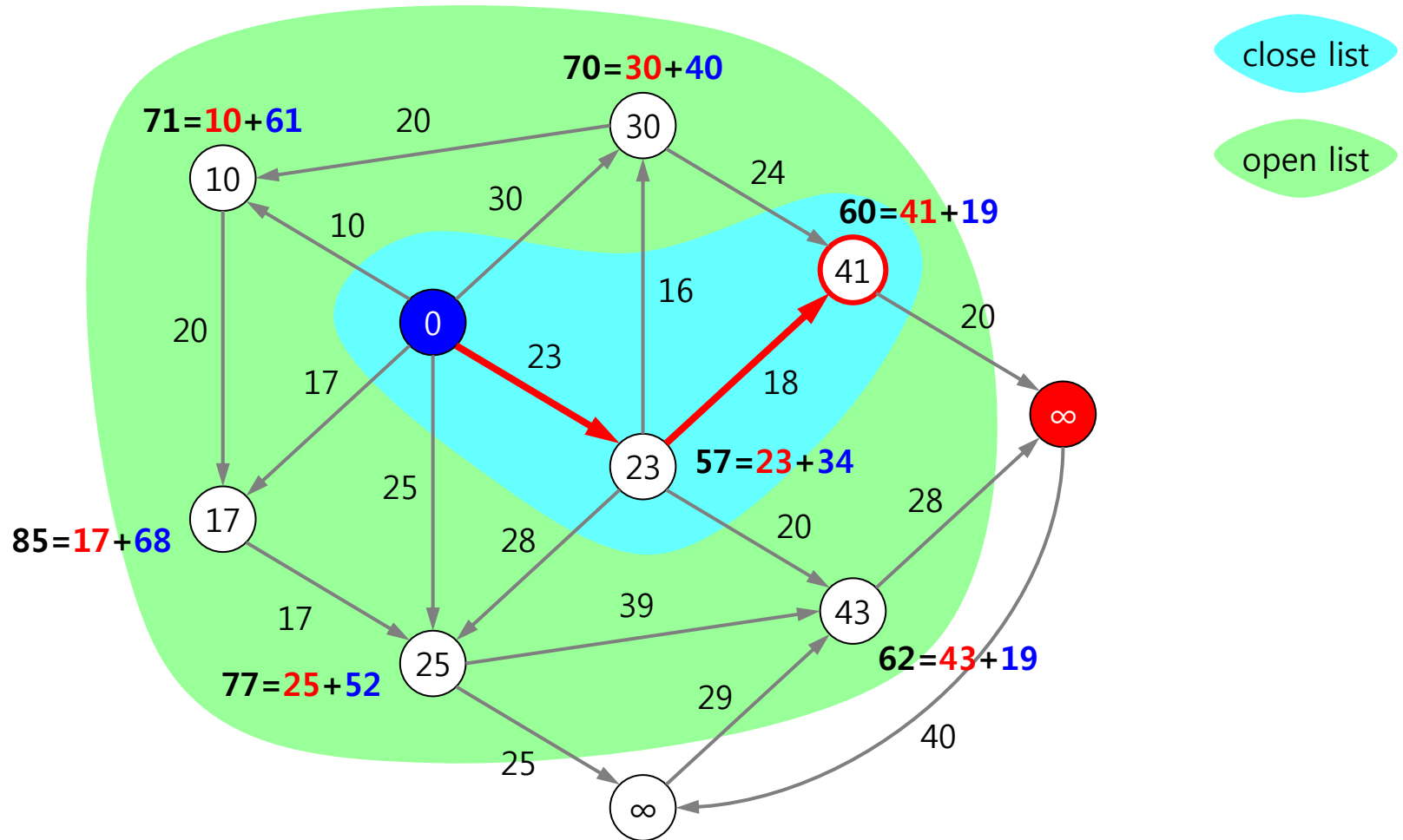


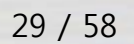
● 잔여 추정 거리

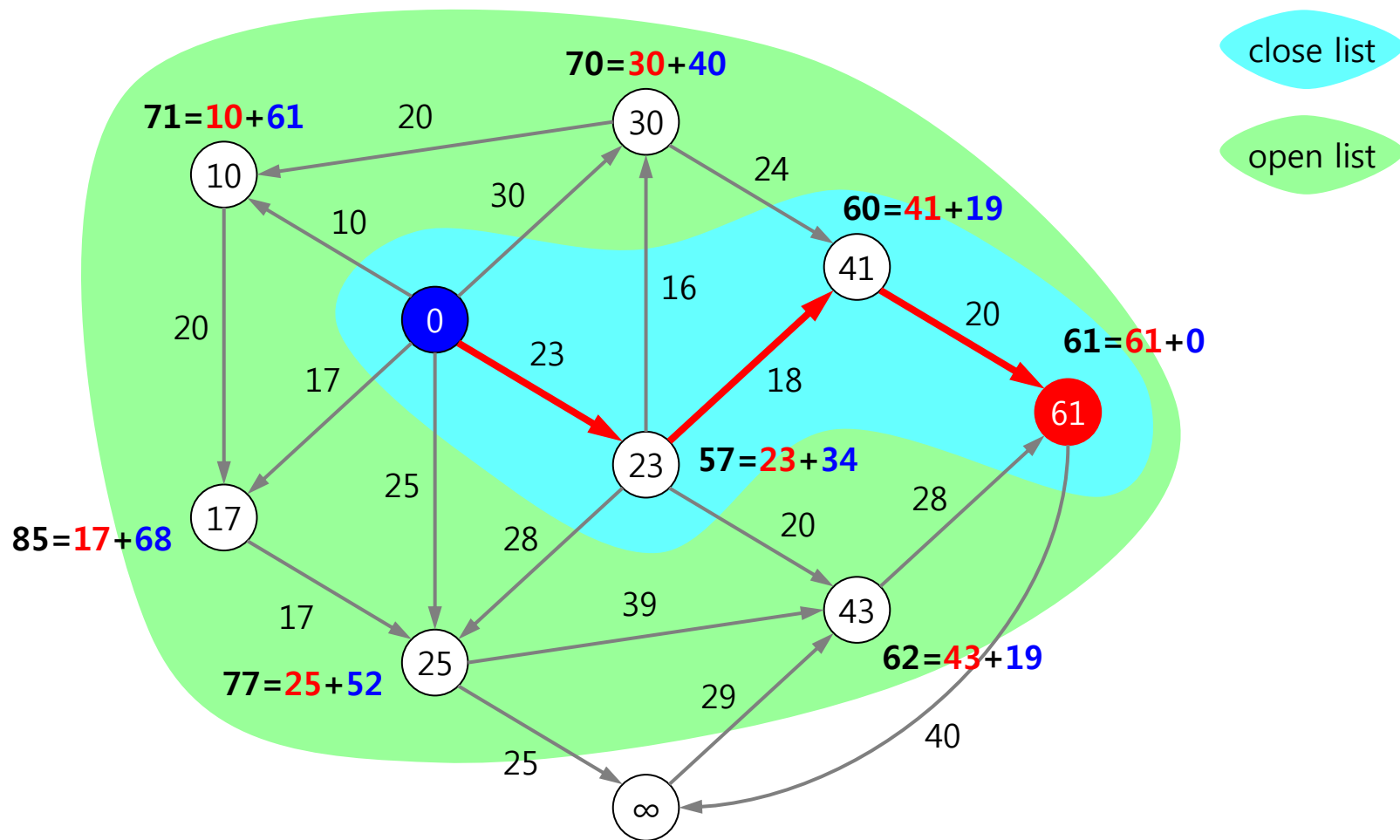








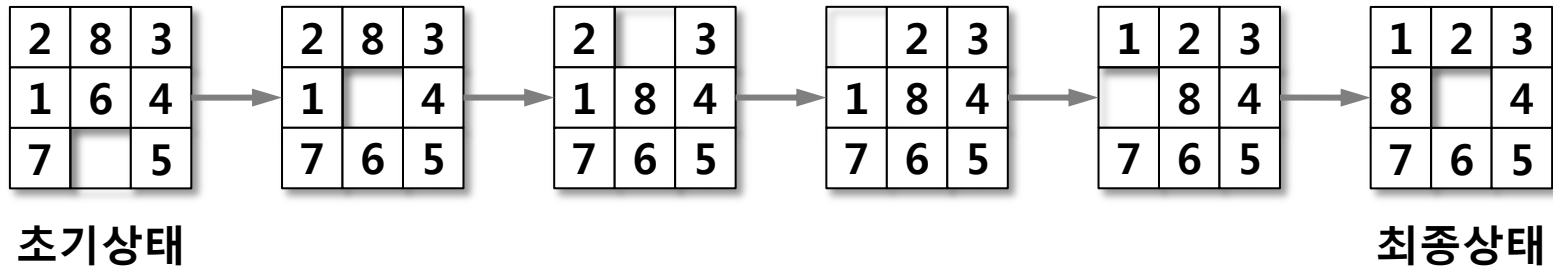




8-퍼즐 문제

- 8-퍼즐 문제

- 8개의 숫자판을 최소횟수로 움직여 원하는 모양을 만드는 문제



- A* 알고리즘의 적용

