게임프로그래밍

게임인공지능

박종승

Dept. of CSE, Incheon Nat. Univ.
jong@inu.ac.kr
http://ecl.inu.ac.kr

게임에서의 인공지능

- 게임 인공지능 동향
  - 1970년대 비디오 게임이 생기면서부터 시작
    - 초기에는 부수적인 요소로 인식되었음
  - 1980년대 후반부터 본격적인 발전
    - 본격적인 인공지능 모듈이 포함되기 시작함
    - SimCity, WarCraft II, Age of Empires 2 등

인공지능의 접근방법

- 상향식(bottom-up) 접근
  - 인간의 뇌의 복잡한 신경망을 전자적으로 복제하는 접근
  - 인공신경망
    - 생물학적인 신경망의 처리구조를 시뮬레이션
    - McCulloch & Pitts(1943): 학습하고 패턴을 인식
    - Frank Rosenblatt(1957): 사람의 사고 절차를 흉내 내고 문자를 인식
  - 인공신경망: 애매하고 복잡한 문제의 해결책으로 제시되어왔음
    - 문자인식, 패턴인식, 음성인식 등
    - 정형화된 모델의 설정이 어려운 경우의 해결책
- 하향식(top-down) 접근
  - 인간의 뇌의 행위를 컴퓨터 프로그램으로 흉내 내려는 시도
  - 대표적으로 전문가시스템
    - 논리와 규칙을 사용하여 주어진 정보로부터 문제를 해결하는 접근
    - 병명을 진단하는 전문가시스템: 의사에게 좋은 보조수단

게임 인공지능이란

- 게임 플레이어: 사람, NPC(non-player character)
- 게임 인공지능(game artificial intelligence)
  - NPC가 지능을 가진 것처럼 행위하도록 하는 기술
    - 환경이나 경험으로부터 지능적인 판단 및 자율적인 행동이 가능
  - NPC가 사람과 동등하게 인간에 대적할 수 있는 지능을 갖추어야 함
  - 지능? – 다양한 의미로 해석될 수 있음
- 사람과 컴퓨터의 사고과정의 차이
  - 사고 과정의 차이
    - 사람: 확산식(diffuse), 연상식(associative), 융화식(integrated)
    - ↔ 컴퓨터: 직접식(direct), 선형식(linear), 산술식(arithmetic)
    - 근본적인 극복이 어려움: 순수한 지능의 구현은 아직 시기상조?
- 게임 인공지능 : 지능 개념의 범위를 축소
  - 게임 인공지능의 조건
    - 논리적으로 그럴듯한 행위를 생성, 게임플레이 균형을 유지, 예측 불가
  - 얼마든지 구현할 수 있음

게임 인공지능의 역사

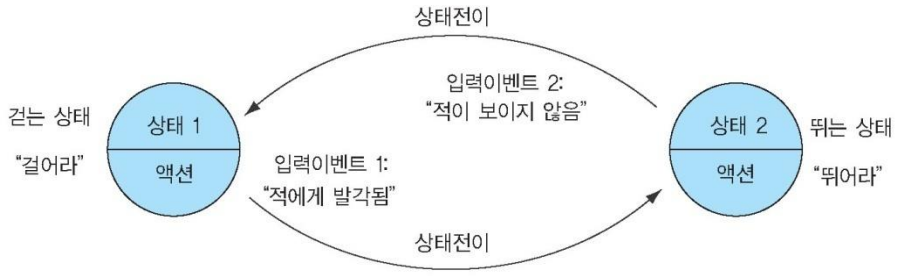
- 출발
  - 1970년대 싱글 플레이어 모드를 지원하는 게임
    - 적군의 행동 패턴을 구현하는 기술
    - Qwak!(1974), Pursuit(1975) 등
  - 더욱 발전된 형태: 적군의 난이도, 행동 패턴
    - Space Invaders(1978), Galaxian(1979), Pac-Man(1980)
- 발전
  - 1990년대 유한상태기계(FSM)를 사용하기 시작함
  - 실시간전략(RTS) 게임들이 등장하기 시작
    - 많은 인공지능 문제들을 고려하기 시작
  - 초기: 상태의 수를 몇 개 정도로만 단순하게 구현.
    - 적군들이 이미 모든 것들을 알고 있으면서 플레이어를 속임
  - 후기: 적군의 행동을 미리 예측할 수 없는 기법들이 제시되었음
  - 최근: 다양한 인공지능 구현
    - 신경망, 유전자알고리즘, 기계학습, 인공생명 등의 복합 사용

