|  |
| --- |
|  |
| 算法实验报告 |
|  |

|  |
| --- |
| 王殊 计算机1604 1611640413 |

# 活动安排：

活动安排问题：

设计思路：

首先要对输入的活动按其结束时间进行非减序排列，利用贪心算法，一开始选择活动1.并初始化为1。然后依次检查活动i是否与当前的活动相容。若相容则将活动i加入到已选择活动的集合A中，否则不选择活动i，而继续检查下一活动与集合A的相容性。

活动i与当前集合A中所有活动相容的充要条件是：Si>=fi.

#include <iostream>

#include <algorithm>

#include <iomanip>

using namespace std;

typedef struct {

int start;

int end;

} activity;

void GreedSlector(int n, activity \*sf)

{

bool A[11] = {};

A[0] = true;

int j = 0;

for (int i = 1; i <= n; i++) {

if(sf[i].start > sf[j].end){

A[i] = true;

j = i;

}else{

A[i] = false;

}

}

cout << "安排的活动如下\n";

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

{

if(A[i]){

cout << "Act "<< setw(2) << i + 1 << ", Start: " << setw(2) << sf[i].start << ", End:" << setw(2)<< sf[i].end << "\n";

}

}

}

activity activities[11] = {

{1,4},

{3,5},

{0,6},

{5,7},

{3,8},

{5,9},

{6,10},

{12,14},

{2,13},

{8,12},

{8,11}

};

int main()

{

sort(activities, activities + 11, [](activity &lhs, activity &rhs){return lhs.end < rhs.end;});

cout << "11个活动的开始时间和结束时间按结束时间按非减序列排列如下：\n";

for(int i=0;i<11;++i)

{

cout << "Act "<< setw(2) << i + 1 << ", Start: " << setw(2) << activities[i].start << ", End:" << setw(2)<< activities[i].end << "\n";

}

GreedSlector(11, activities);

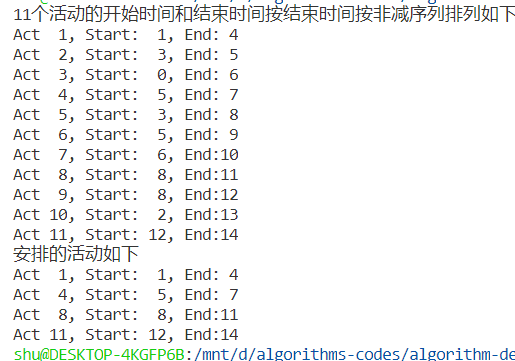
return 0;

}

效率分析：

对活动按照非减序进行排序，采用快速排序算法，由于快速排序基准两侧的大小关系已明确，所以减少了合并的操作。快速排序的运行时间与划分对称有关，其最坏的情况下在划分过程中产生两个分别包含n-1个元素和1个元素的时候，基于此种情况，划分的每一步都不对称，其复杂度为O（n2），在最好的情况下，每次划分的基准都恰好为中值，此时的算法复杂度为O（logn）。对于活动安排问题，贪心算法却总能求得总体的最优解，即它最终确定的相容活动集合A规模最大。

实验结果：



# 单源最短路径

描述

单源最短路径，给定一个带权图，其中每条边的权是非负的实数，计算从源到所有的其他定点的最短路径长度。

设计思路

设置顶点集合S并不断的做贪心选择来扩充这个集合，一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。初始时，S中仅含有源。用数组dist记录当前每个顶点所对应的最短特殊路径长度。

Dijkstra 有两个优化的方式：

1. 使用最小堆： 在选择从v点到可达的权重最小的顶点的时候，加入到索引最小堆。保存的是具有最权重 dist[v][w] 的 w 顶点的在[0, V) 的索引。然后执行放松操作，减小权值
2. 使用拓扑排序：先对图进行拓扑排序，只要是在标记的节点w就不会作为放松操作的操作者重新被访问（但是他会被其他的顶点v作为放松的对象被访问）。

算法实现(C#) – 使用最小堆

/// <summary>

/// Dijkstra 算法求有向图单源最短路径

/// </summary>

class DijkstraSP:IShortestPath<DirectedEdge>

{

private DirectedEdge[] edgeTo;//edgeTo[i] 保存的是 到 i 的边

private Double[] disTo;// disTo[i] 保存的是从 v 到 i 的距离

private Chapter2.SortDemos.IndexPriorityQueue<Double> pq;//始终有最短的边

/// <summary>

/// 构造函数

/// </summary>

/// <param name="graph">有向加权非负权图</param>

/// <param name="s">起点</param>

public DijkstraSP(DirectedWeightedGraph graph, int s)

{

edgeTo = new DirectedEdge[graph.V];

disTo = new Double[graph.V];

pq = new Chapter2.SortDemos.IndexPriorityQueue<double>(graph.V);

for(Int32 v=0;v<graph.V;++v)

{

disTo[v] = Double.PositiveInfinity;

}

disTo[s] = 0.0;//自己到自己为0

pq.Insert(s, 0.0);

while (!pq.IsEmpty())

Relax(graph, pq.DeleteMin());//放松节点，找到到某个点的最短的距离

}

public Double DistTo(Int32 v)

{

return this.disTo[v];

}

public Boolean HasPathTo(Int32 v)

{

return this.disTo[v] != Double.PositiveInfinity;

}

public IEnumerable<DirectedEdge> PathTo(Int32 v)

{

if (!HasPathTo(v))

return null;

Stack<DirectedEdge> edges = new Stack<DirectedEdge>();

//向前退。自己到自己是没有路径的

for (DirectedEdge edge = edgeTo[v]; edge != null; edge = edgeTo[edge.Src])

{

edges.Push(edge);

}

return edges;

}

private void Relax(DirectedWeightedGraph g, int v)

