搜索算法与迷宫问题

17341146 王程钥

1 算法原理

1.1 无信息搜索策略

1） 一致代价搜索

简单的宽度优先搜索从一个起点开始搜索，维护一个先进先出的队列，每次取队头拓展。一致代价搜索是在宽度优先搜索上的一个改进，它维护的是一个按代价排序的优先队列，每次取代价最低的未拓展节点进行拓展。图论算法中的 Dijkstra 算法就使用了一致代价搜索的思路。

值得注意的是，当每次拓展的花费相等时，一致代价搜索的搜索和 bfs 是本质相同的，因为此时可以保证先进队列节点代价一定不大于后进队列的节点。此时就完全没有必要使用一致代价搜索，使用一致代价搜索还会因为需要维护优先队列而在时间复杂度上增加一个 log。

2） 迭代加深搜索

迭代加深搜索枚举最大代价，每次以最大代价为界进行深度受限的深度优先搜索或宽度优先搜索。迭代加深搜索的一种优化是，对于每次失败的搜索保留下边界节点，下一次搜索时直接从这些边界节点继续往下搜索。

3） 双向搜索

双向搜索是一个简单且出色的搜索优化技巧。这种算法从起点和终点分别开始搜索，直到一个节点既被起点开始的搜索搜到又被终点开始的搜索搜到。对于稠密的搜索图，使用双向搜索优化可以使搜索效率得到很大的提升。

1.2 启发式搜索策略

1） A*搜索

和无信息搜索相比，启发式搜索可以利用问题所拥有的起发信息来引导搜索，已达到减少搜索范围，降低问题时间复杂度的目的。启发式搜索引入了估价函数的概念，设 f(x)=g(x)+h(x)，g(x)表示从初始节点到节点 x 付出的实际代价，即一致代价搜索中的代价，而 h(x)则表示从节点 x 到终点的估计代价。h(x)建模了启发式搜索问题中的启发信息，合理定义 h(x)可以大大优化搜索效率。

A*算法和一致代价搜索算法在流程上基本相同，唯一的不同是 A*算法维护的优先队列是按照 f(x)排序的，而一致代价搜索是按照 g(x)排序的。

2 算法流程

Algorithm1 宽度优先搜索/一致代价搜索
Input: 地图 s，起点(Sx,Sy)，终点(Tx,Ty) **Output:** 最短路径 将(Sx,Sy,0)加入队列 Queue While Queue 非空 do

```
    从队列 Queue 中取出队头(x,y,dis)
    For px,py in (x,y)的邻居 do
        If (px,py) 可被访问 and 未被访问
            将(px,py,dis+1) 加入队列 Queue
            记录(px,py)的前驱 Pre(px,py)=(x,y)
            If (px,py)=(Tx,Ty) 结束搜索
While (x,y)!=(Sx,Sy) do
    将(x,y)加入最短路径
    (x,y)转移到前驱 pre(x,y)
```

算法 1 为宽度优先搜索与一致代价搜索的代码，对于宽度优先搜索，Queue 为普通的先进先出的队列，对于一致代价搜索 Queue 为优先队列(堆)。

Algorithm2 迭代加深搜索+优化

```
Input: 地图 s, 起点(Sx,Sy), 终点(Tx,Ty)
Output: 最短路径
将(Sx,Sy,0)加入队列 Queue1, Queue2 为空队列
For dep in (1,inf)
    While Queue1 非空 do
        从队列 Queue1 中取出队头(x,y,dis)
        For px,py in (x,y)的邻居 do
            If (px,py) 可被访问 and 未被访问 and dis+1<=dep
                将(px,py,dis+1) 加入队列 Queue2
                记录(px,py)的前驱 Pre(px,py)=(x,y)
                If (px,py)=(Tx,Ty) 结束搜索
    Queue1=Queue2,Queue2=空队列
While (x,y)!=(Sx,Sy) do
    将(x,y)加入最短路径
    (x,y)转移到前驱 pre(x,y)
```

算法 2 为迭代加深搜索，两个队列均为先进先出的普通队列。

Algorithm3 双向搜索

```
Input: 地图 s, 起点(Sx,Sy), 终点(Tx,Ty)
Output: 最短路径
将(Sx,Sy,0,0)加入队列 Queue
将(Tx,Ty,0,1)加入队列 Queue
While Queue 非空 do
    从队列 Queue 中取出队头(x,y,dis,flag)
    For px,py in (x,y)的邻居 do
        If (px,py) 可被访问 and 未被访问
            将(px,py,dis+1,flag) 加入队列 Queue
            记录(px,py)的前驱 Pre(px,py)=(x,y)
        Else if (px,py) 的 flag 和(x,y)不同
            双向搜索聚集成功，修改 pre，退出循环
While (x,y)!=(Sx,Sy) do
    将(x,y)加入最短路径
    (x,y)转移到前驱 pre(x,y)
```