유한상태기계 기술 현황

- 유한상태기계
  - 유한한 개수의 상태로 NPC의 행동 양식을 표현하는 기법
  - 가장 전통적인 구현 방식. 직관적이고 구현도 쉬움
    - 단순한 게임의 경우에는 널리 사용되는 방식임
    - 효율적이며 실질적인 성과를 제공하는 인기있는 기법
    - 더 진보적인 방법: 신경망이나 유전자알고리즘
      - 실제로는 게임에 거의 적용되지 않고 있음. 왜?
  - 단점
    - 상태의 수가 많아질수록 상태 전이 구조가 크게 복잡해짐
    - 상태전이에 일정한 패턴이 있음
      - 플레이어가 게임을 진행하면서 NPC의 행동 패턴을 예측할 수 있음
  - 미래
    - 단점들을 보완한 계층적 유한상태기계, 퍼지 유한상태기계 등의 등장
    - 인기는 여전할 것임

유한상태기계란

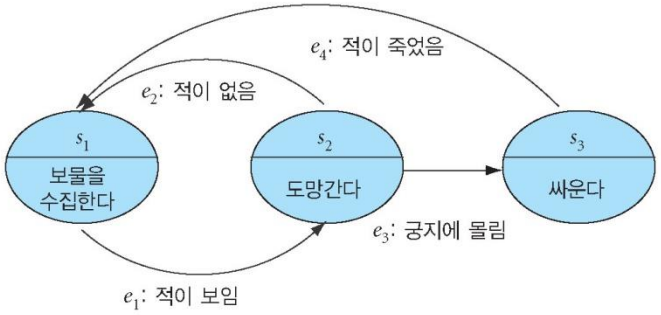
- 유한상태기계(FSM; finite state machine)란
  - 유한상태기계: 복잡한 시스템의 행위를 표현하는 추상적인 모델
    - 제한된 수의 상태를 정의, 상태전이가 상황에 따라 이루어짐
  - 유한상태기계의 요소
    - 상태: 행위를 정의하고 액션을 생성함
      - 상태 집합 S = {s}
    - 상태전이: 각 상태에서 전이 규칙이 만족되면 다른 상태로 이동함
      - 상태전이 집합 T = {t}, t(s,e)=s'
    - 입력이벤트: 규칙의 만족을 유발하여 상태전이를 유도함
      - 발생 가능한 입력이벤트 집합 E = {e}
  - 단순한 유한상태기계의 예



[그림 18-1] 유한상태기계의 예

유한상태기계의 작성

- 캐릭터의 행위 패턴의 모델링
  - 직관적임: 상태들을 정의하고 각 상태에서의 자극에 따른 반응을 표현



[그림 18-2] 단순한 캐릭터의 유한상태기계의 예

- 상태전이집합의 다양한 표현 방법
  - 상태전이표
    - 처리 속도가 빠르고 개발이 쉬움, 메모리 낭비를 초래함
  - 리스트로 표현
    - 메모리 사용이 효율적임

[표 18-1] 유한상태기계의 상태전이 집합을 표로 표현한 예

상태 \ 입력이벤트	e_1	e_2	e_3	e_4
s_1	s_2	–	–	–
s_2	–	s_1	s_3	–
s_3	–	–	–	s_1

[표 18-2] 유한상태기계의 상태전이 집합을 리스트로 표현한 예.