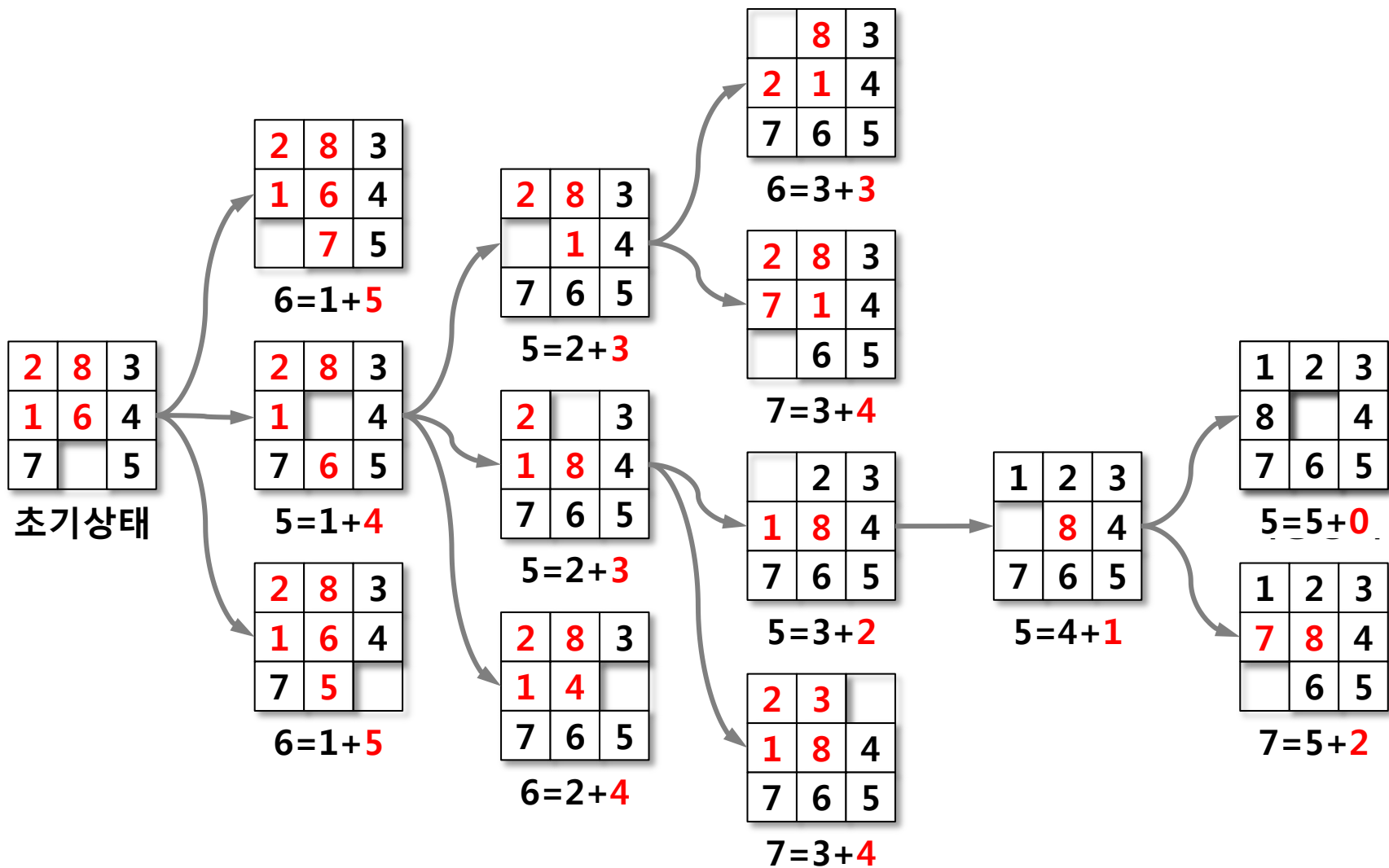
- 평가함수 $f(x) = g(x) + h(x)$

- ◆ $g(x)$: 숫자판을 이동한 회수

- ◆ $h(x)$: 최종상태와 다른 숫자판의 개수

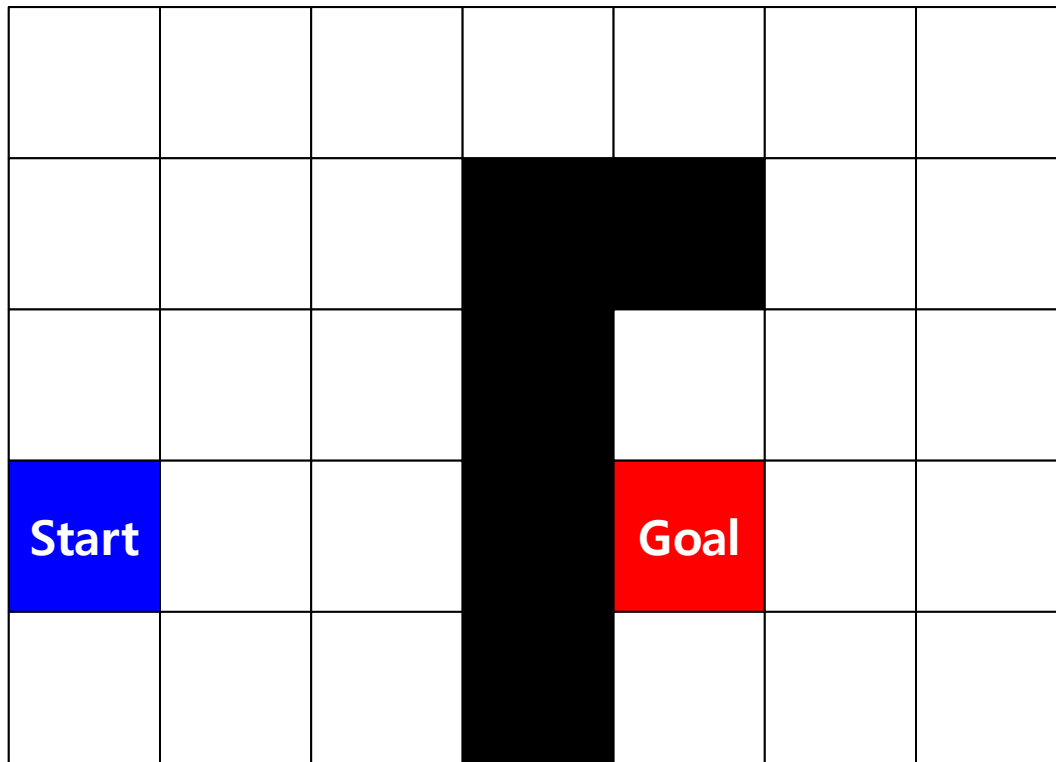
- ▶ 최종상태와 다른 숫자판을 최종상태와 같도록 이동하기 위해서는 최소한 1회 이상의 이동이 필요하므로 $h(x)$ 는 최종 상태로 이동하는 회수보다 크지 않다.

