**Artificial Intelligence**

**assignment #1**

Hanyang University

Computer Science(컴퓨터전공)

2012004087

이기준

**코드설명**

$assignment1\_2012004087\_IDS.cpp

int Go\_Up(int &R, int &C);

int Go\_Down(int &R, int &C);

int Go\_Left(int &R, int &C);

int Go\_Right(int &R, int &C);

int(\*Go[4])(int&, int&) = { Go\_Up, Go\_Left, Go\_Down, Go\_Right };

4 방향으로 이동할 수 있는 함수들이고, 그 함수들을 관리하는 함수포인터

void WHereIsStartPoint(int& R, int &C);

void WHereIsFinishPoint(vector<pair<int, int> >& finishPoint);

각각 출발점(1개)와 도착점(1개 이상)을 알려주는 함수

int \*\* visits;

위치에 방문을 했는지 알려주는 전역 배열

pair<int, int> \*\* trace\_MATRIX;

최적경로를 역추적할 수 있도록 이전 좌표를 알려주는 전역배열

pair<int, int> IDS(int &R, int &C);

IDS 알고리즘으로 최적경로 탐색해주는 함수

typedef struct \_\_stackElement {

int row;

int col;

int dir;

int limit;

}stackElement;

IDS 알고리즘은 DFS를 depth의 한계를 두고 depth를 하나씩 늘려서 탐색을 한다. 결국 DFS이기 때문에 내부적으로 stack을 사용하고 그 stack에 담을 구조체를 정의하였다.

Row, col은 각각 행,열 값이고 dir은 방향성을 나타낸다. 0 = up, 1= left , 2= down , 3= right

Limit은 출발점으로부터 이동한 수이다. Limit이 0이되면 더 이상 진행할 수 없다.

IDS 함수 상세

pair<int, int> IDS(int& R, int& C)

{

int cur\_R, cur\_C, cur\_dir, cur\_limit;

int tmp\_R, tmp\_C;

int limit = 0;

int time = 0;

stack<stackElement> S;

stackElement tmpStackElement;

while (true)

{

limit++;

cur\_limit = limit;

cur\_R = R;

cur\_C = C;

cur\_dir = -1;

//시작점 집어넣기

for (int i = 0; i < 4; ++i) {

tmp\_R = cur\_R;

tmp\_C = cur\_C;

if (Go[i](tmp\_R, tmp\_C)) {

tmpStackElement = { tmp\_R, tmp\_C, i, cur\_limit - 1 };

S.push(tmpStackElement);

trace\_MATRIX[tmp\_R][tmp\_C] = make\_pair(R, C);

}

}

while (!S.empty()) {

//pop 과정

cur\_R = S.top().row;

cur\_C = S.top().col;

cur\_dir = S.top().dir;

cur\_limit = S.top().limit;

visits[cur\_R][cur\_C] = cur\_limit;

S.pop();

//도착확인

if (MATRIX[cur\_R][cur\_C] == 4)

{

R = cur\_R;

C = cur\_C;

return make\_pair(trace.size(), time);

}

//시도(time)증가

time++;

// 스텍에 가능한 이동 집어넣기

if (cur\_limit != 0) {

for (int i = 0; i < 4; ++i) {

if (cur\_dir == (i + 2) % 4)

continue;

tmp\_R = cur\_R;

tmp\_C = cur\_C;

int gogo = Go[i](tmp\_R, tmp\_C);

if (gogo == 1 || (gogo == -1 && cur\_limit > visits[tmp\_R][tmp\_C])) {

tmpStackElement = { tmp\_R, tmp\_C, i, cur\_limit - 1 };

S.push(tmpStackElement);

trace\_MATRIX[tmp\_R][tmp\_C] = make\_pair(cur\_R, cur\_C);

}

}

}

}

}

}

출발점에서 갈 수 있는 모든 경로를 스택에 먼저 담는다. 그 이후 while반복문에서 DFS탐색을 한다. Depth만큼 이동하였지만 결과를 찾지 못하면 depth값을 1증가시킨뒤 처음부터 다시 시작한다.