상태	상태전이
s_1	(e_1, s_2)
s_2	$(e_2, s_1), (e_3, s_3)$
s_3	(e_4, s_1)

유한상태기계의 코딩

- 상태의 갱신
  - 발생된 이벤트를 확인하고 그에 따라 상태를 갱신
  - update 함수를 호출하여 수행됨
    - 이벤트를 확인하고 이벤트에 따라서 상태전이를 수행
- 갱신 함수의 호출 방법
  - 폴링(polling) 방식
    - 주기적으로 매번 함수를 호출. 가장 단순한 형태임.
    - 이벤트가 발생하지 않더라도 주기적으로 계속 호출됨.
  - 이벤트기반 기법
    - 이벤트가 발생할 경우에 해당 이벤트의 처리를 담당하는 콜백함수 호출
  - 다중스레드 기법
    - 이벤트를 확인하고 발생된 이벤트 처리를 담당하는 전용 스레드를 둠
    - 가장 효율적임. 그러나, 디버깅이 어렵고 코딩이 난해함.

유한상태기계의 적용 예

- 상태, 이벤트, 상태전이의 설계
  - 상태의 정의 : STATE_XXX
  - 이벤트의 정의 : EVENT_XXX
  - 전이집합의 정의 : { { STATE_XXX, EVENT_XXX, STATE_XXX }, … }
- 상태 구조체의 구현 – State
  - DWORD stateID
  - std::map<DWORD,DWORD> transitions
- 유한상태기계 클래스의 구현 – FiniteStateMachine
  - std::map<DWORD,State*> states
  - DWORD currentStateID

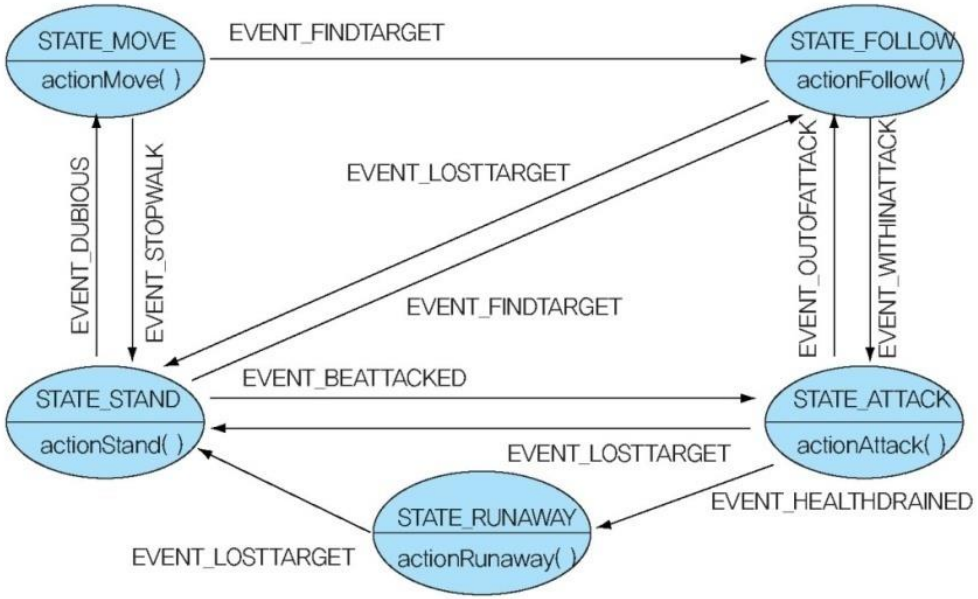
유한상태기계 예제

- 예제

01.FSMNumber
[FSMChasers]

유한상태기계 예제

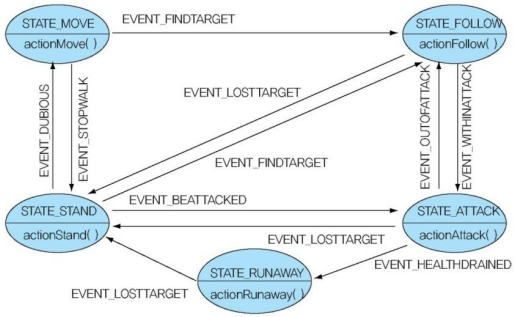
- 캐릭터 클래스의 구현 – Character
  - FinateStateMachine* stateMachine
- 응용프로그램의 구현
  - std::vector<Character*> enemyCharacters
- FSM



[그림 18-3] 단순한 캐릭터의 유한상태기계의 예.