A* 알고리즘을 이용한 8-퍼즐 문제



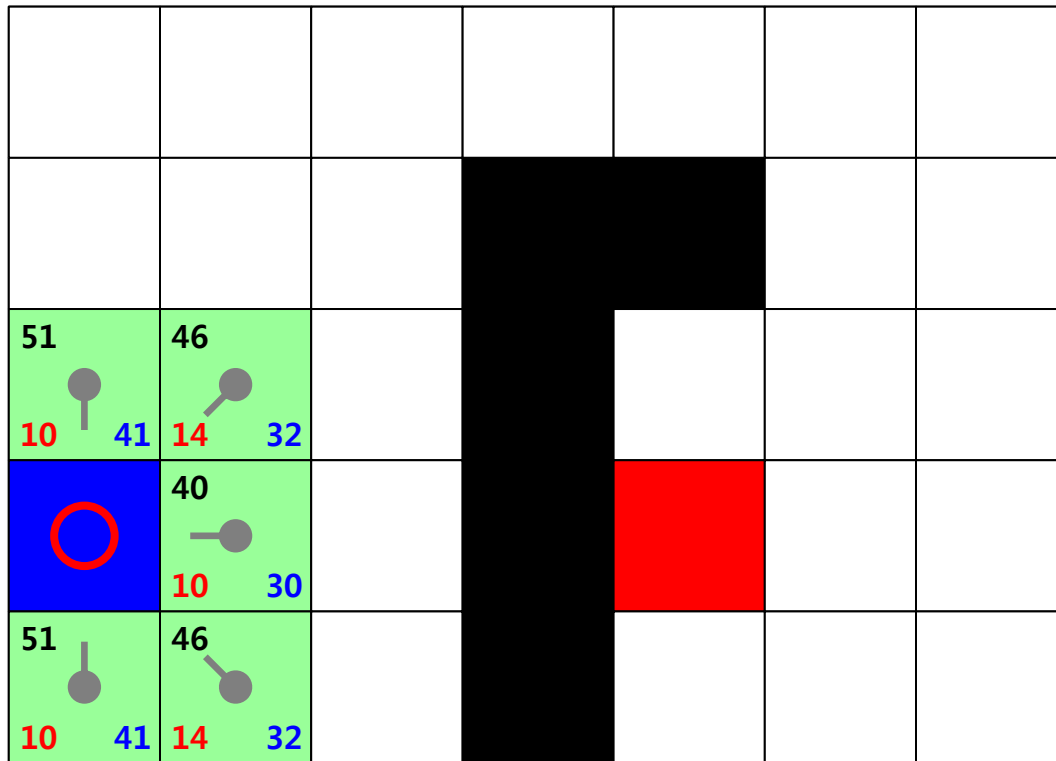
A* 알고리즘을 사용한 길찾기

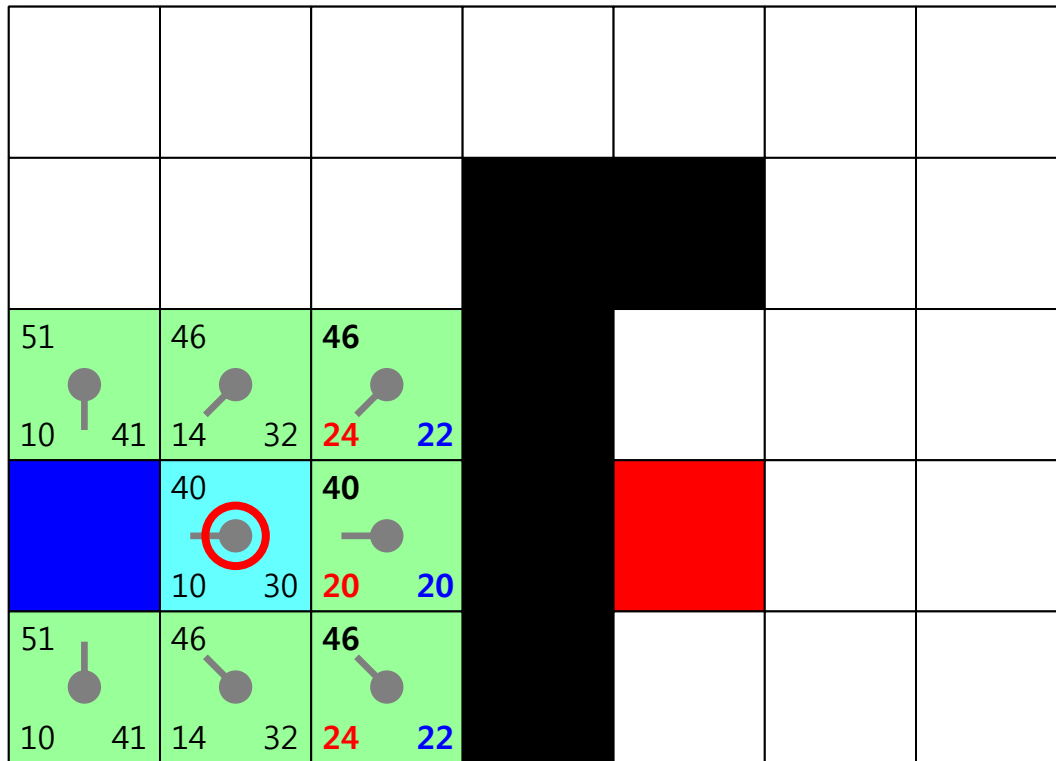
- A* 알고리즘은 장애물이 있는 게임 맵에서 목표위치까지의 경로를 찾는 문제에 효과적임

