二维费用旅行商问题的A*算法

A-star algorithm

软件学院 软件13班 吴业成

题目：DM2022 2-1 快递业务

题目大意：给定一张有向完全图，每条边有长度和费用两种代价，求图中的一条Hamilton路径，满足花费不超过阈值L且长度最短.
数据范围及时限：$3 \le n \le 18$，2s.

简单分析：求长度最短的Hamilton路径，就是常见的一种旅行商问题，最直接的做法就是用DFS枚举节点的排列，加上简单的最优性剪枝，以及本题对费用的可行性剪枝，然后不断更新答案即可.
这是最直观的做法，也就是常见的暴力搜索算法.

可是，这样就能A掉此题吗？

DM2022 2-1 快递业务

Case No.	Result	Time(ms)	Memory(KB)
1	Accepted	0	596
2	Accepted	0	624
3	Accepted	89	624
4	Time Limit Exceeded	≥2000	548
5	Time Limit Exceeded	≥2000	552
6	Time Limit Exceeded	≥2000	548
7	Time Limit Exceeded	≥2000	548
8	Time Limit Exceeded	≥2000	552
9	Time Limit Exceeded	≥2000	552
10	Time Limit Exceeded	≥2000	552

即使加上剪枝，暴搜也实在是太慢了！

思考：有什么改进算法?

- 分支限界法

- 还有其他更加高效且普适的算法吗？
- A*算法

简介

A*搜索算法（英文：A*search algorithm，A*读作 A-star），简称 A*算法，是一种在图形平面上，对于有多个节点的路径求出最低通过成本的算法。它属于图遍历（英文：Graph traversal）和最佳优先搜索算法（英文：Best-first search），亦是 BFS 的改进。

定义起点 s，终点 t，从起点（初始状态）开始的距离函数 $g(x)$，到终点（最终状态）的距离函数 $h(x)$，$h^*(x)$ [1]，以及每个点的估价函数 $f(x) = g(x) + h(x)$。

A*算法每次从优先队列中取出一个 f 最小的元素，然后更新相邻的状态。

如果 $h \leq h*$，则 A*算法能找到最优解。

上述条件下，如果 h 满足三角形不等式，则 A*算法不会将重复结点加入队列。

当 $h = 0$ 时，A*算法变为 Dijkstra；当 $h = 0$ 并且边权为 1 时变为 BFS。

A*算法（https://oi-wiki.org/search/astar）

前置知识1：堆优化的Dijkstra算法

课本介绍的Dijkstra算法是最常用的单源最短路算法，其时间复杂度为$O(n^2)$，对于稀疏图来说效率较低，通过二叉堆优化每一步找距离最小的节点的过程，可将时间复杂度优化至$O(m\log n)$，基本是复杂度最优秀的最短路径算法.

不细讲二叉堆，只提一下怎么用：
STL库有优先队列(priority queue)，已经为我们提供了二叉堆的实现

```
#include <queue>                                        //优先队列在queue头文件中
using namespace std;
priority_queue<int> q1;                                 //声明一个大根堆
priority_queue<int, vector<int>, greater<int> >q2;      //声明一个小根堆
q1.push(x);                                             //将x插入q1
int y=q1.top();                                         //y为q1堆顶(q1中最大元素)
q1.pop();                                               //弹出堆顶
int sz=q1.size();                                       //堆的大小
for(int i=1; i<=5; i++){
    q1.push(i);
    q2.push(i);
}
while(q1.size())printf("%d ",q1.top()), q1.pop();      //输出:5,4,3,2,1
while(q2.size())printf("%d ",q2.top()), q2.pop();      //输出:1,2,3,4,5
```

二叉堆：一种数据结构，支持$O(\log n)$插入，$O(\log n)$删除，$O(1)$查询最值. 是维护最值问题中常用的一种数据结构.

Dijkstra 算法描述如下：

a. 置 $\bar{S}=\{2,3,\cdots,n\}$，$\pi(1)=0$，$\pi(i)=\begin{cases} w_{1i} & i\in\Gamma_1^+ \\ \infty & \text{其他} \end{cases}$

b. 在 $\bar{S}$ 中，令

$$\pi(j)=\min_{i\in\bar{S}}\pi(i),$$

置 $\bar{S}\leftarrow\bar{S}-\{j\}$，

若 $\bar{S}=\Phi$，结束。否则转 c。

c. 对全部 $i\in\bar{S}\cap\Gamma_j^+$，置

$$\pi(i)\leftarrow\min(\pi(i),\ \pi(j)+w_{ji}),$$

转 b。

这一步是一个寻找全局路径距离最小的过程，如果直接扫描则复杂度为 $O(n)$ ，但查询最值如果用堆来做则可优化为$O(1)$.

利用二叉堆优化Dijkstra算法，就是在原算法过程中找出全局$dist$值最小节点这一步改用取出小根堆堆顶即可，不难给出以下代码：

```cpp
void dijkstra(int n, int s){
    memset(vis, 0, sizeof(vis));
    memset(dist, 0x3f, sizeof(dist));
    priority_queue<pair<int, int> >q;                 //声明大根堆, 元素为节点距离与节点构成二元组
    dist[s]=0;
    q.push(make_pair(-dist[s], s));                   //源点入堆, 一个trick:值取负即可将大根堆当作小根堆
    while(q.size()){
        int x=q.front().second; q.pop();              //找全局dist值最小节点
        if(vis[x])continue;                           //保证每个点只执行一次松弛操作
        vis[x]=true;
        for(int i=head[x]; i; i=Next[i]){             //松弛边
            int y=ver[i], z=len[i];
            if(dist[y]>dist[x]+z){
                dist[y]=dist[x]+z;
                q.push(make_pair(-dist[y], y));       //新点入堆
            }
        }
    }
}
```

C++实现堆优化Dijkstra算法

前置知识2：二进制状态压缩

在Hamilton路径问题中，每个状态需要维护所有节点的遍历状态，可以对每个节点开一个长为n的bool数组来实现. 但这种做法空间复杂度过高且效率较低.