유한상태기계 예제

- 상태, 이벤트, 상태전이의 설계



[그림 18-3] 단순한 캐릭터의 유한상태기계의 예.

상태 상수심볼 정의

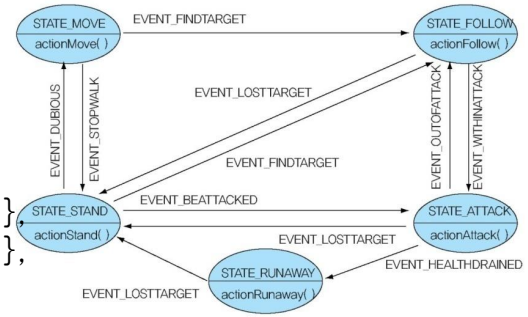
```
enum StateID {
        STATE_STAND, //서있는 상태.
        STATE_MOVE, //이동하는 상태.
        STATE_ATTACK, //공격하는 상태.
        STATE_RUNAWAY, //도망치는 상태.
        STATE_FOLLOW, //추격하는 상태.
};
```

이벤트 상수심볼 정의

```
enum EventID {
        EVENT_FINDTARGET, //목표물을 발견한 이벤트.
        EVENT_LOSTTARGET, //목표물을 놓친 이벤트.
        EVENT_BEATTACKED, //공격당한 이벤트.
        EVENT_HEALTHDRAINED, //체력이 소진된 이벤트.
        EVENT_STOPWALK, //걷기를 중지한 이벤트.
        EVENT_DUBIOUS, //수상함을 발견한 이벤트.
        EVENT_INTOATTACK, //공격가능한 거리로 진입한 이벤트.
        EVENT_OUTOFATTACK, //공격가능범위를 벗어난 이벤트.
};
```

유한상태기계 예제

- 상태, 이벤트, 상태전이의 설계



[그림 18-3] 단순한 캐릭터의 유한상태기계의 예.

적 캐릭터

```
DWORD stateTransitions[][3] = {
        { STATE_STAND, EVENT_FINDTARGET, STATE_FOLLOW },
        { STATE_STAND, EVENT_BEATTACKED, STATE_ATTACK },
        { STATE_STAND, EVENT_DUBIOUS, STATE_MOVE },
        { STATE_MOVE, EVENT_FINDTARGET, STATE_FOLLOW },
        { STATE_MOVE, EVENT_STOPWALK, STATE_STAND },
        { STATE_ATTACK, EVENT_LOSTTARGET, STATE_STAND },
        { STATE_ATTACK, EVENT_HEALTHDRAINED, STATE_RUNAWAY },
        { STATE_ATTACK, EVENT_OUTOFATTACK, STATE_FOLLOW },
        { STATE_FOLLOW, EVENT_WITHINATTACK, STATE_ATTACK },
        { STATE_FOLLOW, EVENT_LOSTTARGET, STATE_STAND },
        { STATE_RUNAWAY, EVENT_LOSTTARGET, STATE_STAND },
};
int numTransitions = 11;
```

주인공 캐릭터

```
DWORD stateTransitions[][3] = {
        { STATE_STAND, EVENT_FINDTARGET, STATE_MOVE },
        { STATE_MOVE, EVENT_STOPWALK, STATE_STAND },
};
int numTransitions = 2;
```

유한상태기계 예제

- 상태 구조체의 구현

상태 구조체 - 자료구조

```
struct State {
        DWORD stateID;
        std::map<DWORD, DWORD> transitions;
};
```

상태 구조체 – 함수들

```
struct State {
        DWORD stateID;
        std::map<DWORD, DWORD> transitions;
        State(DWORD _stateID) { stateID = _stateID; }
        ~State() { transitions.clear(); }
        void addTransition(DWORD inputEvent, DWORD outputStateID) {
                transitions[inputEvent] = outputStateID; }
        DWORD getOutputStateID(DWORD inputEvent) {
                std::map<DWORD, DWORD>::iterator iter =
                        transitions.find(inputEvent);
                if (iter==transitions.end()) return UNDEFINED; //등록된 전이가 없음.
                else return transitions[inputEvent];
        }
};
```