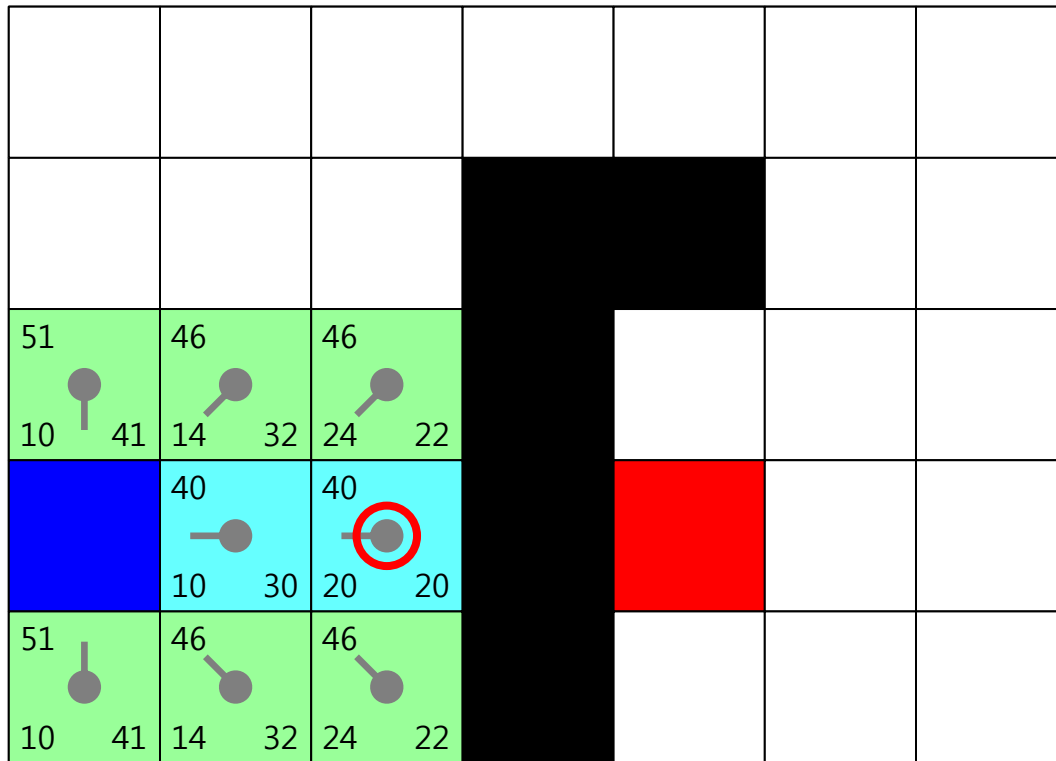


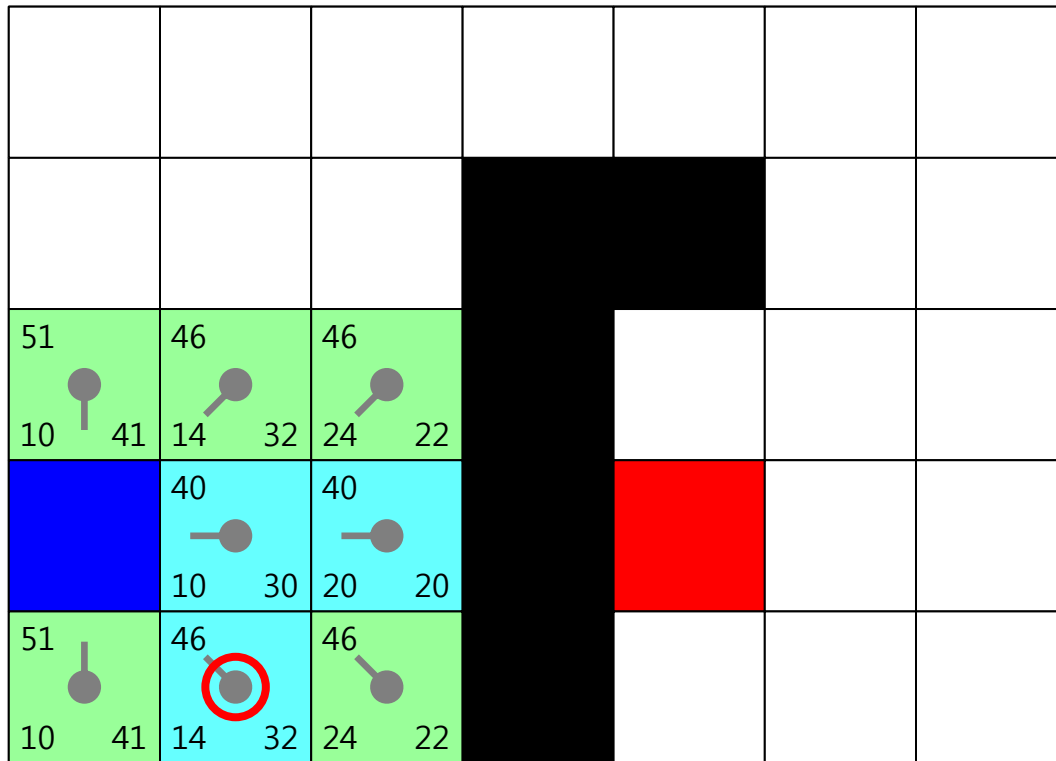
- 도착점까지의 잔여 추정 거리 = 직선거리
 - 셀 간의 거리는 10, 대각선 거리는 14

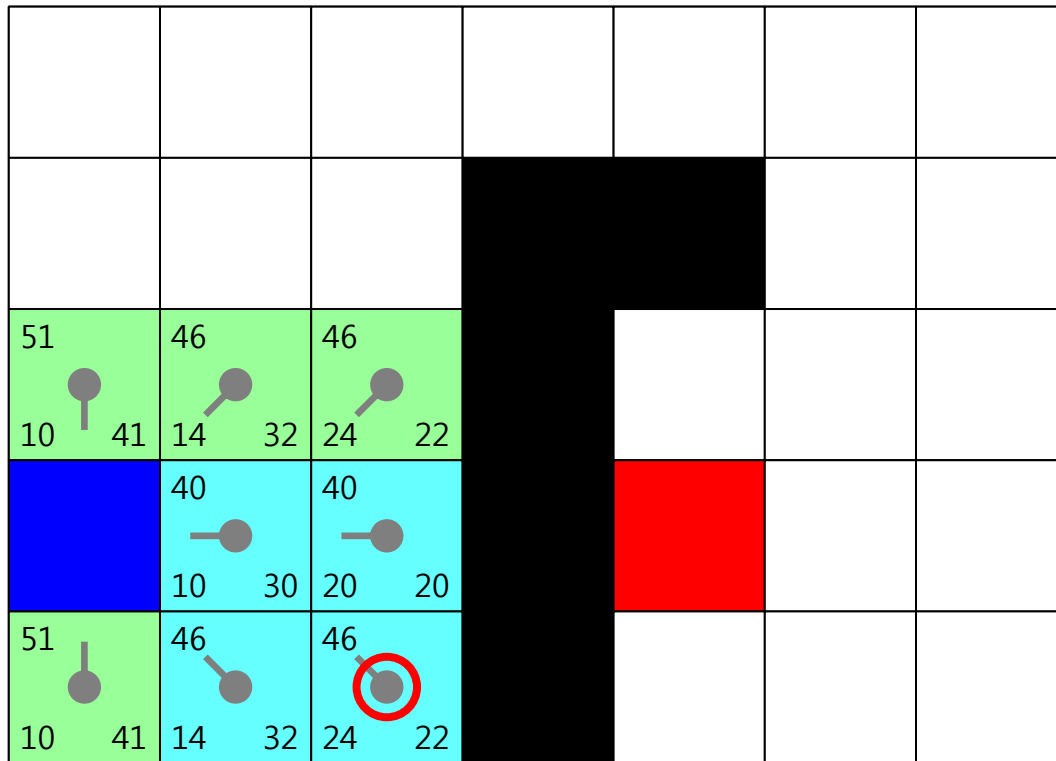
50	42	36	32	30	32	36		
45	36	28				22	28	
41	32	22				10	14	22
40	30	20				0	10	20
41	32	22				10	14	22

