using System;

using System.Collections;

using System.Collections.Generic;

using Rhino;

using Rhino.Geometry;

using Grasshopper;

using Grasshopper.Kernel;

using Grasshopper.Kernel.Data;

using Grasshopper.Kernel.Types;

using System.Linq;

/// <summary>

/// This class will be instantiated on demand by the Script component.

/// </summary>

public class Script\_Instance : GH\_ScriptInstance

{

#region Utility functions

/// <summary>Print a String to the [Out] Parameter of the Script component.</summary>

/// <param name="text">String to print.</param>

private void Print(string text) { /\* Implementation hidden. \*/ }

/// <summary>Print a formatted String to the [Out] Parameter of the Script component.</summary>

/// <param name="format">String format.</param>

/// <param name="args">Formatting parameters.</param>

private void Print(string format, params object[] args) { /\* Implementation hidden. \*/ }

/// <summary>Print useful information about an object instance to the [Out] Parameter of the Script component. </summary>

/// <param name="obj">Object instance to parse.</param>

private void Reflect(object obj) { /\* Implementation hidden. \*/ }

/// <summary>Print the signatures of all the overloads of a specific method to the [Out] Parameter of the Script component. </summary>

/// <param name="obj">Object instance to parse.</param>

private void Reflect(object obj, string method\_name) { /\* Implementation hidden. \*/ }

#endregion

#region Members

/// <summary>Gets the current Rhino document.</summary>

private readonly RhinoDoc RhinoDocument;

/// <summary>Gets the Grasshopper document that owns this script.</summary>

private readonly GH\_Document GrasshopperDocument;

/// <summary>Gets the Grasshopper script component that owns this script.</summary>

private readonly IGH\_Component Component;

/// <summary>

/// Gets the current iteration count. The first call to RunScript() is associated with Iteration==0.

/// Any subsequent call within the same solution will increment the Iteration count.

/// </summary>

private readonly int Iteration;

#endregion

/// <summary>

/// This procedure contains the user code. Input parameters are provided as regular arguments,

/// Output parameters as ref arguments. You don't have to assign output parameters,

/// they will have a default value.

/// </summary>

private void RunScript(List<Rectangle3d> Recs, int GridNum, ref object A)

{

// create all grids

List<Grid> grids = new List<Grid>();

// forrach: enumerates elements of a list and executes for each element

// foreach: (int a in b) variable-int a, collection-b

foreach (Rectangle3d rec in Recs)

{

grids.Add(new Grid(rec));

}

// create the gridmap (int[i,j] length:i, dimension:j)

GridMap gridMap = new GridMap(grids, new int[2] {0, 0}, new int[2] {GridNum - 1, GridNum - 1});

// find the path

List<Grid> path = gridMap.findPath();

// visualize the path with rhino points

List<Point3d> pathpts = GridMap.vizPath(path);

}

// <Custom additional code>

// grid object

public class Grid

{

//initialize

//index in gridmap - location:(x,y)

public int x;

public int y;

//costs A\*algorithm f=g+h

//g:from initial to current h:heuristic value,from current to final

public int gcost;

public int hcost;

public int fcost;

//coords to draw path, center point of grid (x,y)

public double cx;

public double cy;

//obstacle, the one block path, Boolean algebra

public bool isObstacle;

//parent, parent code - children code - Binary tree

public Grid parent;

//constructor 1:valid grid for walking from recs

public Grid(Rectangle3d rec)

{

Point3d center = rec.Center;

this.cx = center.X;

this.cy = center.Y;

this.gcost = int.MaxValue;

this.hcost = int.MaxValue;

this.fcost = int.MaxValue;

this.isObstacle = false;

this.parent = null;

}

//constuctor2: obstacle grid from xi,yi,cx,cy

public Grid(int xi, int yi, double cx, double cy)

{

this.cx = cx;

this.cy = cy;

this.x = xi;

this.y = yi;

this.gcost = int.MaxValue;

this.hcost = int.MaxValue;

this.fcost = int.MaxValue;

this.isObstacle = true;

this.parent = null;

}

}

// the gridmap object

public class GridMap

{

//initialize

//put attribute "," on class or methods, two dimentional array in (x,y)

public Grid[,] grids;

//entry and exit

public Grid grid\_start;

public Grid grid\_end;

//coords

//how many x,y coordinates in grid

public List<double> xvals;

public List<double> yvals;

public int xnum;

public int ynum;

//A star sets

//the grids have been and not been retrieved

public List<Grid> open\_grids;

public List<Grid> close\_grids;

//the 4 directions for walking in next step to process

public int[,] dirs = new int[4, 2] {{0,1},{1,0},{0,-1},{-1,0}};

//constructor's method, all of the grids, pick up entry and exit

public GridMap(List<Grid> grids, int[] start\_id, int[] end\_id)

{

//invoke the methods in private: find all unique xvals and yvals

this.formatGrids(grids);

//put grids to their location by coordinates

this.indexGrids(grids);

//set start and end grid according the array produced by constructors

this.grid\_start = this.grids[start\_id[1], start\_id[0]];

this.grid\_end = this.grids[end\_id[1], end\_id[0]];

//corresponding with three methods as follows g,f,hcost function

this.grid\_start.gcost = 0;

this.grid\_start.hcost = this.hcost(this.grid\_start);

this.grid\_start.fcost = this.fcost(this.grid\_start);

