## **[159740 Studies in Intelligent Systems (159740\_2017\_S2FS\_AKLI)](http://stream.massey.ac.nz/course/view.php?id=33073" \o "159740 Studies in Intelligent Systems)**

**Report of Assignment 1**

**Yonghui Rao 16093645**

In this assignment, LPA\* and D\* Lite algorithms are implemented, this report demonstrates the details of this assignment, the contents include:

#1 Data Structures Explanation

#2 Skeleton C++ code

#3 Snapshots

#4 Experiments Result

#5 Discussion of the Result

#6 User’s Guide

**#1 Data Structures Explanation**

**LPA\* and D\* Lite share the same C++ cell structure:**

struct LpaStarCell

{

LpaStarCell\* move[DIRECTIONS];

set<int> preds; // Used for LPA\*, save all its neighbours which are its predessors

set<int> succs; // Used for LPA\*, save all its neighbours which are its successors

LpaStarCell \*pathPred; // Used for LPA\*, if robot is at cell c, c->pathPred means the cell that the robot should move to in next step

LpaStarCell \*pathSucc; // Used for D\* Lite, if robot is at cell c, c->pathSucc means the previous location of the robot

double linkCost[DIRECTIONS];

int x, y;

double g;

double rhs;

double h;

double key[2];

bool accessed;

bool expanded;

//---------------------

//TYPE: 0 - traversable, 1 - blocked, 9 - unknown, 6 - start vertex, 7 - goal vertex

char type;

//----------------------

};

**Use below line to reuse this structure for D\* Lite:**

typedef LpaStarCell DstarLiteCell;

**LpaStar and DStarLite are class names of the two algorithms. Define the priority queue as a vector:**

vector<LpaStarCell\* > U; //Priority Queue for LPA\*

vector<DstarLiteCell\* > U; //Priority Queue for D\* Lite

**Make the priority queue as a heap and specify the compare method:**

inline bool heapCmp(LpaStarCell \*cell1, LpaStarCell \*cell2)

{

if ((cell1->h + cell1->rhs) > (cell2->h + cell2->rhs))

{

return true;

}

return false;

}

**The important functions of LpaStar:**

bool computeShortestPath(); // Main function for compute shortest path based on current grid.

bool blockCell(int, int); // After the 1st search, user could block one cell so that it could perform replanning.

void disablePathViaCell(LpaStarCell \*cell); // If one cell is blocked, disable the cells whose information are out-dated.

**The important functions of D\* Lite:**

bool computeShortestPath(); // Compute shortest path based on current grid.

void disablePathViaCell(DstarLiteCell \*cell); // If robot detects one cell is blocked, disable the cells whose information areout-dated.

bool dstarliteMain(int startX, int startY, int goalX, int goalY); // Main function for D\* Lite, keep moving the robot and calculating path.

**#2 Skeleton C++ code**

**For LPA\*, functions computeShortestPath and blockCell are shown:**

// Main fuction to compute shortest path

bool LpaStar::**computeShortestPath**()

{

runningSerach = true;

int goalY = goal->y;

int goalX = goal->x;

LpaStarCell \*goal = &maze[goalY][goalX];

while (1)

{

if (U.empty())

{

cout << "U is empty" << endl;

if(goal->rhs == goal->g && goal->rhs < INF)

return true;

return false;

}

LpaStarCell \*topU = U.front();

calcKey(goal);

if (compareKey(goal, topU) || (goal->rhs != goal->g && goal->rhs < INF))

{

pop\_heap(U.begin(), U.end(),heapCmp);

U.pop\_back();

topU->expanded = true;

if (topU->g > topU->rhs)

{

topU->g = topU->rhs;

for (int i = 0; i < 8; i++)

{

LpaStarCell \*neighbour = topU->move[i];

if (neighbour && traversable(neighbour))

{

for (int j = 0; j < 8; j++)

{

if (neighbour->move[j] == topU)

{

neighbour->preds.insert(j);

break;

}

}

updateVertex(neighbour);

}

}

}

else {

topU->g = INF;

for (int i = 0; i < 8; i++)

{

LpaStarCell \*neighbour = topU->move[i];

if (neighbour && traversable(neighbour))

{

for (int j = 0; j < 8; j++)

{

if (neighbour->move[j] == topU)

{

neighbour->preds.insert(j);

break;

}

}

updateVertex(neighbour);

}

}

updateVertex(topU);

}

}

else {

break;

}

}

return true;

}

// Mark the maze cell be blocked, return true if success, else false

bool LpaStar::**blockCell**(int i, int j)

{

LpaStarCell \*blockCell = &maze[i][j];

if (blockCell->type == '1')

{

return false;

}

cout << "block cell:" << j << " " << i << endl;

blockCell->type = '1';

blockCell->g = INF;

blockCell->rhs = INF;

disablePathViaCell(blockCell);

for (int i = 0; i < 8; i++)

{

LpaStarCell \*neighbour = blockCell->move[i];

if (neighbour->type == '1')

continue;

if (neighbour != NULL)

{

// Remove the blockCell from the pred set of neighbour

set<int>::iterator iter = neighbour->preds.begin();

while (iter!= neighbour->preds.end())

{

int predIndex = \*iter;

if (neighbour->move[predIndex] == blockCell)

{

neighbour->preds.erase(iter);

break;

}

iter++;

}

blockCell->linkCost[i] = INF;

for (int j = 0; j < 8; j++)

{

if (neighbour->move[j] == blockCell)

{

neighbour->linkCost[j] = INF;

}

}

updateVertex(neighbour);

}

}

return true;

}

**For D\* Lite, functions computeShortestPath and dstarliteMain are shown:**

// Main fuction for computing shortest path based on current knowledge

bool DStarLite::**computeShortestPath**()

{

while (1)

{

if (U.empty())

{

if (start->rhs == start->g && start->rhs < INF)

return true;

return false;

}

DstarLiteCell \*topU = U.front();

calcKey(start);

if (compareKey(topU->key, start->key) || (start->rhs != start->g))

{

double newKey[2];

double \*oldKey = topU->key;

calcKey(topU, newKey);

pop\_heap(U.begin(), U.end(), heapCmp);

U.pop\_back();

if (compareKey(oldKey, newKey))

{

topU->expanded = true;

calcKey(topU);

insertToU(topU);

push\_heap(U.begin(), U.end(), heapCmp);

}

else if (topU->g > topU->rhs) {

topU->expanded = true;

topU->g = topU->rhs;

for (int i = 0; i < 8; i++)

{

DstarLiteCell \*neighbour = topU->move[i];

if (neighbour && traversable(neighbour))

{

for (int j = 0; j < 8; j++)

{

if (neighbour->move[j] == topU)

neighbour->succs.insert(j);

}

updateVertex(neighbour);

}

}

}

else {

topU->expanded = true;

topU->g = INF;

for (int i = 0; i < 8; i++)

{

DstarLiteCell \*neighbour = topU->move[i];

if (neighbour && traversable(neighbour))

{

for (int j = 0; j < 8; j++)

{

if (neighbour->move[j] == topU)

neighbour->succs.insert(j);

}

updateVertex(neighbour);

}

}

updateVertex(topU);

}

}

else {

break;

}

}

return true;

}

// Main function for 1st search and replanning

bool DStarLite::**dstarliteMain**(int startX, int startY, int goalX, int goalY)

{

initialise(startX, startY, goalX, goalY);

start = &maze[startY][startX];

originStart = start;

goal = &maze[goalY][goalX];

// At the beginning the robot is at the start vertex, the neighbors of the start vertex are detected.

for (int i = 0; i < 8; i++)

{

DstarLiteCell \*neighbour = start->move[i];

if (neighbour && neighbour->type == '8')

neighbour->type = '0'; // We know it is traverable now.