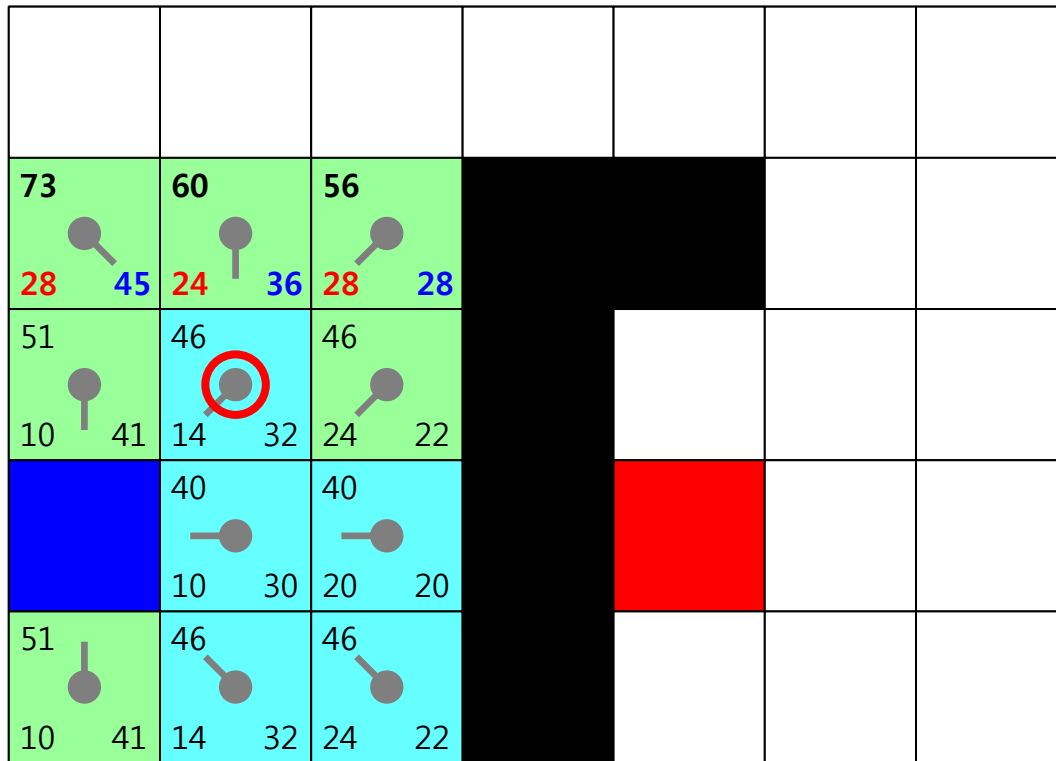


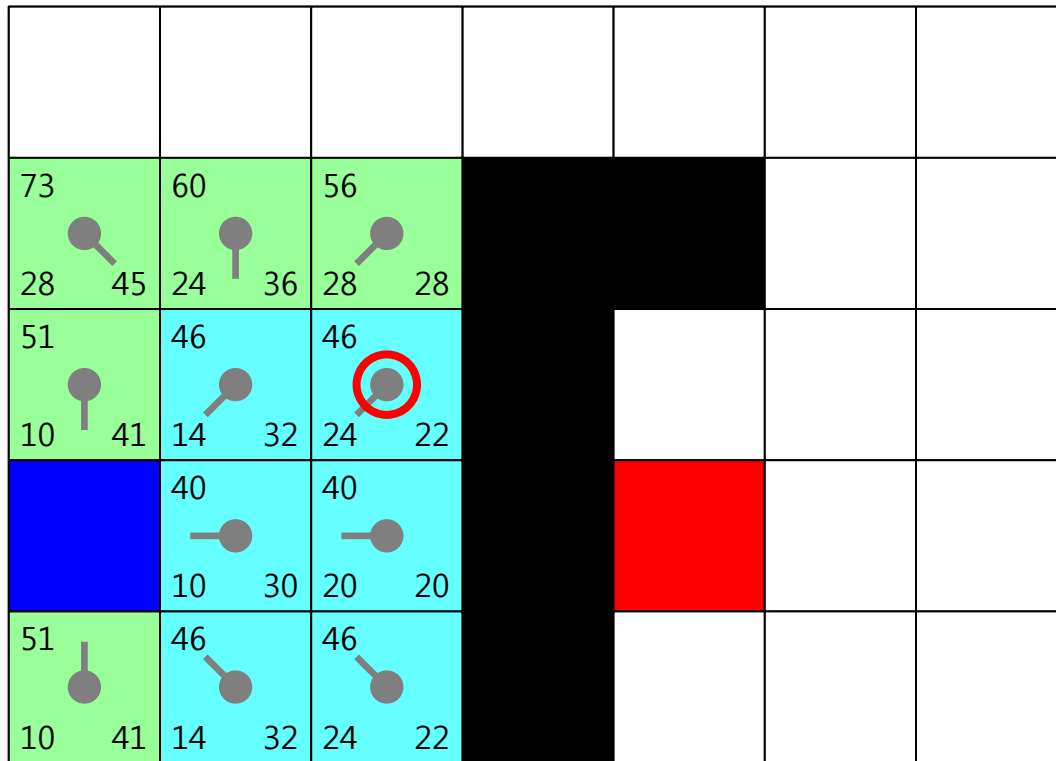




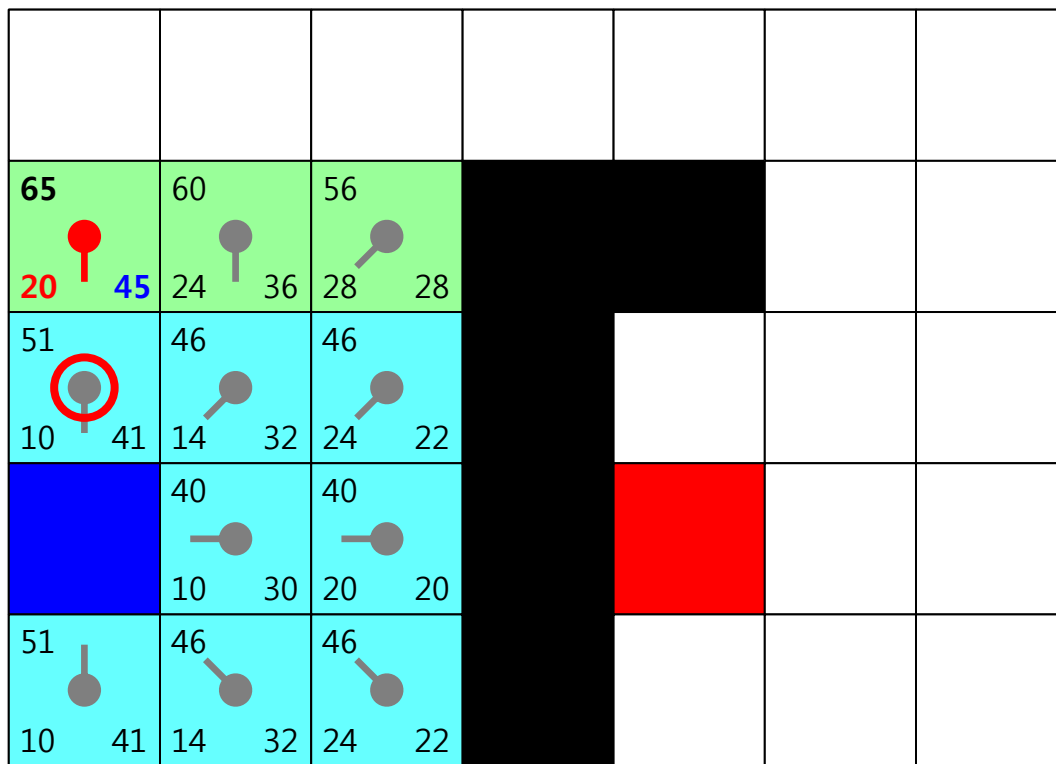


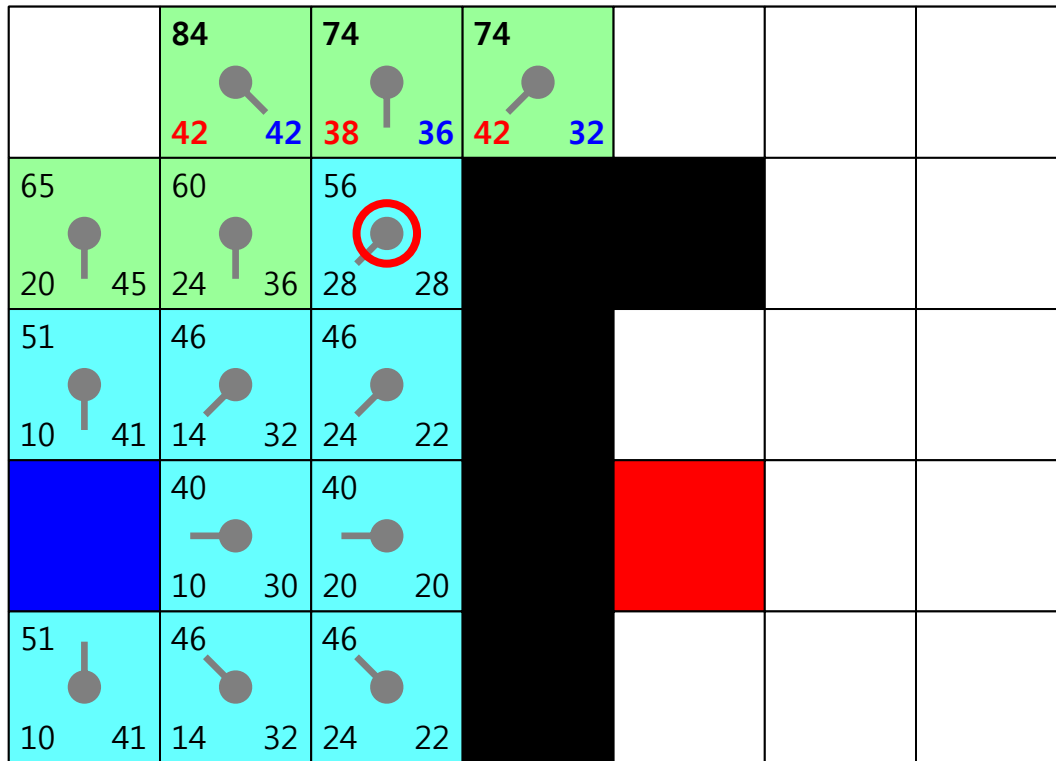


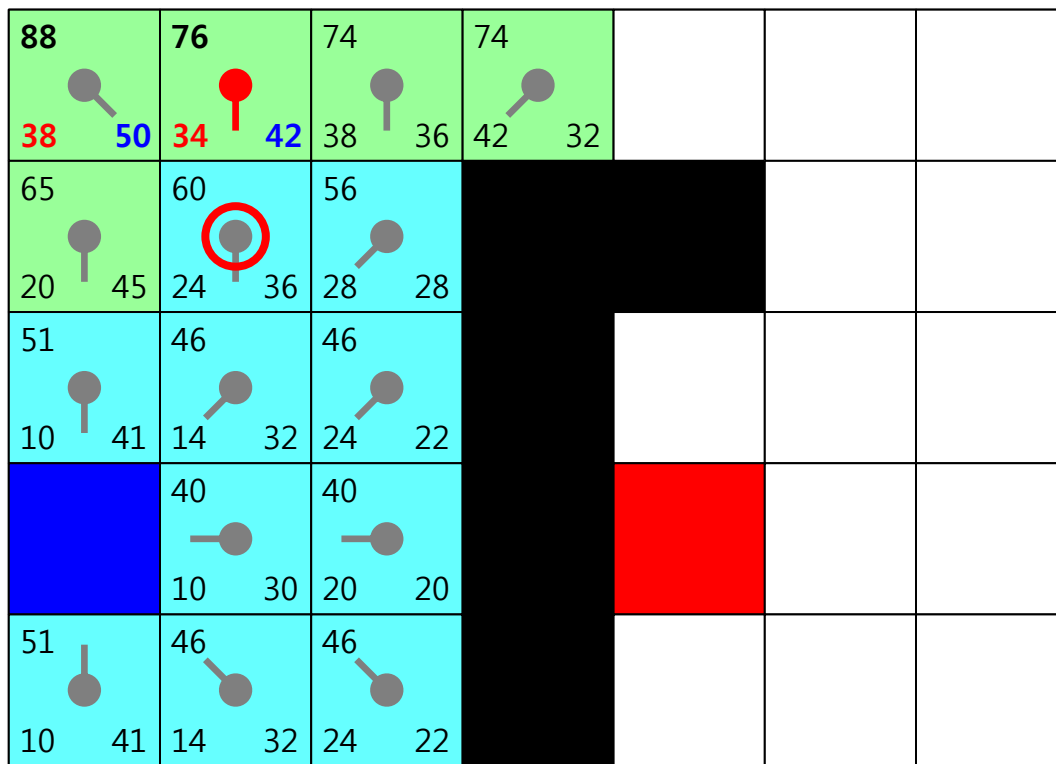