유한상태기계 예제

- 유한상태기계 클래스의 구현

```
class FiniteStateMachine
{
private:
        std::map<DWORD, State*> states;
        DWORD currentStateID;
public:
        void addStateTransition(DWORD stateID, DWORD inputEvent, DWORD outputStateID);
        void issueEvent(int event);
        DWORD getCurrentStateID() { return currentStateID; }
        void setCurrentStateID(DWORD stateID);
};

void FiniteStateMachine::addStateTransition( DWORD stateID, DWORD inputEvent, DWORD outputStateID )
{
        std::map<DWORD, State*>::iterator iter = states.find( stateID );
        if (iter == states.end()) states[stateID] = new State( stateID );
        states[stateID]->addTransition( inputEvent, outputStateID );
}

DWORD FiniteStateMachine::getCurrentStateID() { return currentStateID; }
void FiniteStateMachine::setCurrentStateID(DWORD stateID) {
        std::map<DWORD, State*>::iterator iter = states.find(stateID);
        if (iter == states.end()) return;
        currentStateID = stateID;
}

void FiniteStateMachine::issueEvent(int event) {
        if (currentStateID == UNDEFINED) return;
        DWORD outputStateID = states[currentStateID]->getOutputStateID(event);
        if (outputStateID == UNDEFINED) return;
        currentStateID = outputStateID; //상태 전이!!
}
```

유한상태기계 예제

- 캐릭터 클래스의 구현

캐릭터 클래스

```
class Character
{
private:
		FiniteStateMachine* stateMachine; //캐릭터의 유한상태기계.
		// 캐릭터 개별 특성을 위한 변수들.
		D3DXVECTOR3 position; //캐릭터의 현재 위치.
		D3DXVECTOR3 destPosition; //캐릭터의 목표지점 위치.
		float speed; //캐릭터의 이동속력.
public:
		void update(Character* target, float timeDelta);
private:
		void issueEvent(DWORD event);
		void actionMove(float timeDelta);
		//...그 외의 액션함수들은 생략.
};
```

캐릭터 클래스의 생성자에서..

```
FiniteStateMachine* stateMachine = new FiniteStateMachine();
for (int i=0; i<numTransitions; i++) {
		stateMachine->addStateTransition( stateTransitions[i][0],
				stateTransitions[i][1], stateTransitions[i][2] );
}
stateMachine->setCurrentStateID(STATE_STAND); //초기상태를 지정함.
```

유한상태기계 예제

- 캐릭터 클래스의 구현'

응용프로그램에서 주기적으로 **update()**를 호출..

```
void Character::update(Character* target, float timeDelta) {
        switch (stateMachine->getCurrentStateID()) {
        case STATE_MOVE:
                actionMove(timeDelta); //액션 수행!!
                if (isVisible(target->getPosition())) {
                        setDestPosition(target->getPosition());
                        issueEvent(EVENT_FINDTARGET); //이벤트 발생시킴!!
                }
                break;
        case STATE_STAND:
                //...그 외의 상태들에 대한 처리는 생략.
        }
}

void Character::actionMove(float timeDelta) {
        // position에서 destPosition으로 speed의 속력으로 이동함.
}

void Character::issueEvent(DWORD event) {
        stateMachine->issueEvent(event);
        switch (stateMachine->getCurrentStateID()) {
        case STATE_MOVE: /* 캐릭터를 움직이는 애니메이션을 시작시킴. */ break;
        //...그 외의 처리에 대해서 생략.
        }
}
```

유한상태기계 예제

- 응용프로그램의 구현

응용프로그램의 초기화 함수에서 여러 적군 캐릭터와 한 주인공 캐릭터를 생성..

