Uniform-cost Search

一致代价搜索

1.原理

BFS（Breadth-first Search，宽度优先搜索）只在每步代价一样时才是最优的，如按照结点深度来搜索，每一层其代价都是1.但是如果每步代价不一样，BFS就不是最优了，这时就要构造一个行动代价函数，这就是一致代价搜索（Uniform-cost Search,后称UCS）。UCS与BFS有两点不一样：1）目标结点检测应用于被选择扩展时，原因是第一个生成的目标结点也许是次优的。2）如果在扩展时发现有更好的路径到达目标，那将替换掉它。其实个人觉得还有第三个不同，那就是UCS使用优先级队列，而BFS使用一般队列即可。

3.数据结构

使用邻接矩阵来构造该图。

4.实验过程：本次实验参考页面如下：<https://blog.csdn.net/obestboy/article/details/86604030>

实验中遇到的问题：本次实验使用java语言实现，实验中一些方法都是本学期现学现用的，虽然磕磕绊绊错误很多，但学以致用还是很有成就感的，通过查阅网上资料一个个更正错误，同时更深入地了解了ucs算法的核心思想

5.java源代码：

1）图结构

public class Graph {

final static int MAXSIZE=1000;//最多1000个顶点

String[] vertexes=new String[MAXSIZE]; //顶点数组

int[][] edges=new int[MAXSIZE][MAXSIZE];//邻接矩阵

int verNum,edgeNum;//顶点个数、边条数

public Graph()

{

verNum=0;

edgeNum=0;

}

public Graph(String[] Vertexes,int[][] Edges,int VerNum,int EdgeNum)

{

vertexes=Vertexes;

edges=Edges;

verNum=VerNum;

edgeNum=EdgeNum;

}

}

2）路径结构

public class costPath {

String path;//路径，如3216

int cost;//路径代价

int estCost;//估计路径代价，启发函数或评估函数计算出的代价

public costPath(String Path)

{

path=Path;

cost=0;

estCost=0;

}

public costPath(String Path,int Cost)

{

path=Path;

cost=Cost;

estCost=0;

}

public costPath(String Path,int Cost,int EstCost)

{

path=Path;

cost=Cost;

estCost=EstCost;

}

public void print()

{

System.out.print(path+",cost:"+cost+",estimate Cost:"+estCost+"||");

}

}

4.输出图

public void print()

{

//输出顶点

System.out.print("顶点：");

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

System.out.print(vertexes[i]+" ");

//输出每条边

System.out.println();

System.out.println("边：");

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

{

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

if(edges[i][j]>0) //表示有边，为0就没有了，不考虑负值边

System.out.print(vertexes[i]+"=>"+vertexes[j]+":"+edges[i][j]+" ");

System.out.println();

}

}

5.查找顶点对应的编号

public int searchVertex(String vertex)

{

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

if(vertexes[i]==vertex) return i;

return -1;

}

6.解析路径并输出对应代价

public void parsePath(ArrayList<costPath> path)

{

for(int i=0;i<path.size();i++)

{

costPath subPath=path.get(i);

String[] realPath=subPath.path.split("-");

for(int j=0;j<subPath.path.length();j++)

{

if(j==0) //第一条路径不加=>符号

System.out.print(vertexes[Integer.parseInt(realPath[j])]);

else

System.out.print("=>"+vertexes[Integer.parseInt(realPath[j])];

}

System.out.println(",Cost:"+subPath.cost);//输出代价

}

}

7.判断结点是否在当前路径字符数组

public boolean isInStringArray(String[] s,int v)

{

for(int i=0;i<s.length;i++)

if(Integer.parseInt(s[i])==v) return true;

return false;

}

8.一致代价搜索UCS

思路：用allPath存储找到的路径（如果需要返回多个路径的话），用explored存储已探索结点，用priFrontier存储路径队列（注意UCS需要使用优先级队列）。由于使用了路径类（路径、代价），在路径类优先级队列的比较中需要重写比较器。先测试是否是目标结点，如不是再扩展该结点。

public ArrayList<costPath> ucs(int src,int dst) {

if (src < 0 && dst < 0 && src >= verNum && dst >= verNum) return null;