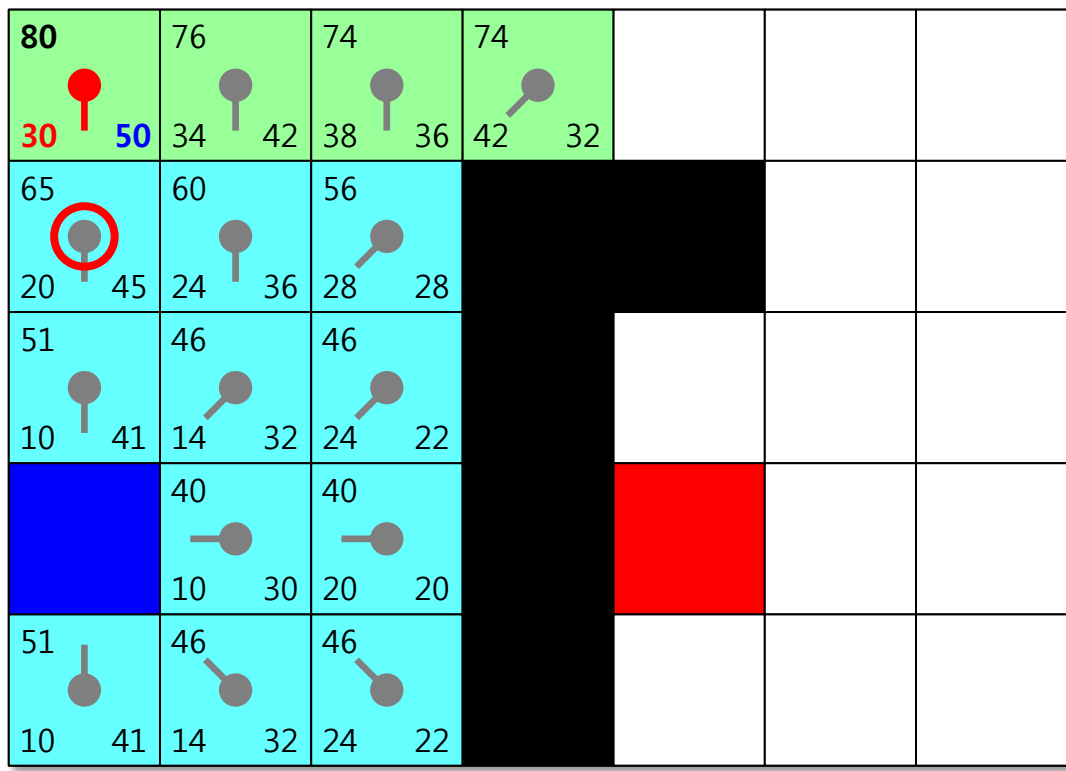


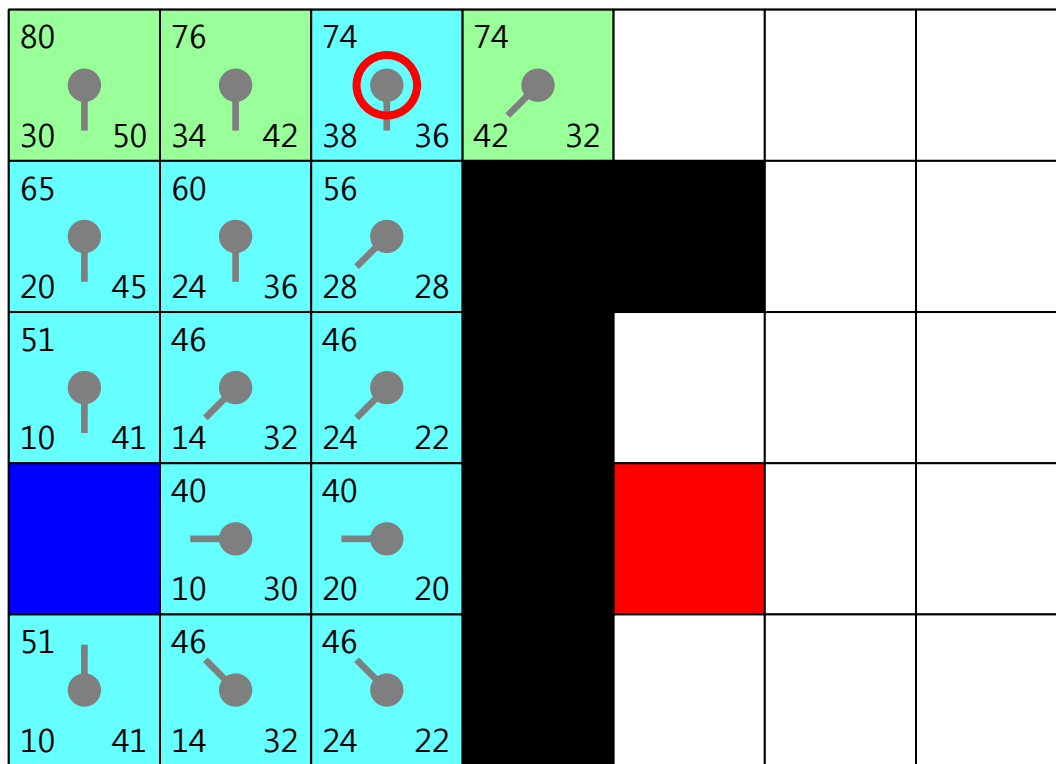
73 28 45 ●	60 24 36 ●	56 28 28 ●	<div></div>			
51 10 41 ●	46 14 32 ●	46 24 22 ●				
	40 10 30 ●	40 20 20 ●				
51 10 41 ●	46 14 32 ●	46 24 22 ●				