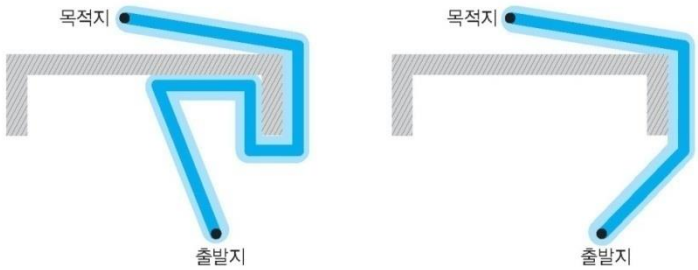
```
std::vector<Character*> enemyCharacters;
//생성 시에 속성값(이동 속력이나 가시거리 등)들을 조금씩 다르게 지정
Character* myCharacter = NULL;
```

주기적으로 **update()** 호출..

```
bool update(float timeDelta) {
        for (std::vector <Character*>::iterator iter = enemyCharacters.begin();
                                        iter != enemyCharacters.end(); iter++ )    {
                Character* ch = *iter;
                ch->update(myCharacter, timeDelta);
        }
        myCharacter->update(myCharacter, timeDelta);
        return true;
}
```

길찾기

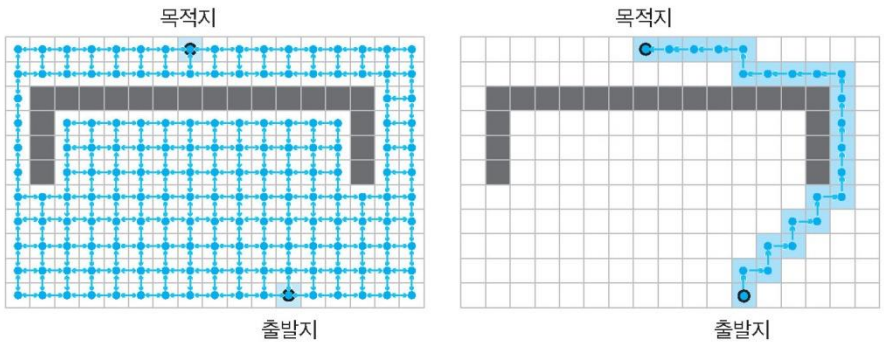
- 길찾기 알고리즘
  - 길찾기 문제
    - 목표 지점까지 가는 최단경로를 찾기
    - 비용 최소화 문제
  - 격자 기반 방법이 현실적임
    - 월드공간을 격자로 분할
  - 그래프
    - 노드 = 격자의 각 셀
    - 링크 = 이동 가능한 이웃 셀과의 연결



[그림 18-4] 길찾기 문제.

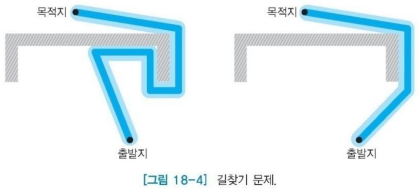
길찾기

- 그래프에서 최단경로 찾기



[그림 18-5] 격자맵에 대한 최단경로 문제.

  - Dijkstra의 알고리즘
    - 방법: 출발지 노드에서 방문하지 않은 노드를 하나씩 방문하면서 목적지 노드까지 진행. 새 노드를 방문할 때에는 시작 노드에서 가장 가까운 노드를 우선적으로 방문함.
    - 항상 최단 경로를 제공함
    - 매우 느림



[그림 18-4] 길찾기 문제.

  - BFS(best-first-search) 알고리즘
    - 방법: 휴리스틱(heuristic)이라는 추정치 값을 사용하는 것 외에는 Dijkstra의 알고리즘과 유사. 시작 노드에서부터 현재 노드까지의 비용 대신 현재 노드에서 목적지 노드까지의 휴리스틱을 고려함.
      - 휴리스틱이란? 휴리스틱이란 현재의 위치에서 목적지까지의 비용을 추정한 값임.
    - 최단경로가 아닐 수 있음
    - 빠름