$assignment1\_2012004087\_GBS.cpp

기본적인 부분은 IDS와 동일하며 새롭게 추가된 부분만 설명한다.

int \*\* HU\_MATRIX;

각각 공간에 휴리스틱을 정의하였다.벽이나 출발점은 100000으로 지정하였고, 통로는 가장 가까운 도착점과의 맨하탄 거리를 값으로 주었다.

typedef struct \_\_pqElement {

int row;

int col;

int ex\_row;

int ex\_col;

int dir;

int distance;

int hu;

bool operator<(const \_\_pqElement & rhs) const

{

return hu> rhs.hu;

}

}pqElement;

GBS알고리즘은 우선순위 큐를 통하여 구현한다. 빠르게 도착점을 찾을 수 있지만 최적경로를 찾지 못할 수 있는 단점이 있다. Row ,col은 현재 행열이며, ex\_row, ex\_col은 추적에 필요한 이동전 좌표를 가르킨다. Dir 은 방향, distance는 출발점으로부터 이동 수다. 그리고 operator를 통해 min heap 처럼 휴리스틱값이 낮은것부터 pop할 수 있게 하였다.

GBS 함수 상세

pair<int, int> GBS(int &R, int &C)

{

int cur\_R, cur\_C, cur\_dir, cur\_walk;

int tmp\_R, tmp\_C;

int time = 0;

int walk = 100000;

priority\_queue<pqElement> PQ;

pqElement tmp\_pqElement;

for (int i = 0; i < 4; ++i) {

tmp\_R = R;

tmp\_C = C;

if (Go[i](tmp\_R, tmp\_C) == 1) {

tmp\_pqElement = { tmp\_R, tmp\_C, R,C,i, walk, HU\_MATRIX[tmp\_R][tmp\_C]};

PQ.push(tmp\_pqElement);

}

}

while (!PQ.empty()) {

//pop 과정

cur\_R = PQ.top().row;

cur\_C = PQ.top().col;

tmp\_R = PQ.top().ex\_row;

tmp\_C = PQ.top().ex\_col;

cur\_dir = PQ.top().dir;

cur\_walk = PQ.top().distance;

PQ.pop();

if (visits[cur\_R][cur\_C] < cur\_walk)

visits[cur\_R][cur\_C] = cur\_walk;

else

continue;

trace\_MATRIX[cur\_R][cur\_C] = make\_pair(tmp\_R, tmp\_C);

//도착확인

if (MATRIX[cur\_R][cur\_C] == 4)

{

// 끝점 반환

R = cur\_R;

C = cur\_C;

return make\_pair(0, time);

}

//시도(time)증가

time++;

// 스텍에 가능한 이동 집어넣기

for (int i = 0; i < 4; ++i) {

if (cur\_dir == (i + 2) % 4)

continue;

tmp\_R = cur\_R;

tmp\_C = cur\_C;

int gogo = Go[i](tmp\_R, tmp\_C);

// if(Go[i](tmp\_R,tmp\_C) == 1){

if (gogo == 1) {

tmp\_pqElement = { tmp\_R, tmp\_C, cur\_R,cur\_C,i, cur\_walk - 1, HU\_MATRIX[tmp\_R][tmp\_C] };

PQ.push(tmp\_pqElement);

}

}

}

return make\_pair(0, 0);

}

구현 방식은 위의 IDS와 전반적으로 비슷하지만 스택을 사용하지 않고 우선순위 큐를 통하여 휴리스틱값이 낮은 값부터 pop한다.

$assignment1\_2012004087\_ASS.cpp

기본적인 부분은 GBS와 동일하며 새롭게 추가된 부분만 설명한다.

typedef struct \_\_pqElement {

int row;

int col;

int ex\_row;

int ex\_col;

int dir;

int distance;

int hu;

bool operator<(const \_\_pqElement & rhs) const

{

return (distance < rhs.distance) || (distance == rhs.distance && hu> rhs.hu);

}

}pqElement;

이동한 거리가 우선적으로 최소여야 하며, 그다음 휴리스틱값과 출발점으로부터 실제거리의 합이 낮아야 pop할 수 있다.

int \*\* A\_MATRIX;

휴리스틱 값과 종합해서 평가하게 될 값이다.출발점으로부터 현 위치까지의 실제거리를 담았다.