//two lists (list wait for operation, and operation finished,add entry to open list)

this.open\_grids = new List<Grid>();

this.close\_grids = new List<Grid>();

this.open\_grids.Add(this.grid\_start);

}

//find set and length of all x and yvals

private void formatGrids(List<Grid> grids)

{

//find all unique xvals and yvals

List<double> xvals = new List<double>();

List<double> yvals = new List<double>();

foreach(Grid grid in grids)

//add value to list

{

xvals.Add(grid.cx);

yvals.Add(grid.cy);

}

//delete repeat data and sort them

xvals = xvals.Distinct().ToList();

yvals = yvals.Distinct().ToList();

xvals.Sort();

yvals.Sort();

// length of value

this.xvals = xvals;

this.yvals = yvals;

this.xnum = xvals.Count;

this.ynum = yvals.Count;

}

//put grids to locations by coords, set their x and y

private void indexGrids(List<Grid> grids)

{

this.grids = new Grid[this.ynum, this.xnum];

//set xi,yi for all grids, find grids index and put them in their location

foreach (Grid grid in grids)

{

grid.x = this.xvals.IndexOf(grid.cx);

grid.y = this.yvals.IndexOf(grid.cy);

this.grids[grid.y, grid.x] = grid;

}

//set the obstacle

for (int icol = 0; icol < this.ynum; icol++)

{

for (int irow = 0; irow < this.xnum; irow++)

{

if (this.grids[icol, irow] == null)

{

Grid grid = new Grid(irow, icol, this.xvals[irow], this.yvals[icol]);

this.grids[icol, irow] = grid;

}

}

}

}

//gcost function

private int gcost(Grid grid)

{

int gc = 0;

while (grid.parent != null)

{

gc++;

grid = grid.parent;

}

return gc;

}

//hcost manhattan distance (straight line distance:|x1-x2|+|y1-y2|)

private int hcost(Grid grid)

{

return Math.Abs(grid.x - this.grid\_end.x) + Math.Abs(grid.y - this.grid\_end.y);

}

//fcost

private int fcost(Grid grid)

{

return grid.gcost + grid.hcost;

}

//key function, find the path

public List<Grid> findPath()

{

//ensure valid start and end point

if (this.grid\_start.isObstacle || this.grid\_end.isObstacle)

{

throw new ArgumentException("Either the start or end grid is obstacle, no path is found");

}

//execute

while (this.open\_grids.Count > 0)

{

//select the grid with lowest cost from the open set

Grid grid = this.selLowestCostGrid();

if (this.isEndGrid(grid))

{

//build path and return

return this.buildPath(grid);

}

//not the end grid, check then put in close list for avoid repeating

this.open\_grids.Remove(grid);

this.close\_grids.Add(grid);

//process the grid in 4 directions

for (int diri = 0; diri < 4; diri++)

{

//parse neighbor grid in the directions (deal with the 4 grids around surrounding the center grid)

this.parseNeighbor(grid, grid.x + this.dirs[diri, 0], grid.y + this.dirs[diri, 1]);

}

}

// path not found

throw new ArgumentException("No path is found from the start grid to the end, delete some obstacles and try again");

}

//select grid with lowest cost from open set

private Grid selLowestCostGrid()

{

List<Grid> tmpgrids = this.open\_grids.OrderBy(g => g.fcost).ToList();

return tmpgrids[0];

}

//check if grid is end

private bool isEndGrid(Grid grid)

{

return grid == this.grid\_end;

}

//check if grid is start

private bool isStartGrid(Grid grid)

{

return grid == this.grid\_start;

}

//build the path by tracing the parent field from end to start

private List<Grid> buildPath(Grid grid)

{

List<Grid> path = new List<Grid>();

while (!this.isStartGrid(grid))

{

path.Insert(0, grid);

grid = grid.parent;

}

//do not forget start grid

path.Insert(0, grid);

return path;

}

//parse the neighbor grid

private void parseNeighbor(Grid parent, int x, int y)

{

//if out of boundary

if (x < 0 || y < 0 || x >= this.xnum || y >= this.ynum)

{

return;

}

Grid grid = this.grids[y, x];

if (grid.isObstacle)

{

return;

}

// if grids already parsed

if (this.close\_grids.Contains(grid))

{

return;

}

/\*

if the grid is in the open set,has been parsed before,

then calculate a new gcost from the parent grid,

compare the new gcost with original one, and update if need

\*/

if (this.open\_grids.Contains(grid))

{

int tmpgcost = parent.gcost + 1;

if (tmpgcost < grid.gcost)

{

// a shorter path found, update

grid.parent = parent;

grid.gcost = tmpgcost;

grid.hcost = this.hcost(grid);

grid.fcost = this.fcost(grid);

return;

}

else

{

return;

}

}

// if the grid is brand new for both the open and clost sets

grid.parent = parent;

grid.gcost = parent.gcost + 1;

grid.hcost = this.hcost(grid);

grid.fcost = this.fcost(grid);

this.open\_grids.Add(grid);

return;

}

// visualize the path

public static List<Point3d> vizPath(List<Grid> path)

{

List<Point3d> pathpts = new List<Point3d>();

foreach (Grid grid in path)

{

pathpts.Add(new Point3d(grid.cx, grid.cy, 0));

}

return pathpts;

}

}

// </Custom additional code>

}