//allPath存储路径及代价

ArrayList<costPath> allPath = new ArrayList<>();

if (src == dst) {

allPath.add(new costPath(String.valueOf(src), 0));

return allPath;

}

//重写优先级队列的比较函数

PriorityQueue<costPath> priFrontier = new PriorityQueue<>(

new Comparator<costPath>() {

public int compare(costPath cp1, costPath cp2) {

return cp1.cost - cp2.cost;

}

});

//explored存储已探索过的结点

ArrayList<Integer> explored = new ArrayList<>();

//源结点入队，设置代价为0

priFrontier.offer(new costPath(String.valueOf(src), 0));

while (!priFrontier.isEmpty()) {

//代价最小的路径出队（优先级队列嘛）

costPath cp = priFrontier.remove();

//p为路径

String p = cp.path;

//获取路径最后一个结点

String[] pathSplit=p.split("-");//拆分路径

int v = Integer.parseInt(pathSplit[pathSplit.length-1]);

if (v == dst) { //目标测试，发现目标

allPath.add(cp);//将路径及代价添加到allPath

//return allPath;//如果只要最优路径直接返回退出即可

}

else {

//将该结点设置为已探索

explored.add(v);

//下面for循环找到结点v的后继结点并做相应处理：入队、替换

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

if (edges[v][i] > 0) { //有边

//组合当前路径及后继结点，备用

String p1 = p+"-"+i;

//后继结点是否在当前路径上，如果是，则会形成环，将丢弃

boolean isInCurrentPath = isInStringArray(pathSplit,i);

//判断该后继结点是否未探索、或者是否不在当前路径上

//未探索过的肯定需要入队

//另外，虽然已探索过，但并不在当前路径上的结点也需入队，

//这样有更多备选的路径，不至于错过短的路径

if (explored.indexOf(i) == -1 || !isInCurrentPath)

//将其入队

priFrontier.offer(new costPath(p1, cp.cost + edges[v][i]));

}

}

}

}

}

return allPath;

}

9.测试及结果

public class Main {

public static void main(String[] args)

{

//顶点表

String[] vers = {"c1", "c2", "c3", "c4", "c5", "c6", "c7", "c8", "c9"};

int[][] edgs={ //邻接矩阵

{0,5,0,0,0,0,0,2,0},

{5,0,2,0,0,1,0,0,0},

{0,2,0,4,5,0,0,0,2},

{0,0,4,0,2,0,0,0,3},

{0,0,5,2,0,6,0,0,0},

{0,1,0,0,6,0,1,2,0},

{0,0,0,0,0,1,0,3,0},

{2,0,0,0,0,2,3,0,0},

{0,0,2,3,0,0,0,0,0}

};

Graph g=new Graph(vers,edgs,9,13);

System.out.println("Uniform-cost Search:");

g.parsePath1(g.ucs(g.searchVertex("c4"),g.searchVertex("c6")));

}

}

运行结果如下：（如果ucs方法里找到目标后直接return allPath，将只会返回第一条路径，也就是最短的或最优的路径）

Uniform-cost Search:

c4=>c3=>c2=>c6,Cost:7

c4=>c5=>c6,Cost:8

c4=>c9=>c3=>c2=>c6,Cost:8

c4=>c3=>c2=>c1=>c8=>c6,Cost:15

c4=>c9=>c3=>c2=>c1=>c8=>c6,Cost:16

c4=>c3=>c2=>c1=>c8=>c7=>c6,Cost:17

c4=>c9=>c3=>c2=>c1=>c8=>c7=>c6,Cost:18

从结果可以看出，路径代价依次增加，如果代价一样，则路径深度浅的排在前面。因此，ucs是最优的

10.分析

UCS以路径代价而不是深度来扩展结点，路径代价最小的优先扩展，BFS只是UCS的一个特例，UCS是最优的，只要不存在0代价结点（我的理解，如自环），每步代价都大于一个很小的数e，那么UCS也是完备的。

假设最优路径的代价为C\*，设每步代价不少于e，则最坏情况的时空复杂度为，其中||表示向下取整，显然比BFS要大很多，因为要探索更多小的路径。