// Block its neighbors whose type is 9, update all edges related to this neighbour.

if (neighbour && (neighbour->type == '9'))

{

neighbour->type = '1'; // We know it is blocked now.

for (int j = 0; j < 8; j++)

{

neighbour->linkCost[j] = INF;

DstarLiteCell \*move = neighbour->move[i];

for (int q = 0; q < 8; q++)

{

if (move->move[q] == neighbour)

move->linkCost[q] = INF;

}

}

}

}

last = &maze[startY][startX];

initStatistic();

resetCellsStatus();

cout << " begin first search..." << endl;

long long t1 = milliseconds\_now();

if (!computeShortestPath())

{

cout << "error: first search failed..." << endl;

return false;

}

long long t2 = milliseconds\_now();

cout << "1st seatch for shortest path finished, time used: " << (t2 - t1) << " ms" << endl;

statCellsStatus(numberOfExpandedStates, numberOfVertexAccesses);

showStatistic();

// Show the first search result

updateUI();

DstarLiteCell \*ss = &maze[2][4];

DstarLiteCell \*newStart = NULL;

while (start != goal)

{

float min\_c\_g = INF;

if (start->succs.size() == 0)

{

cout << "error, no succ" << endl;

return false;

}

set<int>::iterator iter = start->succs.begin();

newStart = NULL;

while (iter != start->succs.end())

{

int succIndex = \*iter;

DstarLiteCell \*succ = start->move[succIndex];

if (succ && succ->type != '1' && succ->type != '9' && start->linkCost[succIndex] < INF)

{

if (min\_c\_g > succ->g + start->linkCost[succIndex])

{

min\_c\_g = succ->g + start->linkCost[succIndex];

newStart = succ;

}

if (newStart == NULL)

newStart = succ;

}

iter++;

}

start = newStart;

cout << "move one step by inputting any key:" << endl;

getch();

updateUI();

cout << "Move to new start: " << start->y << " " << start->x << endl;

bool edgeChanged = false;

for (int i = 0; i < 8; i++)

{

DstarLiteCell \*neighbour = start->move[i];

if (neighbour && neighbour->type == '9')

{

edgeChanged = true;

}

if (neighbour && neighbour->type == '8')

{

neighbour->type = '0';

}

}

if (!edgeChanged)

{

continue;

}

km = km + calc\_H(last, start);

for (int i = 0; i < 8; i++)

{

DstarLiteCell \*neighbour = start->move[i];

if (neighbour && neighbour->type == '9')

{

DstarLiteCell \*newBlockCell = neighbour;

newBlockCell->type = '1'; // We know it is blocked now, update all related edges

newBlockCell->g = INF;

newBlockCell->rhs = INF;

disablePathViaCell(newBlockCell);

for (int j = 0; j < 8; j++)

{

newBlockCell->linkCost[j] = INF;

DstarLiteCell \*move = newBlockCell->move[j]; // Update the edges point to this blocked neighbour

if (move)

{

for (int q = 0; q < 8; q++)

{

if (move->move[q] == newBlockCell)

move->linkCost[q] = INF;

}

// Remove the blockCell from the succ set of neighbour

set<int>::iterator iter = move->succs.begin();

while (iter != move->succs.end())

{

int succIndex = \*iter;

DstarLiteCell \*succ = move->move[succIndex];

if (succ == newBlockCell)

{

move->succs.erase(iter);

break;

}

iter++;

}

updateVertex(move);

}

}

}

}

cout << "replanning start... " << endl;

updateHValues();

initStatistic();

resetCellsStatus();

long long t1 = milliseconds\_now();

if (!computeShortestPath())

{

cout << "error, no path found" << endl;

return false;

}

long long t2 = milliseconds\_now();

cout << "replanning for shortest path finished, time used: " << (t2 - t1) << " ms" << endl;

statCellsStatus(numberOfExpandedStates, numberOfVertexAccesses);

showStatistic();

updateUI();

}

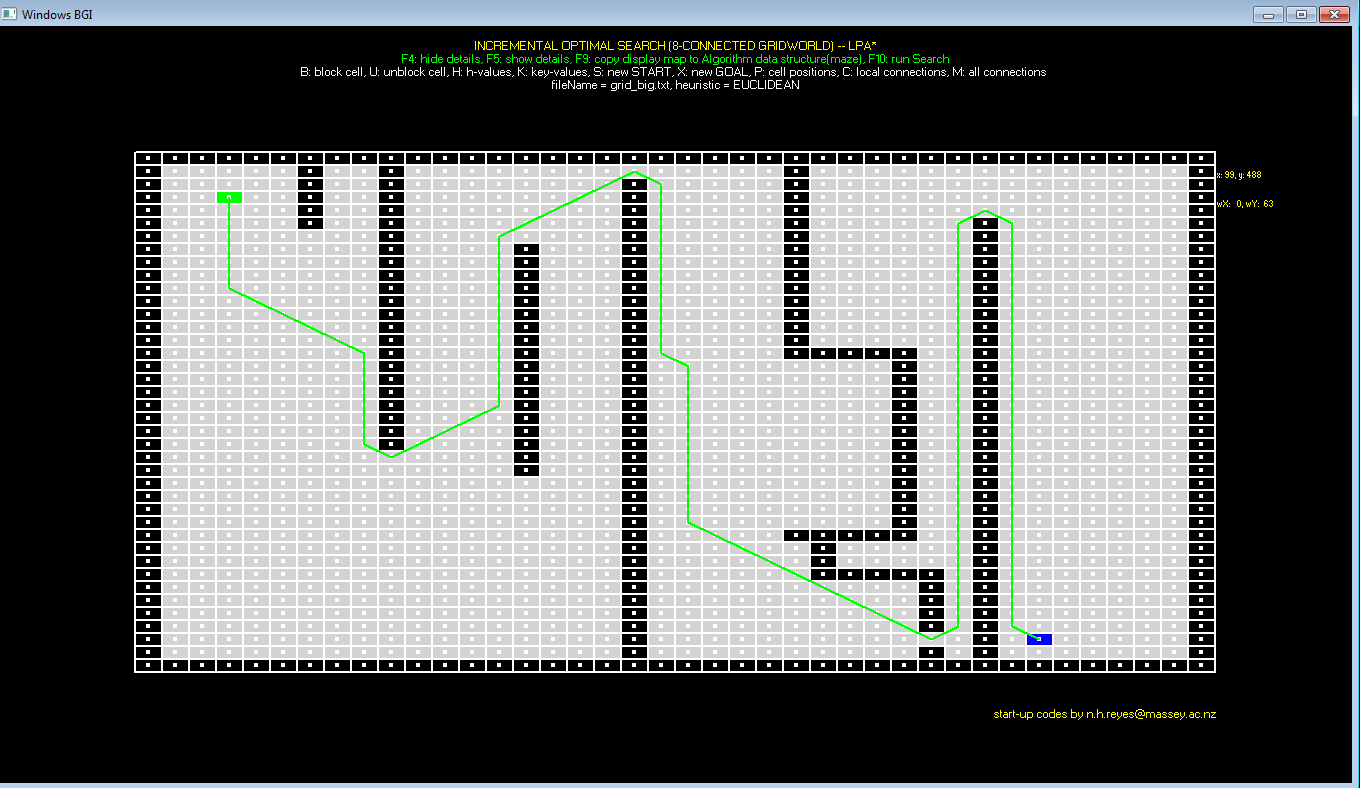
return true;