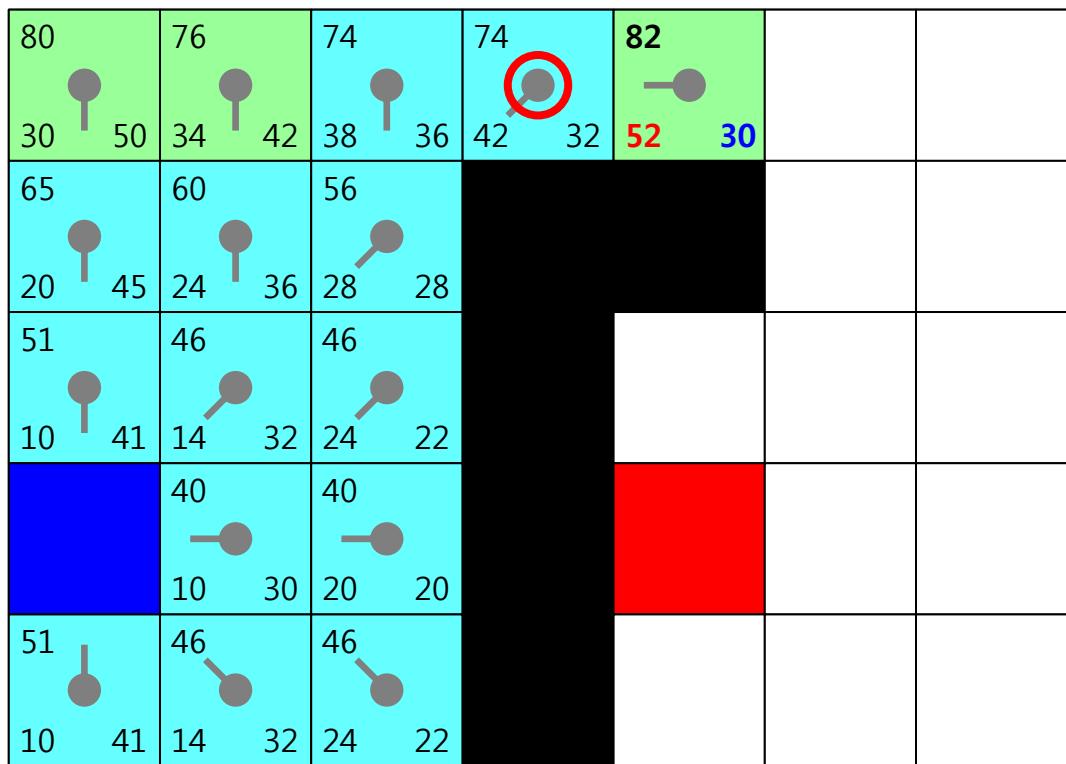


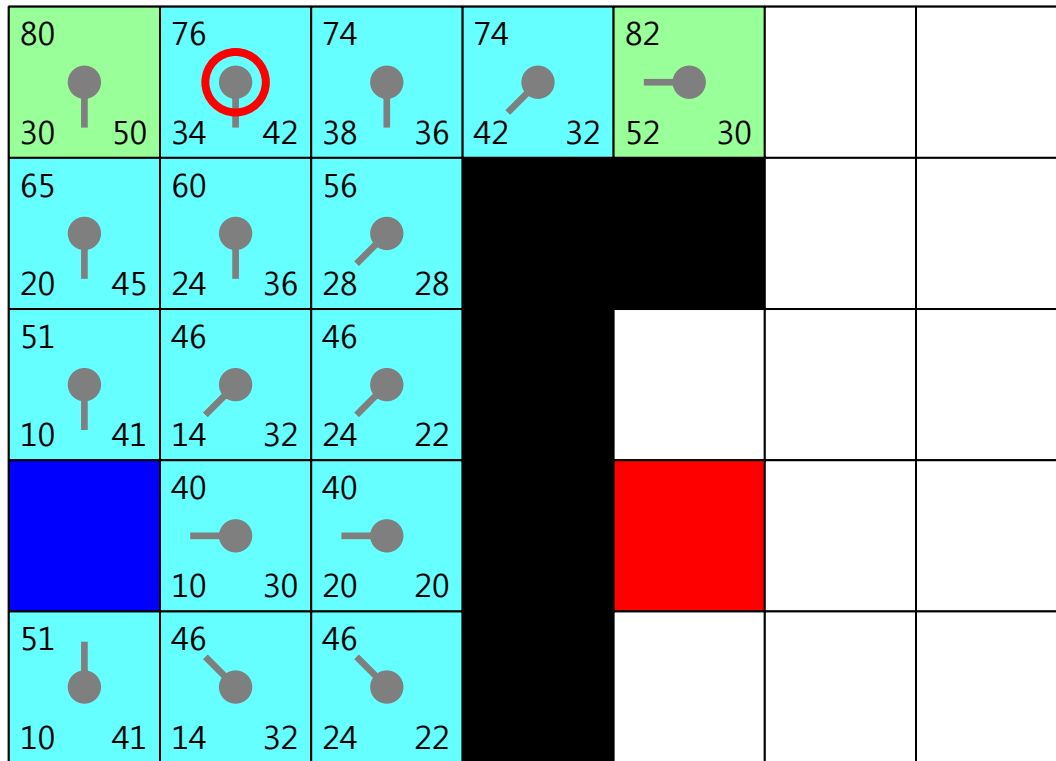


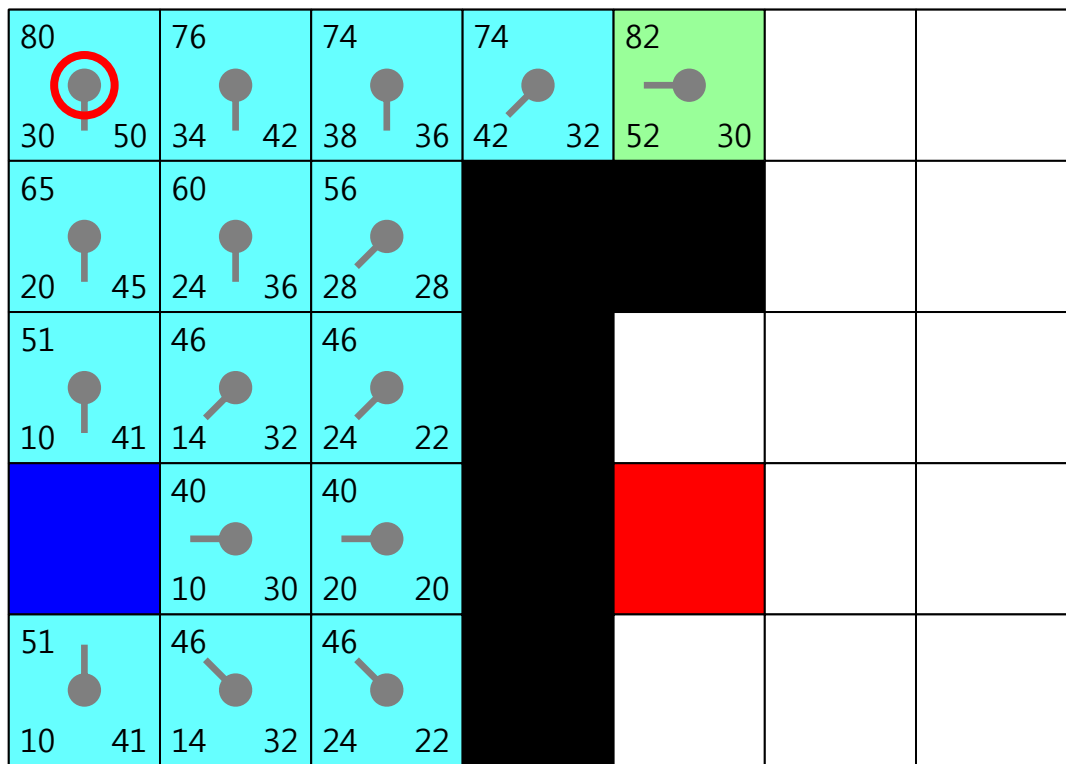







































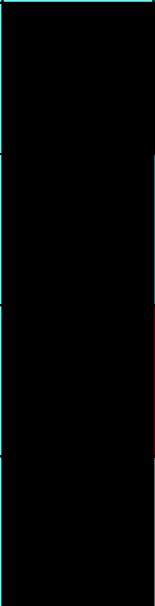







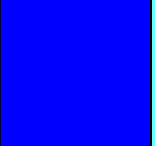



































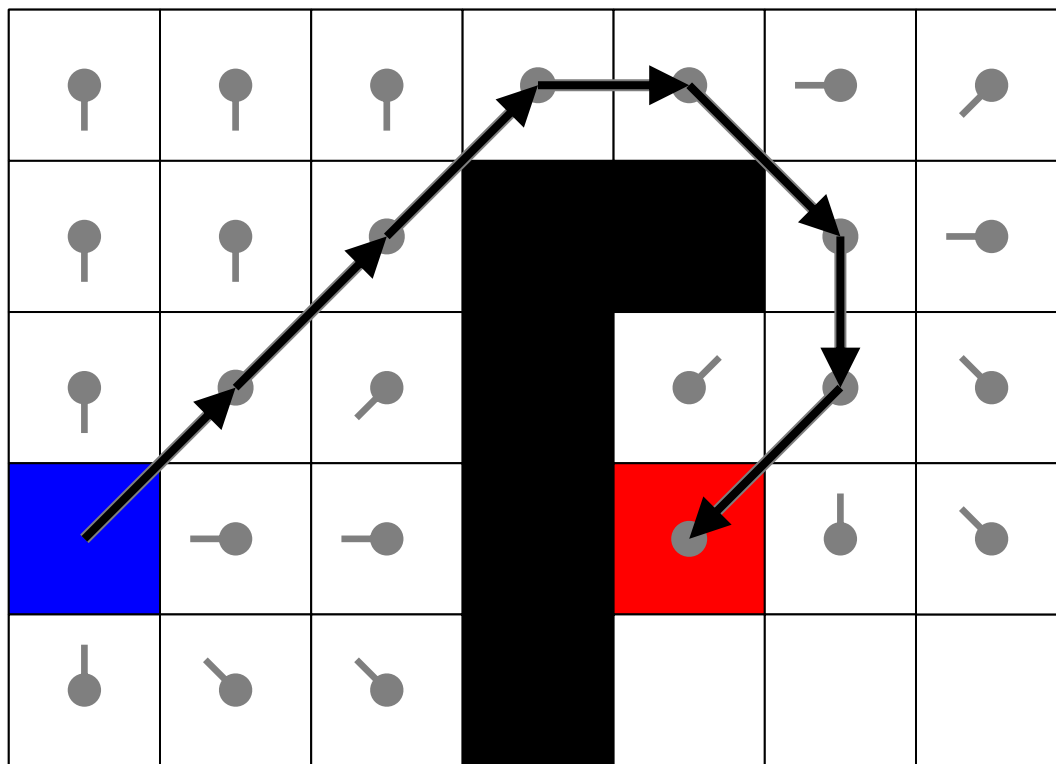


80 30 50 	76 34 42 	74 38 36 	74 42 32 	82 52 30 	94 62 32 	116 80 36
65 20 45 	60 24 36 	56 28 28 			88 66 22 	104 76 28
51 10 41 	46 14 32 	46 24 22 			90 80 10 	102 80 22
	40 10 30 	40 20 20 				
51 10 41 	46 14 32 	46 24 22 				

80  30 50	76  34 42	74  38 36	74  42 32	82  52 30	94  62 32	116  80 36
65  20 45	60  24 36	56  28 28			88  66 22	104  76 28
51  10 41	46  14 32	46  24 22			90  80 10	102  80 22
					90  76 14	
					96  86 10	110  90 20
51  10 41	46  14 32	46  24 22				