算法 3 为双向搜索，本质是一个多源宽度优先搜索。在起点和终点为源的搜索碰头后退出循环，结束搜索。

Algorithm4 A*搜索

Input: 地图 s，起点(Sx,Sy)，终点(Tx,Ty)
Output: 最短路径
将(Sx,Sy,0,H(Sx,Sy))加入队列 Queue
While Queue 非空 do
    从队列 Queue 中取出队头(x,y,dis,val)
    For px,py in (x,y)的邻居 do
        If (px,py) 可被访问 and 未被访问
            将(px,py,dis+1,dis+1+H(px,py)) 加入队列 Queue
            记录(px,py)的前驱 Pre(px,py)=(x,y)
            If (px,py)=(Tx,Ty) 结束搜索
While (x,y)!=(Sx,Sy) do
    将(x,y)加入最短路径
    (x,y)转移到前驱 pre(x,y)

算法 4 为 A*搜索，Queue 为优先队列，是按 val 维护的小根堆。

3 实现细节

```
int uni(int Sx,int Sy,int Tx,int Ty)
{
  int t1[]={-1,1,0,0};
  int t2[]={0,0,-1,1};
  // 四个方向
  int l=1,r=2,steps=0;
  int X[N*N],Y[N*N];
  X[1]=Sx;Y[1]=Sy;flag[Sx][Sy]=1;
  while(l<r)
  {
    int x=X[l],y=Y[l];l++;
    // 取队头
    for(int i=0;i<4;i++)
    {
      int xx=x+t1[i],yy=y+t2[i];
      // 枚举邻居
      if(xx<1||xx>n||yy<1||yy>m)continue;
      if(!flag[xx][yy]&&s[xx][yy]!='1')
      {
        X[r]=xx;Y[r]=yy;r++;steps++;
        f[xx][yy]=f[x][y]+1;
        flag[xx][yy]=1;pre[xx][yy]=i;
        // 加入队列
        if(xx==Tx&&yy==Ty)return steps;
```

```
      }
    }
  }
}
```

一致代价/宽度优先搜索

```
for(int dep=0;;dep++) // 枚举深度
{
  for(int i=1;i<=tot1;i++)
  {
    int x=X[i],y=Y[i];
    for(int j=0;j<4;j++)
    {
      int xx=x+t1[j],yy=y+t2[j];
      if(xx<1||xx>n||yy<1||yy>m)continue;
      if(!flag[xx][yy]&&s[xx][yy]!='1')
      {
        A[++tot2]=xx;B[tot2]=yy;steps++;
        f[xx][yy]=f[x][y]+1;
        flag[xx][yy]=1;pre[xx][yy]=j;
        if(xx==Tx&&yy==Ty)return steps;
      }
    }
  }
  for(int i=1;i<=tot2;i++)X[i]=A[i],Y[i]=B[i];
  tot1=tot2;tot2=0;
  // 修改清空队列
}
```

迭代加深搜索，其余部分与一致代价搜索相同，不做赘述。

```
int prep=i;
if(flag[x][y]==1)x=xx,y=yy;
else prep=(i^1);
while(1)
{
  if(x==Tx&&y==Ty)break;
  int xx=x-t1[pre[x][y]];
  int yy=y-t2[pre[x][y]];
  int tmp=pre[x][y];
  pre[x][y]=prep;
  prep=(tmp^1);
  x=xx;y=yy;
}
pre[Tx][Ty]=prep;
```

