人工智能lab8 实验报告

> 姓名：TRY
>
> 学号：
>
> 专业：计算机科学与技术

> 时间：2020/11/24

一、算法原理

1.1 一致代价搜索

- 本次实验选择的无信息搜索是一致代价搜索（Uniform-cost search, UCS）。
- UCS算法是**BFS算法的变种**。BFS是在等待被扩展的边界队列中，选择队内深度最浅的节点进行扩展；而USC不再扩展深度最浅的节点，而是通过比较路径消耗，对边界队列内的节点进行排序，并选择当前代价最小的节点进行扩展。因此， UCS可以保证无论每步代价是否一致，都能够找到最优解。
- 不过在本次实验的迷宫问题中，由于每一步的代价都是相同的，所以UCS在这个问题上实际退化为BFS。

1.2 $A*$搜索

- 本次实验选择的启发式搜索是$A*$搜索。
- $A*$算法是一种**启发式搜索算法**，可以看成是UCS的升级版，在原有的UCS上就爱上了启发式信息。从起始节点开始，不断查询周围可达节点的状态并计算它们的$f(n), g(n), h(n)$ 的值（可以选择不同的启发式函数），选择估价函数$f(n)$最小的节点进行扩展，并更新已经访问过的节点的$g(n)$，直到达到目标节点。
  - 优点：省略大量无畏的搜索路径，提高搜索效率。
  - 缺点：空间复杂度较高。
- 算法可以用公式 $f(n) = g(n) + h(n)$ 来表示。其中，$f(n)$ 是从初始点经由节点n到目标点的估价函数， $g(n)$ 是状态空间中从初始节点到n节点的实际代价， $h(n)$ 是从n 到目标节点最佳路径的估计代价。
- **不同的启发式函数**$h(n)$
  - **曼哈顿距离**：在正方形网络中，允许向4邻域移动：

    $$h(n) = D * (abs(node.x - goal.x) + abs(node.y - goal.y))$$

    其中，D可设置为方格间移动的最短距离，`node`表示当前节点，`goal`表示目标节点。
  - **欧式距离**：在正方形网络中，允许想任意角度移动：

    $$h(n) = D * \sqrt{(node.x - goal.x)^2 + (node.y - goal.y)^2}$$
  - **对角线距离**（切比雪夫距离）：在正方形网络中，允许朝着对角线方向移动：

    $$h(n) = D * max(abs(node.x - goal.x), abs(node.y - goal.y))$$

        - 不过，在迷宫问题中，只能向4邻域方向移动，所以只能用曼哈顿距离或者欧氏距离来作启发式函数，不可使用对角线距离。并且，这两种距离都保持了一致性。
- 启发式函数通常包含两个特性：
    - 一个是**可采纳性**： $h(n) \leq h^*(n)$ 。这个特性保证了估计值要小于真实值，可采纳性也就意味着最优性。
    - 二是**单调性（一致性）**： $h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2)$ 。可以理解就是$h(n_i)$随着i的增加要和$h^*(n_i)$越来越接近。
    - 只要启发式函数具有一致性，就能在进行环检测之后仍然保持最优性。但如果启发式函数只是可采纳的而非单调的，进行了环检测之后就不一定能保持最优性。

二、算法性能分析

算法	完备性	时间复杂度	空间复杂度	最优性
UCS算法	Yes	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^{1+\lfloor C^*/e \rfloor})$	Yes
$A*$算法	depends on $h(x)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^{1+\lfloor C^*/e \rfloor})$	depends on $h(x)$

- **完备性：**
    - UCS算法：假设每个路径的成本s都大于0，由于UCS是BFS的变种，如果存在路径到达目标点，就一定能搜索出一条路径。
    - $A*$算法：如果启发式函数具有一致性，则算法具有完备性。
- **时空复杂度：**
    - UCS算法：UCS和BFS的时空复杂度都是一样的。假设C^*是起点到终点点最优情况下总代价，而e是整个图中所有路径最小的代价；则从起点到终点**至多**要走$\lfloor C^*/e \rfloor + 1$次（也就是搜索树最差情况下的深度），$b$为每个点的最大分叉数量。
        - 在实验的迷宫的问题，因为UCS退化成了BFS，每个点最多只能向四个方向拓展，也就是b为4，则时空复杂度为$O(4^{d+1})$（d为到终点的最优解需要的步数）。
    - $A*$算法：当$h(n) = 0$的时候，启发式函数是单调的，此时$A*$搜索就变成了一致代价搜索。因此一致代价搜索的时空复杂度的下界也适用于$A*$搜索。也就是说，它仍可能是指数的时空复杂度（除非能找到特别好的启发式函数$h(n)$）。
- **最优性：**
    - UCS算法：假设每条路径之间的成本s是大于0的，若将任意点作为目标点，UCS都能保证第一次探索到目标点的时候是最小的路径，否则当前边界会先选择探索另外一个点而不是目标点。
    - $A*$算法：如果启发式函数具有一致性，则算法具有完备性。