80 30 50 	76 34 42 	74 38 36 	74 42 32 	82 52 30 	94 62 32 	116 80 36 
65 20 45 	60 24 36 	56 28 28 			88 66 22 	104 76 28 
51 10 41 	46 14 32 	46 24 22 			90 80 10 	102 80 22 
	40 10 30 	40 20 20 			90 90 0 	96 86 10 
51 10 41 	46 14 32 	46 24 22 				

80 30 50 	76 34 42 	74 38 36 	74 42 32 	82 52 30 	94 62 32 	116 80 36 
65 20 45 	60 24 36 	56 28 28 			88 66 22 	104 76 28 
51 10 41 	46 14 32 	46 24 22 			90 80 10 	102 80 22 
	40 10 30 	40 20 20 			90 90 0 	96 86 10 
51 10 41 	46 14 32 	46 24 22 				



A* 알고리즘 분석

- A* 길찾기 알고리즘의 특징
 - 장애물이 없는 경우에는 깊이 탐색을 하여 도착점을 향해 직진한다.
 - 장애물을 만나면 너비 탐색을 하여 장애물 주변의 회피로를 찾는다.
- A* 알고리즘의 약점
 - 맵의 크기가 크면 열린 목록이나 닫힌 목록에 매우 많은 수의 노드가 들어가므로 실시간 연산이 어려울 수 있다.
 - 시작점에서 도착점까지의 경로가 존재하지 않은 경우, A* 알고리즘은 시작점에서 갈 수 있는 모든 위치를 검색하므로 매우 비효율적이다.
 - 이러한 문제를 해결하고 효율을 높이기 위한 다양한 방법이 사용된다.

Q&A