双向搜索的修改 pre 标记部分，其余部分与一致代价搜索相同，不同之处仅在于初始有两个节点进入队列。

```
int astar(int Sx,int Sy,int Tx,int Ty,int type=0)
{
  int t1[]={-1,1,0,0};
  int t2[]={0,0,-1,1};
  q.push((info){Sx,Sy,0,calc(Sx,Sy,Tx,Ty)});
  flag[Sx][Sy]=1;
  if(type)
  {
    q.push((info){Tx,Ty,0,calc(Sx,Sy,Tx,Ty)});
    flag[Tx][Ty]=2;
  }
  int steps=0,x,y,dis;
  while(1)
  {
    info tmp=q.top();q.pop();
    x=tmp.x;y=tmp.y;dis=tmp.dis;
    if(!type&&x==Tx&&y==Ty)break;
    for(int i=0;i<4;i++)
    {
      int xx=x+t1[i],yy=y+t2[i];
      if(xx<1||xx>n||yy<1||yy>m)continue;
      if(s[xx][yy]=='1')continue;
      if(flag[xx][yy]==flag[x][y])continue;
      if(!flag[xx][yy])
      {
        if(flag[x][y]==1){
          q.push((info){xx,yy,dis+1,dis+1+calc(xx,yy,Tx,Ty)});
        }
        else{
          q.push((info){xx,yy,dis+1,dis+1+calc(xx,yy,Sx,Sy)});
        }
        flag[xx][yy]=flag[x][y];pre[xx][yy]=i;steps++;
      }
      else
      {
        // 同双向搜索，略
        return steps;
      }
    }
  }
  return steps;
}
```

启发式搜索与双向启发式搜索。Type 变量决定是否双向，0 表示单向，1 表示双向。

4 实验结果与分析

4.1 问题描述与分析

本次实验要求使用搜索算法解决一个迷宫问题，该问题的迷宫大小是 18*37。

对于本问题，搜索每进行一步的代价是 1，因此一致代价搜索等价于宽度优先搜索，本次实验中的一致代价搜索的代码直接按照宽度优先搜索编写。

在所有算法中，我都使用了宽度优先搜索算法，并利用空间换时间的思想开了一个二维数组 f[x][y]用于存储起点到每个点(x，y)的最短距离。可以证明对于每个点，使用宽度优先搜索第一次拓展到的方案的代价一定是最小的。因此使用该方法，每个点只需要被转移一次。对于使用了优先队列优化的算法时间复杂度为 O(nm*log(nm))，对于没使用优先队列优化的算法时间复杂度为 O(nm)，其中 n 和 m 为地图大小。

因为 f 数组和地图的大小是同阶的，都是 O(n*m)，所以这种算法并不会提高算法的空间复杂度。

本次实验中，我使用了一致代价搜索，加优化的迭代加深搜索，双向搜索，A*搜索，双向 A*搜索这 5 种算法，并通过拓展次数比较 5 种算法的效率。这里的拓展次数是指进队列的节点的数量，无论该节点是否被拓展都会被记录。

表 1 记录了这 5 中算法的完备性，最优性，时间复杂度，空间复杂度。这里的时间复杂度包括两种时间复杂度，1 是这些算法的 global 的复杂度，2 是这些算法对于本问题的实际复杂度。表 2 为 5 中算法实际运行的性能。

算法	完备性	最优性	time1	space1	time2	space2
一致代价	Yes	Yes	$O(b^d)$	$O(b^d)$	$O(nm)$	$O(nm)$
迭代加深+优化	Yes	Yes	$O(b^d)$	$O(bd)$	$O(nm)$	$O(nm)$
双向 bfs	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$	$O(nm)$	$O(nm)$
A*搜索	Yes	Yes	$O(b^d)$	$O(b^d)$	$O(nmlog)$	$O(nm)$
双向 A*搜索	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$	$O(nmlog)$	$O(nm)$

(表 1)

算法	拓展次数(steps)
一致代价搜索(bfs)	269
迭代加深搜索+优化	269
双向 bfs	191
A*搜索	227
双向 A*搜索	180

(表 2)

4.2 启发式函数

我使用当前点到终点的曼哈顿距离作为估价函数。该估价函数的可采纳性显然，下证它的单调性。

证明：$H(x,y,Tx,Ty) = |x - Tx| + |y - Ty|$ 的单调性。

考虑一个状态(x,y)，考虑下一个状态(x',y')，如果(x',y')在(x,y)到(Tx,Ty)的最短路上，则$h(x,y) = h(x',y') + 1 = h(x',y') + cost(x,y,x',y')$，满足单调性约束。若不在最短路上，则$h(x',y') > h(x,y) + 1$，$h(x,y) < h(x',y') - 1 + 1 = h(x',y') - 1 + cost(x,y,x',y') < h(x',y') + cost(x,y,x',y')$，满足约束。