}

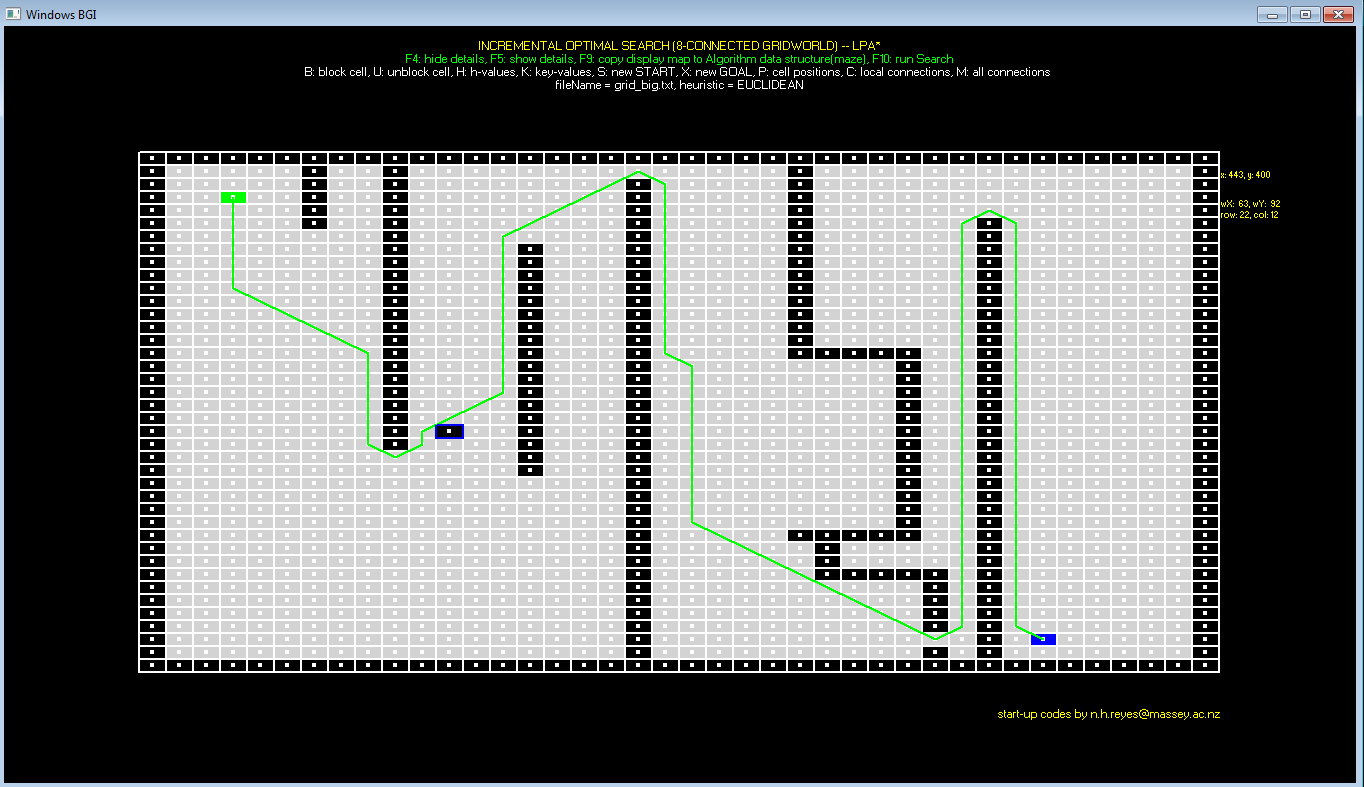
**#3 Snapshots**

Here are some screenshots for the grid file grid\_big.txt

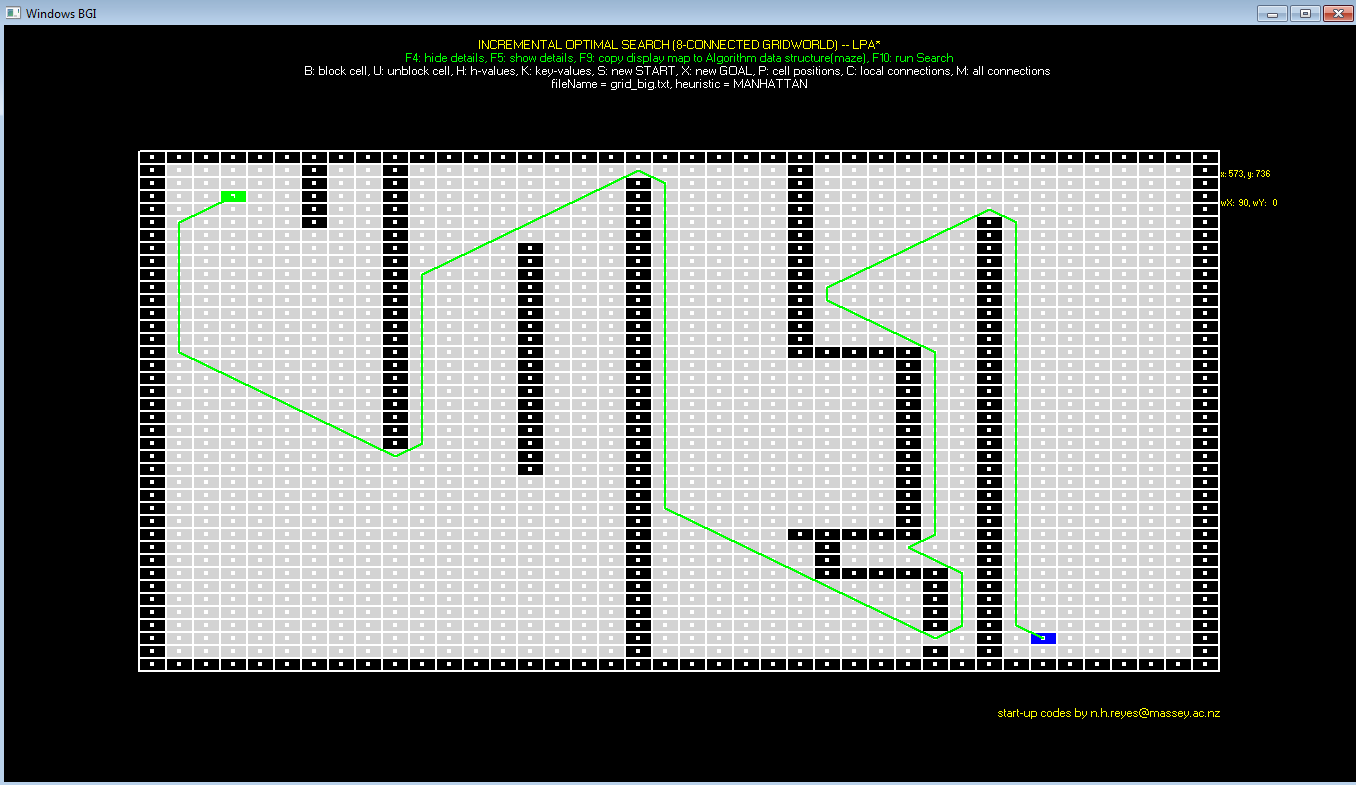
**LPA\* -- Euclidean -- 1st search**

****

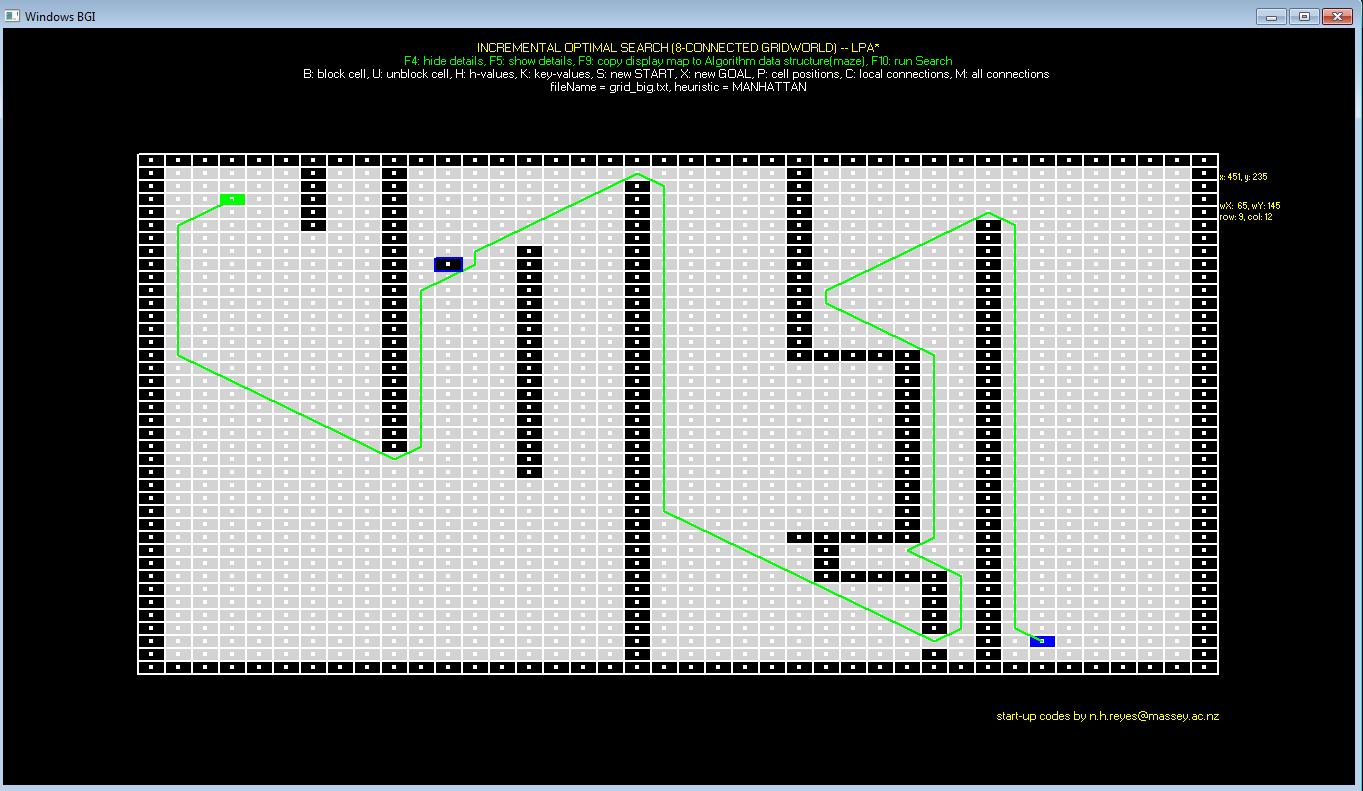
**LPA\* -- Euclidean -- Replanning**

****

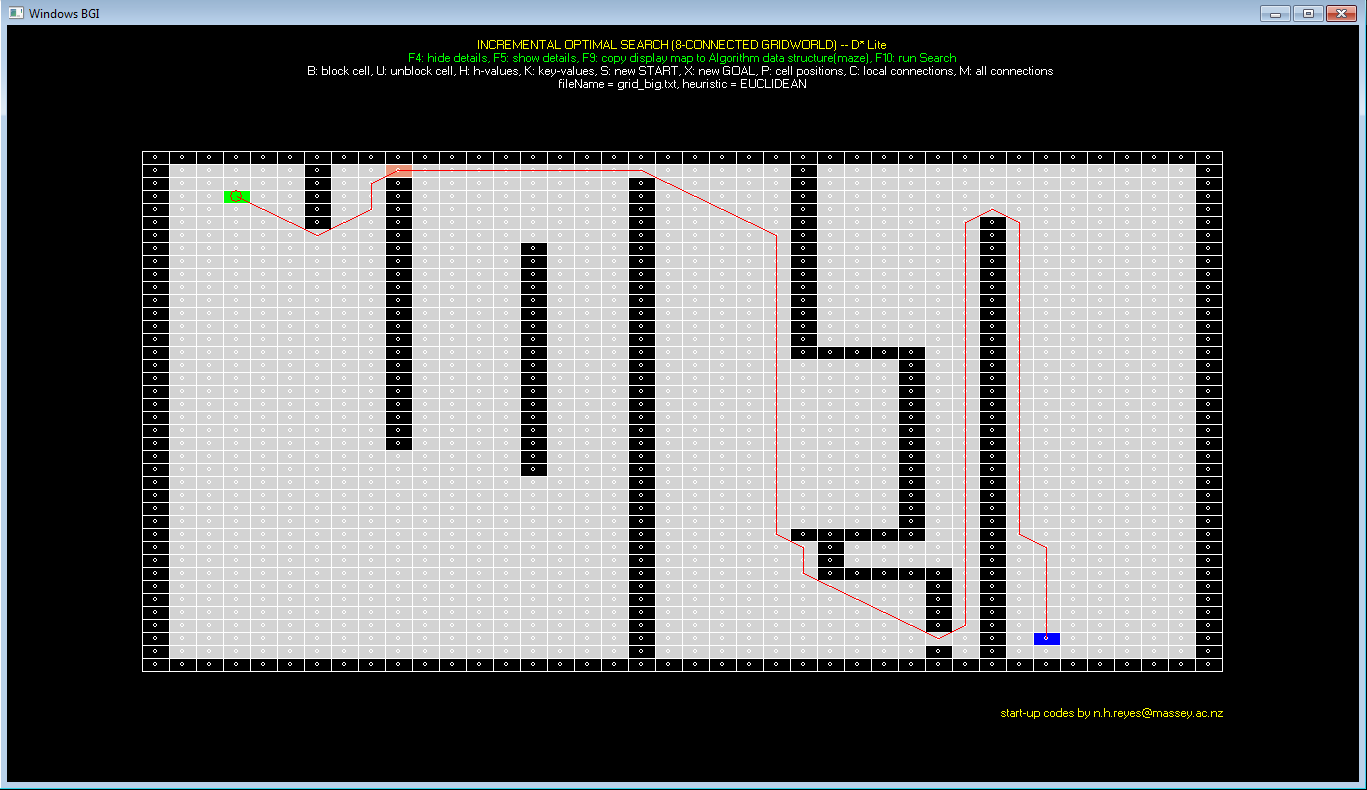
**LPA\* -- Manhattan-- 1st search**

****

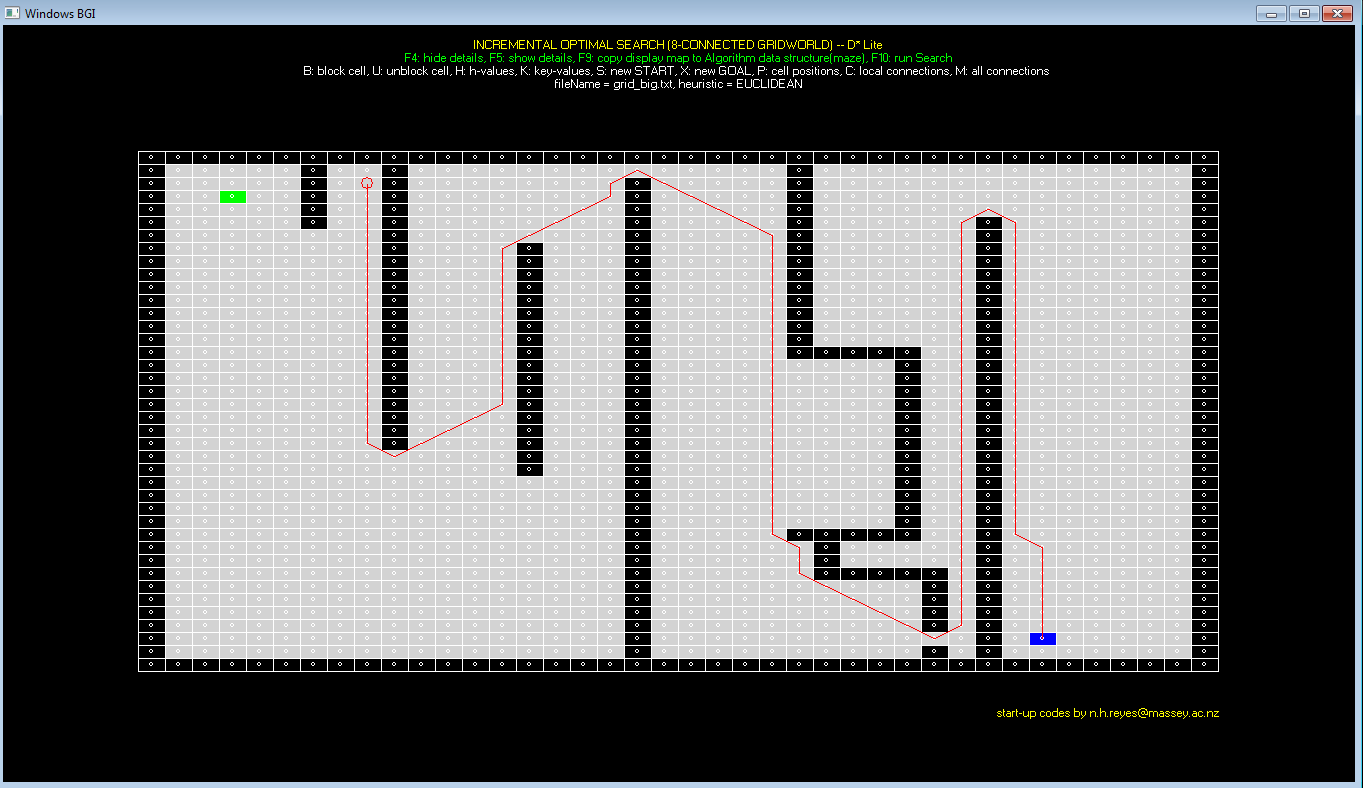
**LPA\* -- Manhattan--Replanning**



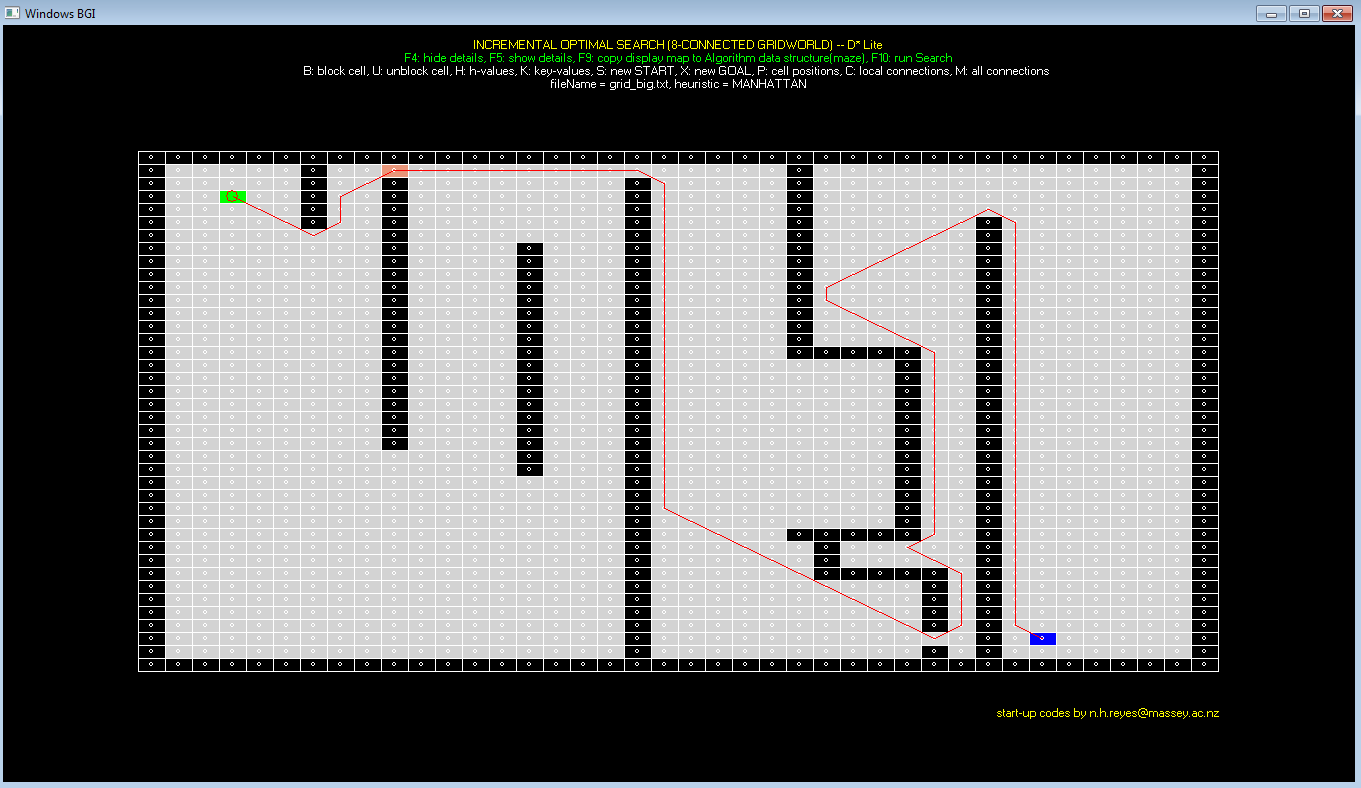
**D\* Lite -- Euclidean -- 1st search**



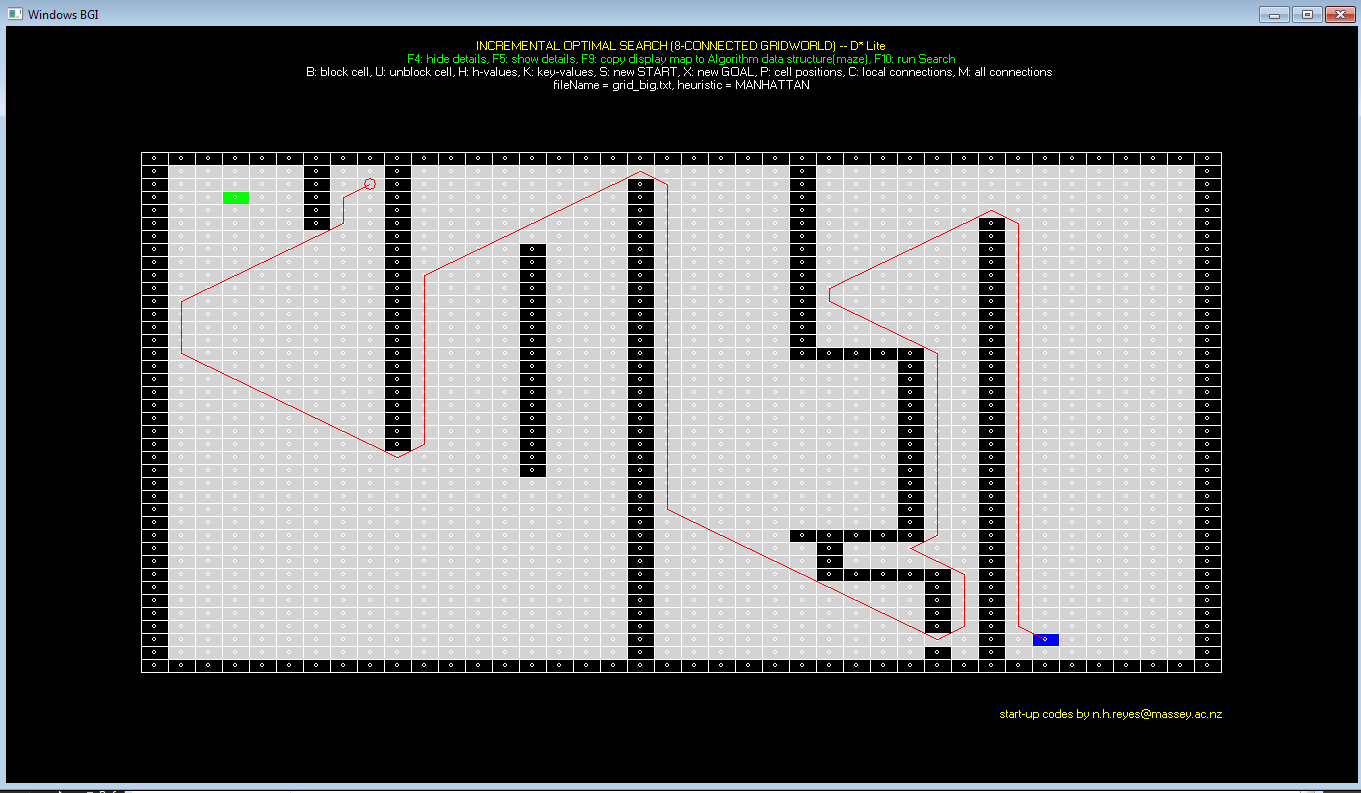
**D\* Lite -- Euclidean -- Replanning**

****

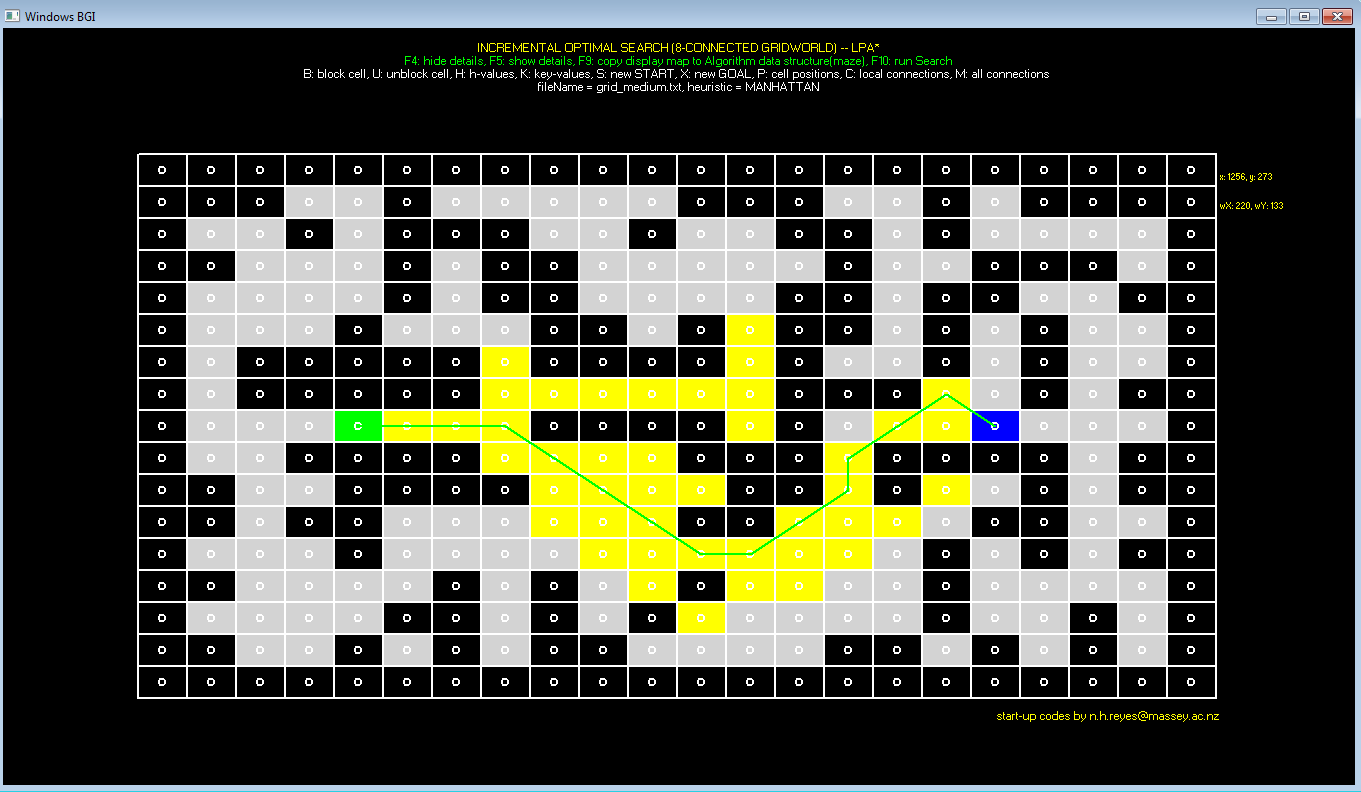
**D\* Lite -- Manhattan -- 1st search**



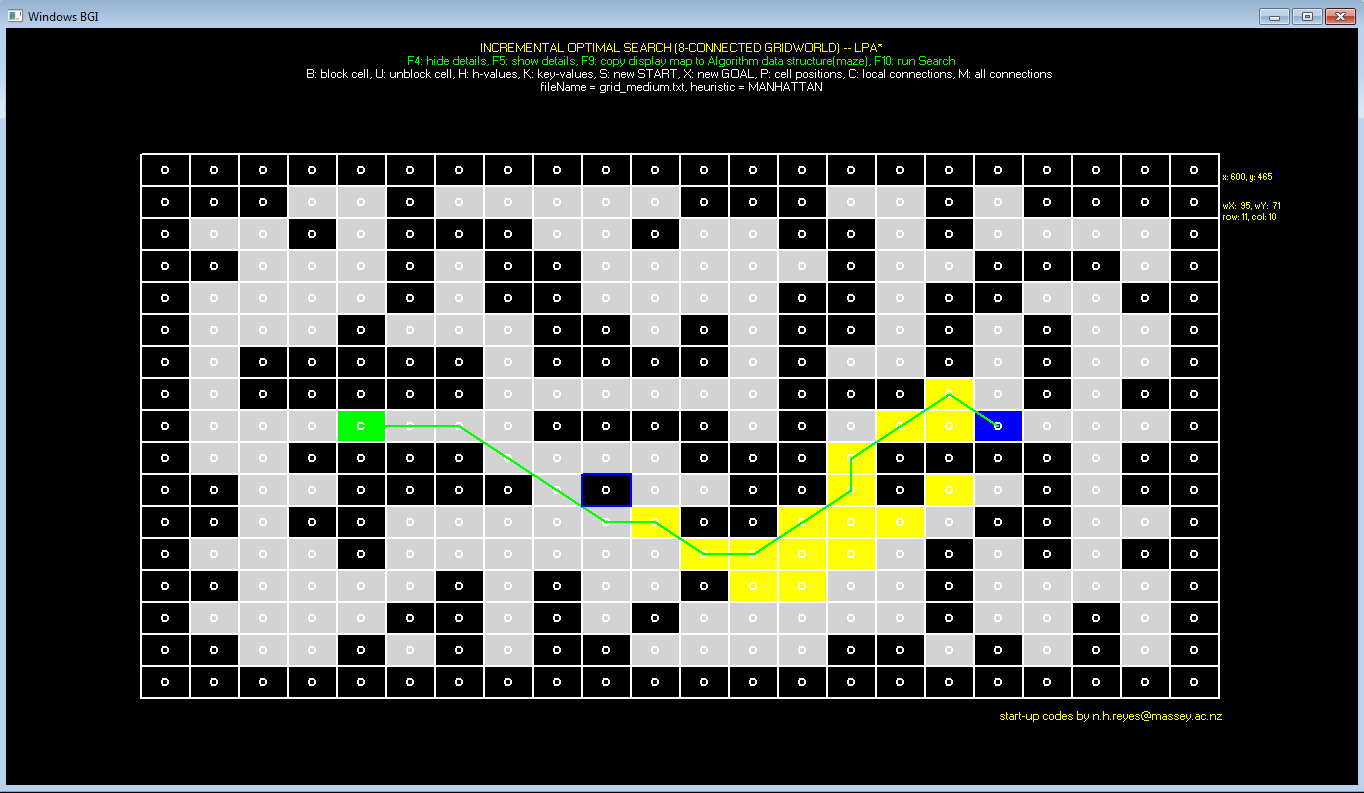
**D\* Lite -- Manhattan -- Replanning**

****

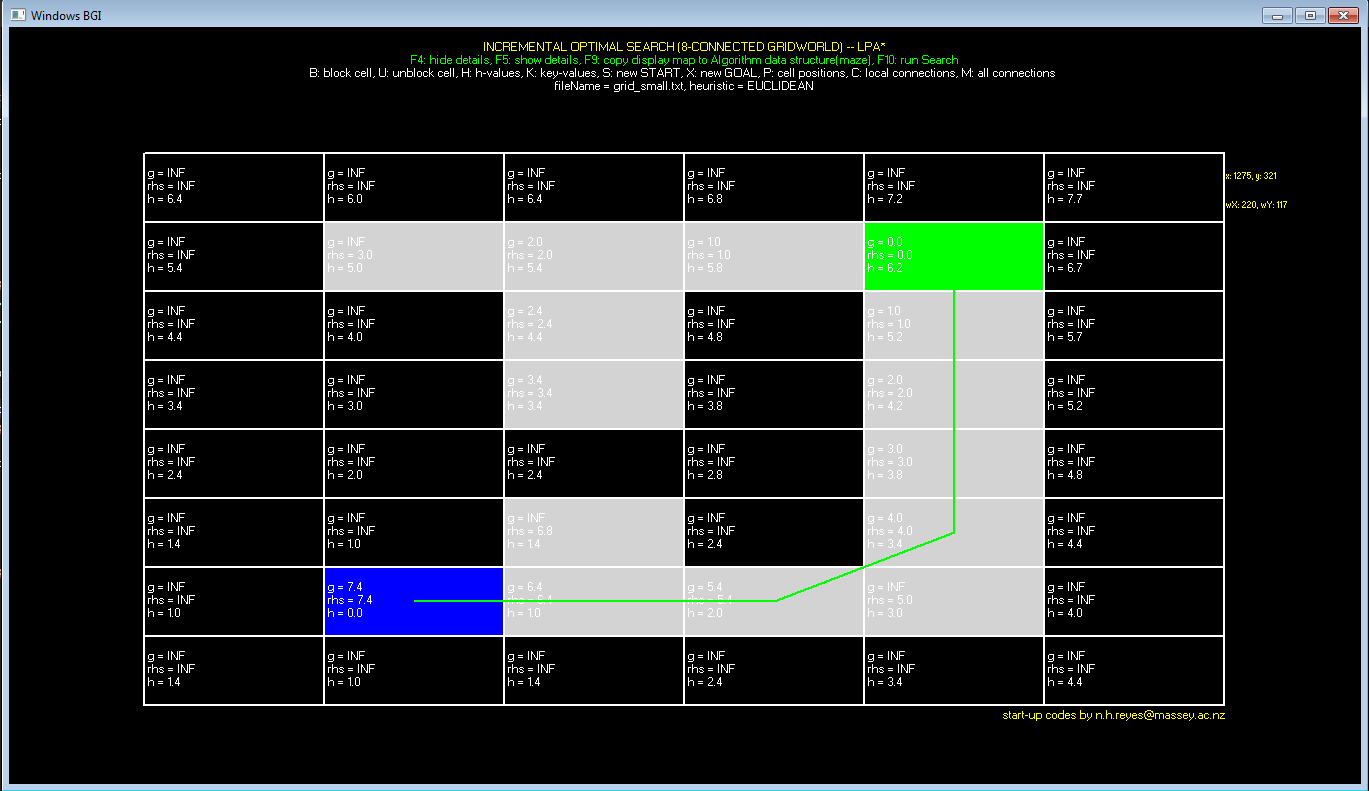