注意到，可以按节点顺序将bool数组写成一个01字符串$\{0,1\}^n$，如010111表示节点0,2未访问，而节点1,3,4,5已访问过.

再进一步，由于每个二进制01串可以看作一个整数的二进制表示，所以可以直接利用一个整数$state$来表示整个bool数组，如010111可写作$2^1+2^3+2^4+2^5=58$. 在C++中，一个int型变量最多可维护31个节点组成的状态.

二进制状态压缩，就是将长为m的bool数组用一个m位二进制整数维护的方法.

同时，借助C++的位运算操作，可以实现对原bool数组中对应下标元素(从0开始)的存取

(n>>k)&1　　取出整数n在二进制下的第k位

n^(1<<k)　　将整数n在二进制下的第k位取反

n|(1<<k)　　将整数n在二进制下的第k位赋值1

状态压缩表示简便，且可以提升程序的时间效率和空间效率

应用举例：枚举集合$\{0,1,2,\cdots,n-1\}$的所有子集

```
for(int i = 0; i < 1 << n ; i++){
    for(int j = 0; j < n; j++)
        if((i >> j) & 1)printf("%d ", j);
    printf("\n");
}
```

Note: 借助状态压缩，可以利用动态规划的算法在$O(n^2*2^n)$的复杂度内解决一般的旅行商问题，感兴趣的同学可以参阅相关资料

A*算法

图论中的路径查找问题的三种算法：
BFS(普通队列)：从起点开始**盲目**地四处扩张，直到到达目标节点.
Dijkstra(优先队列)：每次选取**当前代价**最小的节点进行扩展，以便快速地接近目标节点.
A*(优先队列+启发式估价函数)：每次选取**当前代价+预估代价**最小的节点进行拓展，充分利用节点信息来评估节点的优先级，更快地接近目标节点.

举个例子，B同学想从D地走到C地，路上每一个地方P的当前代价即为B已经走过的距离，预估代价则是B对到C地还要走的距离的一个估计（如地图上P和C地的直线距离）.

形式上讲，A*算法会给每个节点一个优先级$f(node)$：

$$f(node) = g(node) + h(node)$$

$g(node)$是节点node距离起点的距离(如本题中已经选择边的长度和)
$h(node)$是节点node距离终点的估计代价(如本题中由当前状态扩展为完整的Hamilton路径还需选择边的长度估计)，也就是A*算法的启发式估价函数.
$f(node)$是每个节点在优先队列中的优先级，A*算法的搜索过程每一步都会选择f值最小的节点来扩展

启发式估价函数

启发式估价函数会影响A*算法的正确性和效率.
设$d(node)$为node到达目标节点的最短距离,当$h(node) > d(node)$时，A*算法所得到的结果不一定正确，但可以证明，当对任意节点node, $h(node) \le d(node)$，A*算法一定能给出最优解. 于是，在求精确解时A*算法要求h是乐观的，即$h(node) \le d(node)$恒成立.
$h(node)$越接近$d(node)$，A*算法的效率越高.
特别地，当$h(node) \equiv 0$时，A*算法退化为Dijkstra算法.

因此启发式估价函数的设计是A*算法的重点.
主要的设计思路与剪枝的设计思路相同，考虑距离下界.

几个例子：
在迷宫问题中，估价函数可以设计为当前点到终点的曼哈顿距离(两点的横坐标之差+纵坐标之差).
在地图上，估价函数可以设计为当前点到终点的欧式距离.
在八数码问题中，估价函数可以设计为当前状态与目标状态之间9个方格中的不同数字个数.

A*算法的实现与Dijkstra算法非常相似，只多了启发式估价函数的定义以及堆节点优先级的区别

现在我们可以给出A*算法的伪代码:

```
搜索节点：
    成员：状态、实际代价、实际代价与预估代价之和
    定义乐观估价函数h(状态)
    定义小于运算符(以实际代价与预估代价之和为键值)

A*()算法：
    声明搜索队列（优先队列）q
    初始节点s入队
    while(队q非空):
        取出队头节点u
        判断u是否是目标节点，若是则更新答案，结束
        if(u状态已经访问过)continue
        标记u状态
        从u扩展到其他状态v
            v入队
    结束
```

虽然讲的比较多，但A*算法核心代码写起来还是比较短的.

回到本题，每个搜索队列的节点node应包含一个状态整数state(状态压缩方法)，目前所在点，长度和费用代价以及长度和费用的优先级估价(f函数).
本题的估价函数$h(node)$可以设计为:

$$h(node) = \sum_{v \notin state} \min\{d(u,v), v \neq u\}$$

即与每个还未遍历节点相连的最短的边的长度之和.

然后拓展时，只让费用优先级不超过L的节点入队，每次取出长度优先级最大(值最小)的节点拓展即可

有了节点和乐观估价函数的定义，不难写出本题的A*算法代码.

结果如何？

DM2022 2-1 快递业务

Case No.	Result	Time(ms)	Memory(KB)
1	Accepted	0	604
2	Accepted	0	636
3	Accepted	0	692
4	Accepted	0	740
5	Accepted	0	944
6	Accepted	13	7312
7	Accepted	4	3000
8	Accepted	34	17052
9	Accepted	39	19040
10	Accepted	30	15692

可以看出，A*算法的效率还是比较高的

Q&A Time

分享到此结束，谢谢各位!
也欢迎大家和我交流!