{

foreach(DirectedEdge edge in g.GetEdge(v))

{

int w = edge.End;

//如果存在放松的条件

if (disTo[w] > disTo[v] + edge.Weight)

{

//存在，需要更新

disTo[w] = disTo[v] + edge.Weight;

edgeTo[w] = edge;

//重新累计从 s 到某个点的最短距离。这个距离只有可能减小

if (pq.Contains(w))

pq.Change(w, disTo[w]);

else

pq.Insert(w, disTo[w]);

}

}

}

public static void Main()

{

System.IO.TextReader s = System.IO.File.OpenText("tinyEWDAG.txt");

DirectedWeightedGraph graph = new DirectedWeightedGraph(s);

Console.Write("请输入起点：");

int st = int.Parse(Console.ReadLine());

DijkstraSP dijkstraSP = new DijkstraSP(graph, st);

for(int i=0;i<graph.V;++i)

{

if (i == st)

{

Console.WriteLine("{0}->{0} : 0.0", st);

continue;

}

if(!dijkstraSP.HasPathTo(i))

{

Console.WriteLine("{0}->{1} : No paths", st, i);

continue;

}

Double len = 0.0;

foreach(var edge in dijkstraSP.PathTo(i))

{

Console.Write(edge);

Console.Write(",");

len += edge.Weight;

}

Console.WriteLine(" : {0}", len);

}

}

}

算法实现(C#)-使用拓扑排序

/// <summary>

/// 效率高的处理。使用了拓扑排序。因为不可能再遇到需要放松的节点了。按照一定的顺序放松

/// </summary>

class AcyclicSP : IShortestPath<DirectedEdge>

{

private DirectedEdge[] edgeTo;//edgeTo[i] 保存的是 到 i 的边

private Double[] distTo;// disTo[i] 保存的是从 v 到 i 的距离

/// <summary>

/// 构造函数

/// </summary>

/// <param name="g">有向加权非负权图</param>

/// <param name="s">起点</param>

public AcyclicSP(DirectedWeightedGraph g, int s)

{

edgeTo = new DirectedEdge[g.V];

distTo = new Double[g.V];

for (Int32 v = 0; v < g.V; ++v)

{

distTo[v] = Double.PositiveInfinity;

}

distTo[s] = 0.0;

Topological top = new Topological(g);

foreach(int v in top.Order())

{

this.Relax(g, v);

}

}

public Double DistTo(Int32 v)

{

return this.distTo[v];

}

public Boolean HasPathTo(Int32 v)

{

return this.distTo[v] != Double.PositiveInfinity;

}

public IEnumerable<DirectedEdge> PathTo(Int32 v)

{

if (!HasPathTo(v))

return null;

Stack<DirectedEdge> edges = new Stack<DirectedEdge>();

//向前退。自己到自己是没有路径的

for (DirectedEdge edge = edgeTo[v]; edge != null; edge = edgeTo[edge.Src])

{

edges.Push(edge);

}

return edges;

}

private void Relax(DirectedWeightedGraph g, int v)

{

foreach (DirectedEdge edge in g.GetEdge(v))

{

int w = edge.End;

//如果存在放松的条件

if (distTo[w] > distTo[v] + edge.Weight)

{

distTo[w] = distTo[v] + edge.Weight;

edgeTo[w] = edge;

}

}

}

public static void Main()

{

System.IO.TextReader s = System.IO.File.OpenText("mediumEWD.txt");

//System.IO.TextReader s = System.IO.File.OpenText("tinyEWDAG.txt");

DirectedWeightedGraph graph = new DirectedWeightedGraph(s);

Console.Write("请输入起点：");

int st = int.Parse(Console.ReadLine());

AcyclicSP sp = new AcyclicSP(graph, st);

for (int i = 0; i < graph.V; ++i)

{

if (i == st)

{

Console.WriteLine("{0}->{0} : 0.0", st);

continue;

}

if (!sp.HasPathTo(i))

{

Console.WriteLine("{0}->{1} : No paths", st, i);

continue;

}

Double len = 0.0;

foreach (var edge in sp.PathTo(i))

{

Console.Write(edge);

Console.Write(",");

len += edge.Weight;

}

Console.WriteLine(" : {0}", len);

}

}

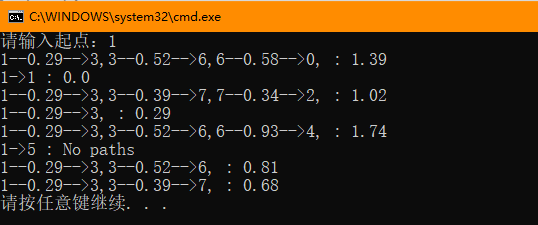
}

效率分析：

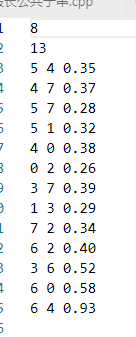
|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 要求 | 权重的比较次数 | 所需空间 |
| Dijkstra（最小堆） | 权重都为正 | E\*log(V) | V(保存索引最小堆) |
| Dijkstra（拓扑排序） | 无环有向图（由拓扑排序决定） | E+V | V(入度数组) |

结果

1. 使用最小堆优化：8个点13条边



数据:



1. 拓扑排序优化：250个点2546条边

结果（部分路径）：

1--0.07484-->107,107--0.0564-->69,69--0.11896-->128,128--0.03633-->78,78--0.10966-->77,77--0.10655-->187,187--0.04215-->231,231--0.11935-->191, : 0.66424

1--0.0955-->200,200--0.09984-->223,223--0.09036-->198,198--0.09053-->94,94--0.11772-->18,18--0.07335-->14,14--0.06649-->133,133--0.06257-->13,13--0.08128-->192, : 0.77764