**LPA\* -- Manhattan -- 1st Search -- Highlight Expanded Cells**



**LPA\* -- Manhattan -- Replanning -- Highlight Expanded Cells**



**LPA\* -- Show Details**

****

**#4 Experiments Result**

**Heuristic = Euclidean distance**

**1) Grid file: grid\_small.txt (8x6)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 23 | 7.41 | 12 | 15 | 1 |
| Re-planning | 2 | 8.24 | 4 | 6 | 1 |
| D\* Lite | 1st Search | 27 | 6.82 | 8 | 12 | 1 |
| Re-planning | 3 | 10.24 | 10 | 14 | 0 |

**2) Grid file: grid\_medium.txt (17x22)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 23 | 16.8995 | 42 | 64 | 1 |
| Re-planning | 28 | 17.4853 | 16 | 47 | 1 |
| D\* Lite | 1st Search | 18 | 13.8284 | 17 | 28 | 0 |
| Re-planning | 22 | 19.7279 | 93 | 106 | 2 |

**3) Grid file: grid\_big.txt (40x40)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 61 | 156.426 | 1047 | 1098 | 25 |
| Re-planning | 54 | 156.426 | 5 | 13 | 1 |
| D\* Lite | 1st Search | 48 | 127.698 | 742 | 769 | 16 |
| Re-planning | 34 | 155.355 | 140 | 184 | 2 |

**Heuristic = Manhattan-8 distance**

**1) Grid file: grid\_small.txt (8x6)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 23 | 7 | 13 | 15 | 3 |
| Re-planning | 2 | 7 | 1 | 1 | 1 |
| D\* Lite | 1st Search | 25 | 6 | 8 | 12 | 0 |
| Re-planning | 3 | 9 | 10 | 14 | 0 |

**2) Grid file: grid\_medium.txt (17x22)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 26 | 14 | 45 | 71 | 1 |
| Re-planning | 27 | 14 | 17 | 37 | 1 |
| D\* Lite | 1st Search | 28 | 13 | 21 | 32 | 1 |
| Re-planning | 23 | 16 | 88 | 106 | 2 |

