**A\*算法：**

A\* [1]  （A-Star)算法是一种静态路网中求解[最短路径](https://baike.baidu.com/item/%E6%9C%80%E7%9F%AD%E8%B7%AF%E5%BE%84/6334920?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)最有效的直接[搜索方法](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E6%96%B9%E6%B3%95/19133144?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)，也是许多其他问题的常用[启发式算法](https://baike.baidu.com/item/%E5%90%AF%E5%8F%91%E5%BC%8F%E7%AE%97%E6%B3%95/938987?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)。注意——是最有效的直接[搜索算法](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E7%AE%97%E6%B3%95/2988274?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)，之后涌现了很多预处理算法（如CH），在线查询效率是A\*算法的数千甚至上万倍。

公式表示为： f\*(n)=g\*(n)+h\*(n),

其中， f\*(n) 是从初始状态经由状态n到目标状态的最小代价估计，

g\*(n) 是在[状态空间](https://baike.baidu.com/item/%E7%8A%B6%E6%80%81%E7%A9%BA%E9%97%B4/5129038?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)中从初始状态到状态n的最小代价，

h\*(n) 是从状态n到目标状态的路径的最小估计代价。

（对于[路径搜索](https://baike.baidu.com/item/%E8%B7%AF%E5%BE%84%E6%90%9C%E7%B4%A2/5135146?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)问题，状态就是图中的节点，代价就是距离）

真实h(n)的选取：

保证找到[最短路径](https://baike.baidu.com/item/%E6%9C%80%E7%9F%AD%E8%B7%AF%E5%BE%84?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)（最优解的）条件，关键在于估价函数f(n)的选取（或者说h(n)的选取）。

以h(n)表达状态n到目标[状态估计](https://baike.baidu.com/item/%E7%8A%B6%E6%80%81%E4%BC%B0%E8%AE%A1/1993607?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)的距离，那么h(n)的选取大致有如下三种情况：

如果h(n)< h\*(n)，这种[情况](https://baike.baidu.com/item/%E6%83%85%E5%86%B5?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/A%2A%E7%AE%97%E6%B3%95/_blank)下，搜索的点数多，搜索范围大，效率低。但能得到最优解。

如果h(n)=h\*(n)，此时的搜索效率是最高的。

如果 h(n)>h\*(n)，搜索的点数少，搜索范围小，效率高，但不能保证得到最优解。

**具体搜索过程**

一.首先需要创建一张地图，可以有障碍物但是必须进行标记。

二.设置路径的起点和终点。

三.开始搜索路径：

1.初始化open列表，close列表，pre列表和valueF列表；

2.将起点加入open列表，然后将其周围四个点（或八个点，由需求决定）加入到open列表。将起点从open列表中移除并移动到close列表；

1. 依次判断周围这四个点是否在close列表中且是否越界，如果不在，以此计算周围点的G，H并更新F，如果对应单元格在valueF中的值为初始化值或较大，那么更新单元格对应valueF值，记录pre值。伪代码如下：

if (node\_next.F < valueF[node\_next.x][node\_next.y] || valueF[node\_next.x][node\_next.y] == 0)

{

// 保存该节点的父节点

pre[node\_next] = node\_current; //将父亲点添加到pre，建立逻辑联系

valueF[node\_next] = node\_next.F; // 修改该节点对应的valF值

open.add(node\_next); //当前点添加到open列表

}

四.从周围的点中找出F最小的点，获得周围点的集合，然后将这个F最小的点从open列表中移除并移动到close集合中；

五．跳转第 3 步。

结束条件

终点单元格被加入open列表并且被作为当前格查询时；

open列表被清空，表示不可能到达终点。

**优缺点：**

优点：利用启发式函数减少了搜索的节点数量，因此在某些情况下比其他算法更高效。

缺点：需要设计和选择合适的启发式函数，且对于存在负权重边的情况不适用

**代码：**

#include <algorithm>

#include <cmath>

#include <iostream>

#include <queue>

#include <string>

#include <vector>

#define N 10 // 地图的阶数

using namespace std;

typedef struct NODE

{

int x, y; // 节点所在位置

int F, G, H; // G:从起点开始，沿着产的路径，移动到网格上指定方格的移动耗费。

// H:从网格上那个方格移动到终点B的预估移动耗费，使用曼哈顿距离。

// F = G + H

NODE(int a, int b) { x = a, y = b; }

// 重载操作符，使优先队列以F值大小为标准维持堆

bool operator<(const NODE &a) const

{

return F == a.F ? G > a.G : F > a.F;

}

} Node;

// 定义方向

//const int next\_position[8][2] = {{-1, -1}, {-1, 0}, {-1, 1}, {0, -1}, {0, 1}, {1, -1}, {1, 0}, {1, 1}};

const int next\_position[4][2] = {{-1, 0}, {0, -1}, {0, 1}, {1, 0}};

priority\_queue<Node> open; // 优先队列，就相当于open表

// 棋盘

int map[N][N] = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0},

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

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

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

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

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

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

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

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

{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}};

bool close[N][N]; // 访问情况记录，close列表

int valueF[N][N]; // 记录每个节点对应的F值

int pre[N][N][2]; // 存储每个节点的父节点

int Manhattan(int x, int y, int x1, int y1)