ASS함수 상세

int ASS(int &R, int &C)

{

int cur\_R, cur\_C, cur\_dir, cur\_walk;

int tmp\_R, tmp\_C;

int time = 0;

int walk = 100000;

priority\_queue<pqElement> PQ;

pqElement tmp\_pqElement;

for (int i = 0; i < 4; ++i) {

tmp\_R = R;

tmp\_C = C;

if (Go[i](tmp\_R, tmp\_C) == 1) {

tmp\_pqElement = { tmp\_R, tmp\_C, R,C,i, walk, HU\_MATRIX[tmp\_R][tmp\_C] + A\_MATRIX[tmp\_R][tmp\_C] };

PQ.push(tmp\_pqElement);

// trace\_MATRIX[tmp\_R][tmp\_C] = make\_pair(R, C);

}

}

while (!PQ.empty()) {

//pop 과정

cur\_R = PQ.top().row;

cur\_C = PQ.top().col;

tmp\_R = PQ.top().ex\_row;

tmp\_C = PQ.top().ex\_col;

cur\_dir = PQ.top().dir;

cur\_walk = PQ.top().distance;

PQ.pop();

if (visits[cur\_R][cur\_C] < cur\_walk)

visits[cur\_R][cur\_C] = cur\_walk;

else

continue;

trace\_MATRIX[cur\_R][cur\_C] = make\_pair(tmp\_R, tmp\_C);

//도착확인

if (MATRIX[cur\_R][cur\_C] == 4)

{

// 끝점 반환

R = cur\_R;

C = cur\_C;

return time;

}

//시도(time)증가

time++;

// 스텍에 가능한 이동 집어넣기

for (int i = 0; i < 4; ++i) {

if (cur\_dir == (i + 2) % 4)

continue;

tmp\_R = cur\_R;

tmp\_C = cur\_C;

int gogo = Go[i](tmp\_R, tmp\_C);

// if(Go[i](tmp\_R,tmp\_C) == 1){

if (gogo == 1) {

tmp\_pqElement = { tmp\_R, tmp\_C, cur\_R,cur\_C,i, cur\_walk - 1, HU\_MATRIX[tmp\_R][tmp\_C] + A\_MATRIX[tmp\_R][tmp\_C] };

PQ.push(tmp\_pqElement);

}

else if ((gogo == -1) && (visits[tmp\_R][tmp\_C] < cur\_walk - 1)) {

tmp\_pqElement = { tmp\_R, tmp\_C, cur\_R,cur\_C,i, cur\_walk - 1, HU\_MATRIX[tmp\_R][tmp\_C] + A\_MATRIX[tmp\_R][tmp\_C] };

PQ.push(tmp\_pqElement);

}

}

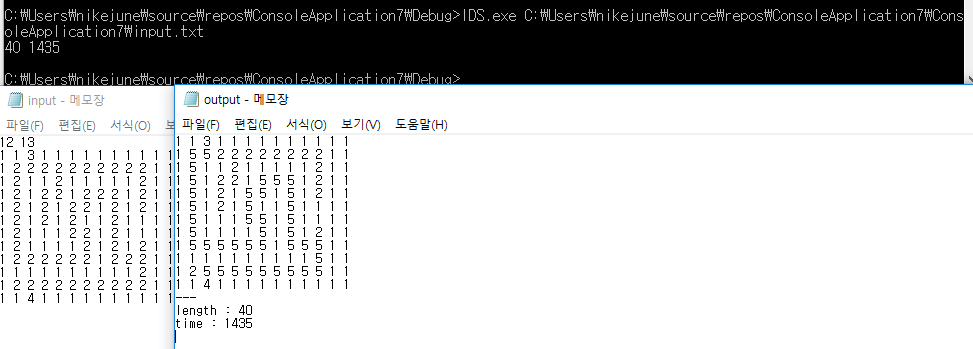
}

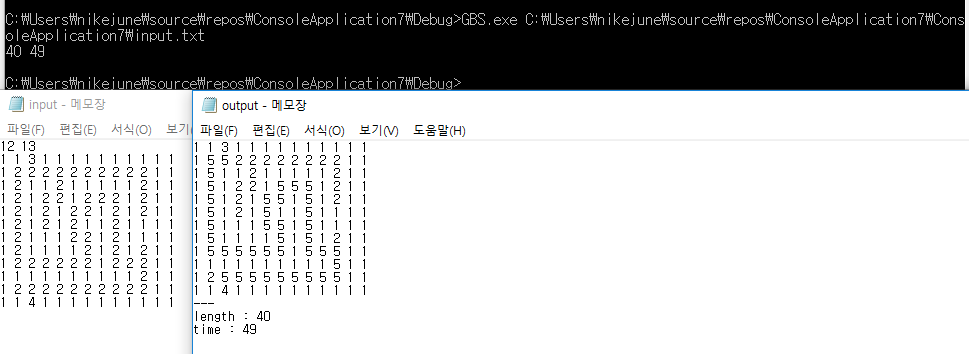
return -1;