**3) Grid file: grid\_big.txt (40x40)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | | Max Queue Length | Path Length | NO. of state expansions | Vertex accessed | Actual Running Time(ms) |
| LPA\* | 1st Search | 53 | 144 | 1186 | 1220 | 25 |
| Re-planning | 53 | 144 | 591 | 648 | 12 |
| D\* Lite | 1st Search | 50 | 119 | 744 | 779 | 13 |
| Re-planning | 61 | 145 | 226 | 281 | 5 |

**#5 Discussion of the Result**

The experiments result shows that, for grid\_small(8x6) and grid\_medium(17x22), Both LPA\* and D\* Lite algorithms could compute the shortest path very quickly. In LPA\*, replanning always uses less compute resources than 1st search and so replanning is much more efficient than 1st search. When it comes to D\* Lite, replanning may consume less resources than 1st search but some times it could consume more, for example, when process grid\_medium.txt with D\* Lite, replanning expands 93 vertexes but 1st search only expands 17 vertexes. The reason is the robot moves to a location where is harder to reach the goal cell.

**#6 User’s Guide**

#1 Compile the application via make command after enter the folder.

#2 Once it is compiled successfully, use below command to run:

Main grid\_file [e/m] [l/d]

e represents Euclidean distance.

m represents Manhattan distance.

l represents LPA\* algorithm.

d represents D\* Lite algorithm.

For example:

Command “main grid\_big.txt e l” would load the grid file “grid\_big.txt”, heuristic function is specified as euclidean distance and the algorithm is specified as LPA \*.

#3 If the algorithm is LPA\*, press F9 to copy map to maze, press F10 to run first search, the path will be shown on map and statistic results will be printed on terminal.

After the path is shown, you can block one cell via typing “B”, then application will re-planning automatically.

#4 If the algorithm is D\* Lite, press F9 to copy map to maze, press F10 to run first search, the path will be shown on map and statistic results will be printed on terminal.

The robot will also be displayed by a red circle, you can press any key so that robot would move one step along the path. When the robot detect the next cell is blocked, it will re-planning automatically and then move along the new path, until it reaches the goal cell.

#5 Configure Display:

Turn on or turn off “Highlight expanded cell” by F6 or F7

Turn on or turn off “Show Details” by F5 or F4

Regarding to LPA\*, you can configure these two options at any time.

Regarding to D\* Lite, you can only configure them before running the first search(F10).