综上，定理得证。

如表 2，使用这种启发式函数的 A*算法可以取得不错的优化效果，和朴素的一致代价搜索相比，使用该方法在迷宫问题上可以减少 42 次拓展。

4.3 双向搜索优化

首先我尝试使用双向搜索优化一致代价搜索/bfs，即做一个多源 bfs。可以看到使用双向搜索拓展次数大大降低，只需要 191 次拓展就可以找到最优解。

我又尝试将双向搜索和启发式搜索和结合，使用双向启发式搜索进行实验。对于从起点开始的搜索以到终点的曼哈顿距离作为估价函数，对于从终点开始的搜索以到起点的曼哈顿距离作为估价函数。如表 2 可以看到使用双向 A*算法只需要 180 次拓展就可以完成搜索并得到正确的路径。这一结果比双向搜索和启发式搜索都更加优秀。

4.4 实验结果总结

如表 2，使用上文提到的 5 种算法搜索次数均在 300 次以内，其中最快的双向 A*搜索只需要拓展 180 次即可找到最优解。以上 5 种算法都可以找到最优路径，最优路径如表 3 所示。

```
1111111111111111111111111111111111
1000000000000000000000000##########S1
1011111111111111111111111#1111111101
1011000100010000001111111###11000001
1011010101010111101111111#11011111
1011010101010000000#####11#11000001
1011010101010101011110#111#####11111101
1010010101000100001#111111110000001
1011010101111111011#########11011111
1011010001100000001111111#11000001
1000011111101111111000000011#11111101
1111110000001000000001111011#10000001
1000000111111011111010000011#11011111
1011111100000010000000011111#11000001
1000000001111110111111111111#11001101
1111111111####################11111101
1E##########11111111111111111000000001
1111111111111111111111111111111111111
```

（表 3,红底#为最短路径）

5 思考题

问题：这些策略的优缺点是什么？它们分别适用于怎样的场景？

5.1 优缺点

1） bfs

优点：保证最优性和完备性。

缺点：空间复杂度较大，为完整状态空间。

2） 一致代价搜索

优点：保证最优性和完备性，对于无负代价的搜索等价于动态规划，保证局部最优性。

缺点：需要维护较复杂的数据结构(堆)，且空间复杂度较大，为完整状态空间。

3） 迭代加深搜索

优点：保证最优性和完备性，空间复杂度优秀。

缺点：不加优化会造成搜索冗余。

4）双向搜索

优点：保证最优性和完备性，对于稠密图可以大量减少搜索的状态数（状态数可以开根号），空间时间复杂度都很优秀。

缺点：不具有普适性，对于有些问题目标状态不唯一，无法进行双向搜索。

5）A*搜索

优点：保证最优性和完备性，对于无负代价的搜索等价于动态规划，保证局部最优性，引入估价函数提升搜索效率减少冗余搜索。

缺点：需要维护较复杂的数据结构(堆)，且空间复杂度较大，为完整状态空间。

6）IDA*搜索

优点：保证最优性和完备性，空间复杂度优秀，引入估价函数提升效率减少冗余搜索。

缺点：只能进行 dfs，如果进行 bfs 则和 A*搜索没有差别。使用 DFS 有时不方便对状态记忆化提高时间效率。

5.2 适用场景

1）对于一步拓展代价为 1 的搜索，宽度优先搜索等价于一致代价搜索。对于这种场景的无信息搜索一般采用简单的 bfs 即可。如果状态空间有限且搜索图单层节点不是很多的话，可以考虑使用循环队列优化的 bfs，空间复杂度和迭代加深相同。对于拓展代价不一定为 1 的搜索，更适合使用一致代价搜索，比如经典的求解最短路的算法 Dijkstra 算法。

2）对于起点状态和终点状态均已知的搜索，可以采用双向搜索优化。这可以大幅度减少搜索的状态数（开根号）。另外对于搜索步骤可分割的问题，也可以采用双向搜索优化（又称 meet in the middle，折半搜索）。比如经典的 01 背包问题，可以将 n 种物品分为两半，分别 2^n/2 次搜索，后将两次搜索的结果合并。

3） 迭代加深搜索及 A*搜索的迭代加深优化(IDA*)适用于状态空间很大，无法将所有状态都存下来的情况。

4）启发式搜索是常用的优化手段。一个好的启发式函数能够防止搜索误入歧途。

5）对于本次实验的迷宫问题，起始状态终止状态均已知，状态空间很小，有合适的估价函数，比较合适的方法是双向启发式搜索。从第 4 节的实验结果也可以看出这点。

6　结论

本次实验我设计了多种搜索策略，根据第四章的实验结果，最终使用双向 A*算法，仅用 180 步拓展就找到了迷宫的最短路。由此可见，双向搜索和启发式搜索可以优化迷宫问题的搜索过程，减少冗余搜索，提升效率。