{

return (abs(x - x1) + abs(y - y1)) \* 10;

}

bool isValidNode(int x, int y, int xx, int yy)

{

if (x < 0 || x >= N || y < 0 || y >= N)

return false; // 判断边界

if (map[x][y] == 1)

return false; // 判断障碍物

// 两节点成对角型且它们的公共相邻节点存在障碍物，在8方向时用

if (x != xx && y != yy && (map[x][yy] == 1 || map[xx][y] == 1))

return false;

return true;

}

void Astar(int x0, int y0, int x1, int y1)

{

// 起点加入open列表

Node node(x0, y0);

node.G = 0;

node.H = Manhattan(x0, y0, x1, y1);

node.F = node.G + node.H;

valueF[x0][y0] = node.F;

open.push(node);

while (!open.empty())

{

Node node\_current = open.top(); //取优先队列头元素，即周围单元格中代价最小的点

open.pop(); //从open列表中移除

close[node\_current.x][node\_current.y] = true; // 访问该点，加入close列表

if (node\_current.x == x1 && node\_current.y == y1) // 到达终点

break;

// 遍历node\_top周围的4个位置，如果是next\_position有8，那么就需要遍历周围8个点

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

{

Node node\_next(node\_current.x + next\_position[i][0], node\_current.y + next\_position[i][1]); // 创建一个node\_top周围的点

// 该节点坐标合法 且没有被访问

if (isValidNode(node\_next.x, node\_next.y, node\_current.x, node\_current.y) && !close[node\_next.x][node\_next.y])

{

// 计算从起点并经过node\_top节点到达该节点所花费的代价

node\_next.G = node\_current.G + int(sqrt(pow(next\_position[i][0], 2) + pow(next\_position[i][1], 2)) \* 10);

// 计算该节点到终点的曼哈顿距离

node\_next.H = Manhattan(node\_next.x, node\_next.y, x1, y1);

// 从起点经过node\_top和该节点到达终点的估计代价

node\_next.F = node\_next.G + node\_next.H;

// node\_next.F < valueF[node\_next.x][node\_next.y] 说明找到了更优的路径，进行更新

// valueF[node\_next.x][node\_next.y] == 0 说明该节点还未加入open表中，则加入

if (node\_next.F < valueF[node\_next.x][node\_next.y] || valueF[node\_next.x][node\_next.y] == 0)

{

// 保存该节点的父节点

pre[node\_next.x][node\_next.y][0] = node\_current.x;

pre[node\_next.x][node\_next.y][1] = node\_current.y;

valueF[node\_next.x][node\_next.y] = node\_next.F; // 修改该节点对应的valueF值

open.push(node\_next);

}

}

}

}

}

void PrintPath(int x1, int y1)

{

if (pre[x1][y1][0] == -1 || pre[x1][y1][1] == -1)

{

cout << "no path to get" << endl;

return;

}

int x = x1, y = y1;

int a, b;

while (x != -1 || y != -1)

{

map[x][y] = 2; // 将可行路径上的节点赋值为2

a = pre[x][y][0];

b = pre[x][y][1];

x = a;

y = b;

}

// ' '表示未经过的节点， '#'表示障碍物， '@'表示可行节点

string s[3] = {" ", " #", " @"};

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

{

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

cout << s[map[i][j]];

cout << endl;

}

}

int main(int argc, char \*argv[])

{

fill(close[0], close[0] + N \* N, false); // 将visit数组赋初值false

fill(valueF[0], valueF[0] + N \* N, 0); // 初始化F全为0

fill(pre[0][0], pre[0][0] + N \* N \* 2, -1); // 路径同样赋初值-1

// // 起点 // 终点

int x0 = 2, y0 = 4, x1 = 8, y1 = 6;

// printf("input start: ");

// scanf("%d%d", &x0, &y0);

// printf("iinput destination: ");

// scanf("%d%d", &x1, &y1);

if (!isValidNode(x0, y0, x0, y0))

{

printf("Invalid input.\n");

return 0;

}

Astar(x0, y0, x1, y1); // A\*算法

PrintPath(x1, y1); // 打印路径

return 0;

}

**Johnson算法**

**Johnson算法是求稀疏图的多元最短路径的算法，权值可以为负，但是不可以有负环。Johson算法是Bellman-Ford算法, Reweighting(重赋权重)和Dijkstra算法的大综合。主要的思想是使用dijstra算法对每个结点求单源最短路，但是dijstra不能解决有负权值的边，所有需要给边重新赋值，且赋值后最短路径与原来的最短路径的距离和path相同。使用斐波那契堆作为优先级队列时时间复杂度是O(V^2logV + VE)。**