三、伪代码

- USC的伪代码：

```
Function USC
Input: maze, S, E   /*迷宫，起点，终点*/
Output: curNode or None, X, Y /*found_route:curNode,not_found:None;X,Y saves
the explored nodes*/

count := 1  /*time complexity*/
```

```
length := 1 /*space complexity*/
S := StartNode, E := EndNode
frontier := a priority queue by cost
explored := 2D lists with dafault "False"   /*False:unexplored,
True:explored*/
frontier.append(S)
X, Y = empty lists
actions =[[1,0],[-1,0],[0,1],[0,-1]]    /*up/down/left/right*/
while True:
    if empty(frontier) then     return None, X, Y   end if
    curNode := frontier.pop()
    if curNode = E then     return curNode, X, Y    end if
    explored[curNode] := True
    count += 1
    insert curNode into X, Y lists
    for action in actions:
        newNode := curNode + action
        if newNode is out of maze then   continue   end if
        if !explored(newNode) and maze(newNode)!='1' then
            if newNode in frontier and newNode.cost< frontier_node.cost then
                replace frontier_node with newNode
            else if newNode not in frontier then
                frontier := insert(forntier,newNode)
            end if
        end if
    end for
end while
```

- $A*$搜索算法的伪代码：整体与USC伪代码相同，就是新建节点的时候需要传入`node.estimate`（也就是$h(n)$）的值，且排序比较的时候将$g(n)$之间的大小比较变成了$g(n)+h(n)$的大小比较，即加上当前结点的启发式函数值的结果。

```
Function A_search
Input: maze, S, E   /*迷宫，起点，终点*/
Output: curNode or None, X, Y /*found_route:curNode,not_found:None;X,Y saves
the explored nodes*/

count := 1  /*time complexity*/
length := 1 /*space complexity*/
S := StartNode, E := EndNode
frontier := a priority queue by cost
explored := 2D lists with dafault "False"   /*False:unexplored,
True:explored*/
frontier.append(S)
X, Y = empty lists
actions =[[1,0],[-1,0],[0,1],[0,-1]]    /*up/down/left/right*/
while True:
    if empty(frontier) then     return None, X, Y   end if
    curNode := frontier.pop()
    if curNode = E then     return curNode, X, Y    end if
    explored[curNode] := True
    count += 1
    insert curNode into X, Y lists
    for action in actions:
        newNode := curNode + action
        if newNode is out of maze then   continue   end if
        if !explored(newNode) and maze(newNode)!='1' then
```

```
            if newNode in frontier and newNode.cost+newNode.estimate<
frontier_node.cost+frontier_node.estimate then
                replace frontier_node with newNode
            else if newNode not in frontier then
                frontier := insert(forntier,newNode)
            end if
        end if
    end for
end while
```

四、代码实现

4.1 UCS搜索

- 定义一个节点类，记录每个点的坐标、到达这个点的最小花费、祖先节点。并重定义比较符号，使得class可以根据cost来比较大小（优先队列中使用）：

```
class Node:
    def __init__(self, location, estimate=0, cost=0, father=None):
        self.location = location    # tuple: [x,y]
        self.cost = cost    # g(x)
        self.father = father
        self.estimate = estimate    # h(x)

    def __lt__(self, other):        # redefine '<'(used in heappush)
        return self.cost + self.estimate < other.cost + other.estimate
```

- 对于`frontier`的排序，使用优先队列，具体实现使用了小顶堆。主要使用到的操作如下：

```
import heapq
heapq.heappush(frontier,S)  # insert
curNode=heapq.heappop(frontier)     # delete the min one
heapq.heapify(frontier)     # update the heap
```