A* 길찾기 알고리즘

- A* 길찾기 알고리즘
  - Dijkstra 알고리즘 + BFS 알고리즘 – 항상 최단 경로를 제공함, 빠름
- 비용 : f(전체 비용) = g + h
  - g = 시작 노드에서부터 현재 노드까지의 실제 비용 (확정된 값)
    - g는 흔히 경로의 연결된 노드들 간의 거리의 합으로 정의
  - h = 현재의 노드에서 목적지 노드까지의 비용의 추정치
    - 휴리스틱: Manhattan 거리, Euclidean 거리, Minkowski 거리
      - Manhattan 거리 $h = |x - x'| + |y - y'|$
        - » 도시블록(city-block) 거리 또는 L_1 거리라고도 함
      - Euclidean 거리 $z = \sqrt{(x-x')^2 + (y-y')^2}$
      - Minkowski 거리: 일반화된 표현 $(|x-x'|^m + |y-y'|^m)^{1/m}$
- 노드의 표현
  - 각 노드는 구조체로 표현
    - 각 노드 구조체의 필드: 비용을 위한 g,h,f 필드 및 부모노드의 기억을 위한 parent 필드
      - 시작 노드의 parent 필드는 NULL임.

A* 길찾기 알고리즘

- **A* 알고리즘의 구현**
  - 시작 위치 노드: ns, 목표 위치 노드: ng
    - ns 초기화 코드 :

```
ns.g = 0;
ns.h = computeHeuristic(ns, ng);
ns.f = ns.g + ns.h;
ns.parent = NULL;
```

  - 휴리스틱 h를 구하는 함수: computeHeuristic()
    - 두 노드의 Manhattan 거리를 리턴
  - 데이터구조 준비: 2개의 노드 리스트
    - Q: 열린 리스트: 아직 탐색하지 않은 노드들의 리스트
    - C: 닫힌 리스트: 이미 탐색한 노드들의 리스트
    - Q,C 초기화 코드 :

```
C.clear();
Q.clear();
Q.add(ns);
```

A* 길찾기 알고리즘

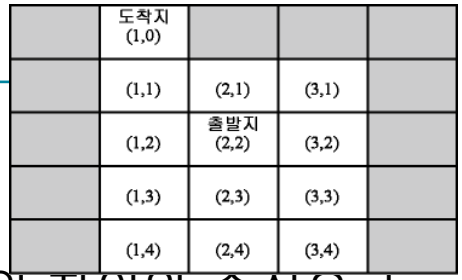
- **A* 알고리즘의 구현'**
  - 열린 리스트 Q가 비어있지 않는 동안 while 문을 반복 :

```
while (!Q.isEmpty()) {
        n = Q.removeFirstNode(); //n은 f가 가장 작은 노드
        if (n == ng) { return ( makePath(n) ); }
        for each m in successors(n) do {
                /* m에 대한 처리 */
        }
        C.add(n);
}
/* 경로를 발견할 수 없음. */
```

  - /* m에 대한 처리 */

```
new_g = n.g + cost(n,m);
if ( (Q.contains(m,&prev_g) || C.contains(m,&prev_g)) && prev_g <= new_g) {
        continue; /* 이번 m을 무시하고 다음 m을 처리하도록 for루프로 진행함. */
}
//add m to Q
if (C.contains(m)) C.remove(m);
if (Q.contains(m)) Q.remove(m);
m.parent = n; //부모노드로의 링크 설정.
m.g = new_g; //n.g+cost(n,m)임.
m.h = computeHeuristic(m); //휴리스틱 함수.
m.f = m.g+m.h;
Q.add(m); //f값이 오름차순이 되도록 추가
```

A* 알고리즘 적용 예



- **알고리즘 적용 예**
  - 출발지점 노드는 (2, 2)이고, 도착지점 노드는 (1, 0)
  - 휴리스틱 h : 도시블록 거리 (노드 위치의 수평으로의 차이와 수직으로의 차이의 합)
    - 즉 수직 방향이나 수평 방향으로의 이웃 노드로의 거리는 1이고 대각선 방향으로의 이웃 노드로의 거리는 2
  - 초기화:
    - Q={**(2,2)**}
      - 이제 Q의 각 노드의 g값은 0이고 f값은 3임.
  - While반복1:
    - Q에서 첫 번째 노드 **(2,2)**를 꺼내서 n에 지정
    - n에 연결된 이웃 노드들인 (2,1),(1,2),(3,2),(2,3)들에 대해서 각각 for 문을 반복
      - 모두 새로운 노드들이므로 Q에 추가하여 Q={**(2,1)**,**(1,2)**,**(3,2)**,**(2,3)**}가 된다.
      - 이제 (2,2)를 처리하였으므로 C에 추가하여 C={(2,2)}가 된다.
      - 이제 Q의 각 노드의 g값은 모두 1이고 f값은 각각 3,3,5,5이다. C의 각 노드의 g값은 0이다.