}

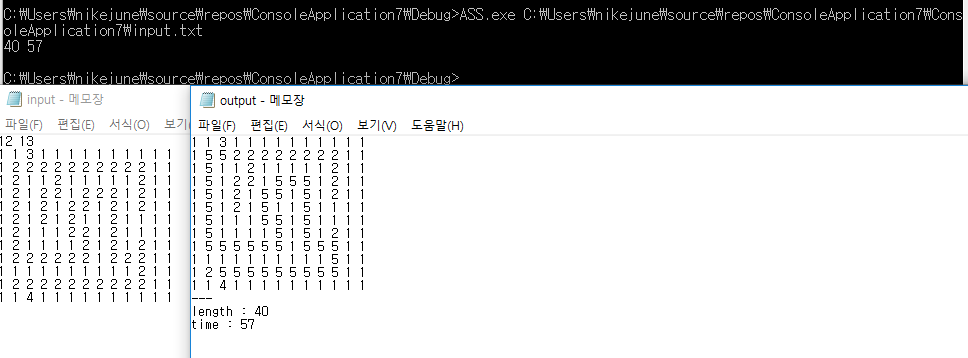
**실험결과**

Input\_ex1.txt

IDS

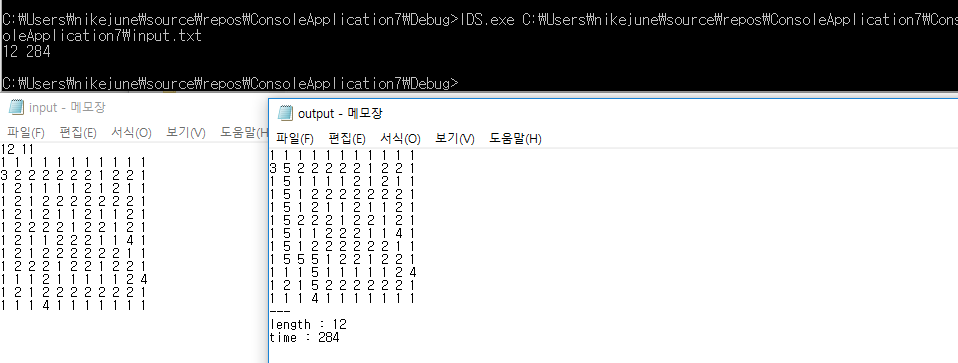
**GBS**

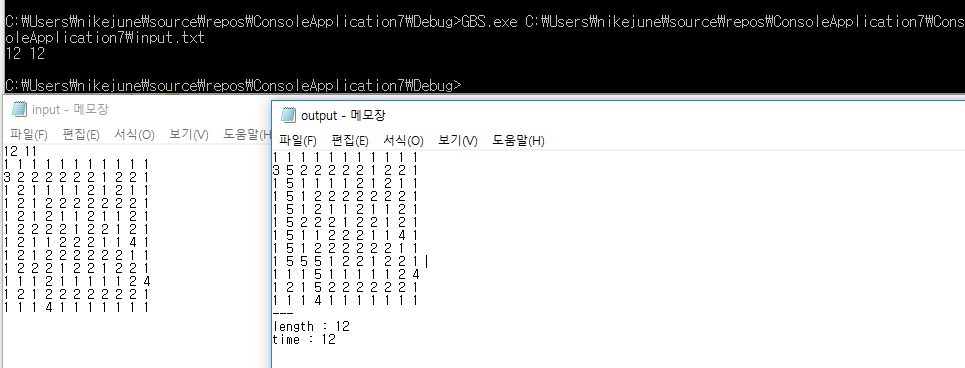
**ASS**

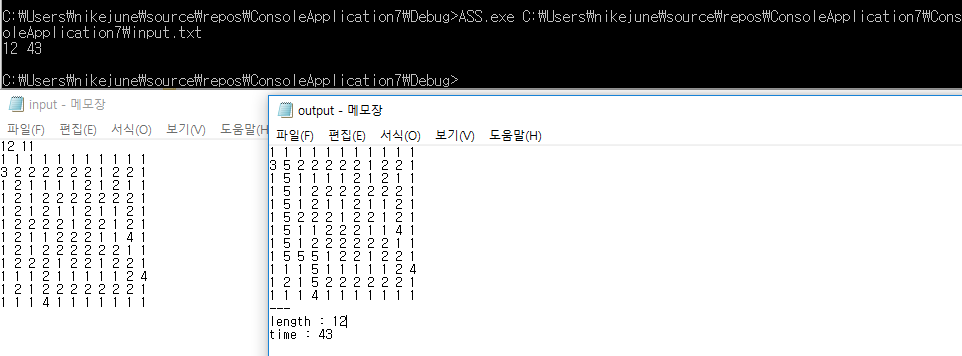


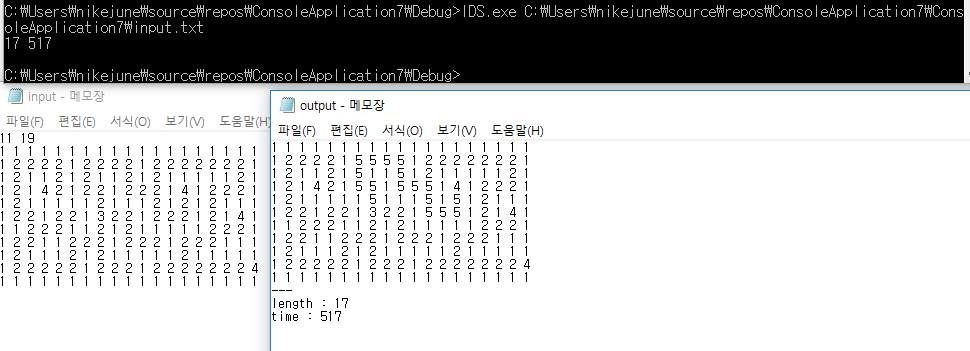
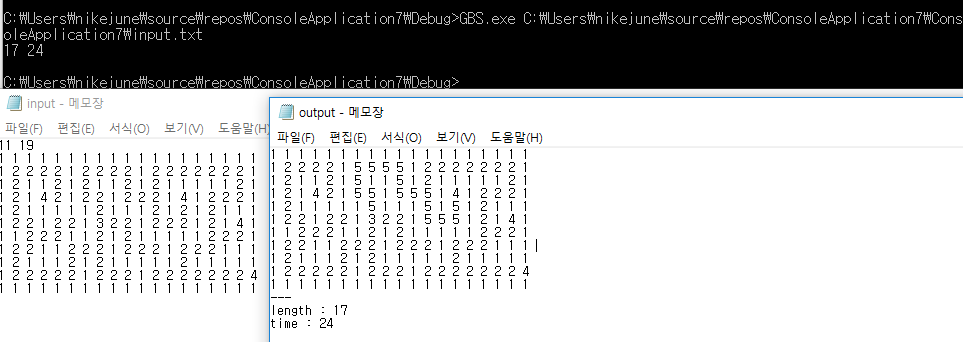
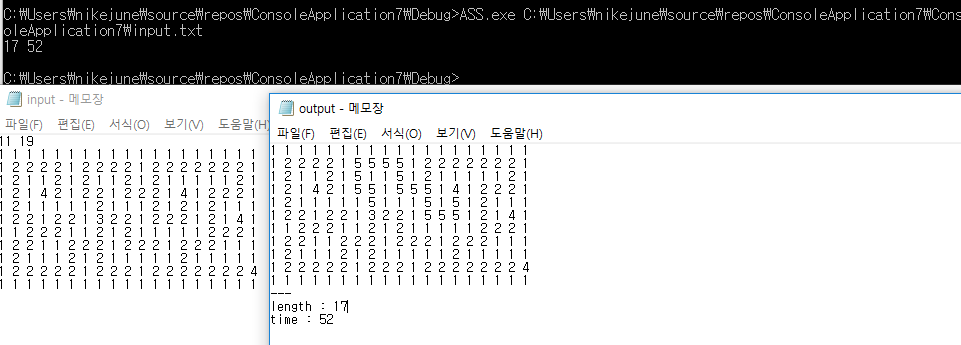
Input\_ex2.txt

IDS



GBS 

ASS

Input\_ex3.txt  
IDSGBSASS

**IDS**는 optimal하지만 반복적으로 탐색을 진행하기 때문에 time수가 GBS,ASS에 비해 많은 연산을 함을 알 수 있다.  
**GBS**의 time은 IDS, ASS에 비해 매우 작다. 하지만 이 경로는 optimal하지 않을 수 있다. 도착점이 2개 이상일 경우 가깝지 않은 도착점을 알려 줄 때도 있다.

**ASS**는 optimal하면서도 time수가 작은 편이다.

**해당 코드에 대한 컴파일 방법 및 사용 버전**

**OS : Windows 10 / IDE & compile : Visual Studio 2017**

**Language : C++**