**Johnson算法具体步骤：**

**1.初始化，把一个node q添加到图G中，使node q 到图G每一个点的权值为0。**

**2.使用Bellman-Ford算法，从源点为q，寻找每一个点 v从q到v的最短路径h(v)，如果存在负环的话，算法终止。**

**3.使用第2步骤中Bellman-Ford计算的最短路径值对原来的图进行reweight操作（重赋值）：边<u,v>的权值w(u,v)，修改成w(u,v)+h(u)-h(v)。**

**4.最后，移去q，针对新图（重赋值之后的图）使用Dijkstra算法计算从每一个点s到其余另外点的最短距离。**

**（5.使用dijstra计算完最短路后，需要d(i,j) = d'(i,j) + h(v) - h(u)才是最后结果）**

优缺点

优点：通过引入辅助节点和重新赋权重的方式，将全源最短路径问题转化为多次的单源最短路径问题。

缺点：时间复杂度为O(V^2log(V) + VE)，其中V为节点数量，E为边数量。

**代码**

**#include** <iostream>**#include** <vector>**#include** <queue>**#include** <algorithm>**using** **namespace** std;

**struct** **Edge** {

**int** u, v, w;};vector**<**Edge**>** edges;vector**<**vector**<**Edge**>>** adj;vector**<int>** dist;

*// Bellman-Ford子过程***bool** **BellmanFord**(vector**<**Edge**>** **&**edges, **int** n, **int** s) {

dist.resize(n**+**1, INT\_MAX);

dist[s] **=** 0;

**bool** negtiveCycle;

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

negtiveCycle **=** false;

**for**(**auto** e: edges) {

**if**(dist[e.u] **<** INT\_MAX **&&** dist[e.v] **>** dist[e.u] **+** e.w) {

dist[e.v] **=** dist[e.u] **+** e.w;

negtiveCycle **=** true;

}

}

**if**(**!**negtiveCycle) **break**;

}

**return** negtiveCycle;}

*// Dijkstra子过程*vector**<int>** Dijkstra(**int** n, **int** s) {

priority\_queue**<**pair**<int**, **int>**, vector**<**pair**<int**, **int>>**, greater**<**pair**<int**, **int>>>** pq;

vector**<int>** dist(n**+**1, INT\_MAX);

vector**<bool>** vis(n**+**1, false);

dist[s] **=** 0;

pq.push({0, s});

**while** (**!**pq.empty()) {

**auto** node **=** pq.top();

pq.pop();

**int** u **=** node.second;

**if**(vis[u]) **continue**;

vis[u] **=** true;

**for**(**auto** e : adj[u]) {

**int** v **=** e.v, w **=** e.w;

**if**(dist[u] **<** INT\_MAX **&&** dist[v] **>** dist[u] **+** w) {

dist[v] **=** dist[u] **+** w;

pq.push({dist[v], v});

}

}

}

**return** dist;}

**int** **main**() {

**int** n **=** 5;

edges **=** {{1,2,3}, {3,2,4}, {1,3,8}, {2,4,1}, {2,5,7}, {1,5,**-**4}, {4,1,2}, {5,4,6}, {4,3,**-**5}};

*// 构建新图 G'，新增结点 s，并将设置到各结点的权重为0，即： w(s,v)=0 v in G.V* **int** s **=** 0;

vector**<**Edge**>** edges\_new **=** edges;

**for**(**int** v **=** 1; v **<=** n; **++**v) {

edges\_new.push\_back({s, v, 0});

}

*// 执行 BellmanFord算法，计算s的单源最短路径* **bool** hasCycle **=** BellmanFord(edges\_new, n, s);

**if**(hasCycle) cout **<<** "the input graph contains a negative-weight cycle" **<<** endl;

*// 重新赋予权重 w'(u,v) = w(u,v) + h(u) - h(v)* **for**(**auto** **&**e : edges\_new) {

e.w **=** e.w **+** dist[e.u] **-** dist[e.v];

}

*// 构建新图G'的邻接表，Dijkstra算法需要使用到* adj.resize(n**+**1);

**for**(**auto** **const** e : edges\_new) {

adj[e.u].push\_back(e);

}

*// 在新图上以每个顶点为源点，执行Dijkstra算法，计算单源最短路径* vector**<**vector**<int>>** dist\_all(n**+**1, vector**<int>**(n**+**1, 0));

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

**auto** d **=** Dijkstra(n, i);

**for**(**int** j **=** 1; j **<=** n; **++**j) {

**if**(d[j] **==** INT\_MAX) **continue**;

dist\_all[i][j] **=** d[j] **+** dist[j] **-** dist[i]; *// 恢复权值* }

}

**return** 0;}

**文献来源：**

# **A\*算法(A-star Algorithm)搜索最短路径 -csdn作者** [A91A981E](https://blog.csdn.net/m0_46304383" \o "A91A981E" \t "https://blog.csdn.net/m0_46304383/article/details/_blank)

# **路径规划——A\*算法 知乎作者 [搬砖的旺财](https://www.zhihu.com/people/happy_siyuzhou" \t "https://zhuanlan.zhihu.com/p/_blank)**

# **所以结点对的最短路径：Johnson算法 知乎作者 [Kalai](https://www.zhihu.com/people/junge-45" \t "https://zhuanlan.zhihu.com/p/_blank)**

**johnson最短路径 csdn作者** [\_zhj](https://blog.csdn.net/zhj_fly" \o "_zhj" \t "https://blog.csdn.net/zhj_fly/article/details/_blank)