A* 알고리즘 적용 예

- **알고리즘 적용 예**
  - While반복2:
    - Q에서 첫 번째 노드 **(2,1)**을 꺼내서 n에 지정
    - n에 연결된 이웃 노드들인 (1,1),(3,1),(2,2)들에 대해서 각각 for 문을 반복
      - (1,1)과 (3,1)은 처음 방문하는 노드이므로 Q에 추가한다.
        - » g값은 모두 2이고 f값은 각각 3,5이다.
      - (2,2)는 C에 이미 있으므로 C에 있는 (2,2)의 g값인 0보다 새로 계산된 g값인 2가 더 크므로 무시한다.
      - 이제 Q={(1,2),**(1,1)**,(3,2),(2,3),**(3,1)**}이고 C={(2,2),(2,1)} 이다.
        - » F가 오름차순이 되도록 삽입 (동일하면 기존 뒤에 추가한다고 가정)
      - 이제 Q의 각 노드의 g값은 1,2,1,1,2이고 f값은 각각 3,3,5,5,5 이다. C의 g값은0,1 이다.
  - While반복3:
    - Q에서 첫 번째인 **(1,2)**를 꺼내서 n에 지정
    - n에 연결된 이웃 노드들인 (1,1),(2,2),(1,3)들에 대해서 for 문을 반복
      - (1,1)은 이미 Q에 있고 Q에 있는 노드의 g값이 2이고 새로 계산한 g값도 2이므로 무시한다.
      - (2,2)는 이미 C에 있고 C에 있는 노드의 g값이 0이고 새로 계산한 g값이 2이므로 무시한다.
      - (1,3)는 새 노드이므로 Q에 추가한다.
        - » g값이 2이고, f값이 5이다.
      - 이제 Q={(1,1),(3,2),(2,3),(3,1),**(1,3)**}이고 C={(2,2),(2,1),(1,2)}이다.
      - 이제 Q의 각 노드의 g값은 2,1,1,2,2이고 f값은 각각 3,5,5,5,5 이다. C의 g값은0,1,1 이다.

A* 알고리즘 적용 예

- **알고리즘 적용 예**
  - While반복4:
    - Q에서 첫 번째인 **(1,1)**을 꺼내서 n에 지정
    - n에 연결된 이웃 노드들인 (1,0),(2,1),(1,2)들에 대해서 for 문을 반복
      - (1,0)은 새 노드이므로 Q에 추가
        - » g값이 3이고 f값이 3이다
      - (2,1)과 (1,2)는 모두 C에 있으며 기존의 g값인 1이 새로 계산된 g값인 3보다 작으므로 무시한다.
      - 이제 Q={**(1,0)**,(3,2),(2,3),(3,1),(1,3)} 이고 C={(2,2),(2,1),(1,2),(1,1)} 이다.
      - 이제 Q의 각 노드의 g값은 3,1,1,2,2이고 f값은 각각 3,5,5,5,5 이다. C의 g값은 0,1,1,2 이다.
  - While반복5:
    - Q에서 첫 번째인 **(1,0)**을 꺼내서 n에 지정
    - n은 목적지 노드이므로 경로를 찾았다.
    - 이제 경로를 출력하자.
      - » (1,0)의 parent 값이 (1,1)이고,
      - » (1,1)의 parent 값이 (2,1)이고,
      - » (2,1)의 parent 값이 (2,2)이고,
      - » (2,2)의 parent 값이 NULL이다.
    - 따라서 최종적인 경로는 <(2,2), (2,1), (1,1), (1,0)> 으로 구해진다.

길찾기 예제

- 예제

02.PathFindingConsole
[PathFindingMaze]