- UCS函数：定义了一致代价搜索的过程

```
def usc_search(maze, S, E):       # USC search
    count = 1   # time-complexity
    length = 1  # space-complexity
    row, col = len(maze), len(maze[0])
    frontier = []   # use heapq to construct a priority-queue
    explored = [[False] * col for _ in range(row)]      # False:unexplored,
True:explored
    heapq.heappush(frontier, S)     # push S to heapq
    X, Y = [], []   # to plot
    while 1:
        if len(frontier) == 0:  # frontier is empty
            return None, X, Y
        curNode = heapq.heappop(frontier)   # current explored node
        if curNode.location == E.location:  # find target node
            X.append(curNode.location[1])
            Y.append(-curNode.location[0])  # '-' since up/down is converse
when constructing the maze(readfile)
```

```
            print("Time complexity is %d" % count)
            print("Space complexity is %d" % length)
            return curNode, X, Y
        # update explored for curNode
        explored[curNode.location[0]][curNode.location[1]] = True
        count += 1  # update time
        X.append(curNode.location[1])
        Y.append((-curNode.location[0]))

        for action in actions:
            new_location = [x + y for x, y in zip(curNode.location, action)]
            # judge whether new_location is valid or not
            if new_location[0] < 0 or new_location[0] >= row or
new_location[1] < 0 or new_location[1] >= col:
                continue
            # if new_location isn't explored and can be explored
            if explored[new_location[0]][new_location[1]] is False and
maze[new_location[0]][new_location[1]] != '1':
                flag = True     # symbolizes whether new_location is in
frontier
                for node in frontier:       # check whether new_node is
already in the frontier
                    if node.location == new_location and node.cost >
curNode.cost + 1:  # if in and cost is less, update
                        node.cost = curNode.cost + 1
                        node.father = curNode
                        heapq.heapify(frontier)     # update heapq
                        flag = False
                        break
                if flag:    # if new_location not in frontier, insert into
frontier
                    new_node = Node(new_location, curNode.cost + 1, curNode)
                    heapq.heappush(frontier, new_node)
                    if len(frontier) > length:      # update length
                        length = len(frontier)
```

- 画出最优路线图`draw_route`：通过回溯法，根据节点的`father`值来回溯到起点，输出路径（不是探索路径，只是从起点到终点的一条最优路径）
    - 由于读取文件的时候是从第一行开始读取并构建`location`的，所以其实真正构建的`x`,`y`坐标和实际`maze.txt`中的图像呈对称关系，画图时为了呈现出`maze.txt`的样子，需将坐标反转。

```
def draw_route(node, S, E):
    X, Y = [], []
    curNode = node
    while curNode is not None:
        X.append(curNode.location[1])
        Y.append(-curNode.location[0])
        curNode = curNode.father
    plt.figure(3)
    plt.xlim(0, 36)  # 0~35
    plt.ylim(-18, 0)  # -17~0
    plt.title('Route of UCS')
    plt.xlabel('x')
    plt.ylabel('y')
```

```
    plt.annotate('S', xy=(S.location[1], -S.location[0]), xytext=
(S.location[1] + 0.5, -S.location[0] + 0.1))
    plt.annotate('E', xy=(E.location[1], -E.location[0]), xytext=
(E.location[1] - 0.5, -E.location[0] + 0.5))
    plt.plot(X, Y)      # continuous graph
    plt.show()
```

- 画出探索过的几点`draw_explored`：利用USC函数返回的`X`,`Y`数组，将探索过的节点用散点图画出

```
def draw_explored(X, Y):
    plt.figure(2)
    plt.xlim(0, 36)     # 0~35
    plt.ylim(-18, 0)    # -17~0
    plt.title('Explored nodes of UCS')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.annotate('S', xy=(S.location[1], -S.location[0]), xytext=
(S.location[1] + 0.5, -S.location[0]+0.1))
    plt.annotate('E', xy=(E.location[1], -E.location[0]), xytext=
(E.location[1] - 0.5, -E.location[0]+0.5))
    plt.scatter(X, Y)   # discrete nodes
    # plt.plot(X,Y)
    plt.show()
```

4.2 $A*$搜索

- 主要函数和USC搜索相同，主要不同在于类的构造：需要加上$h(n)$的值，并修改`__lt__`函数。

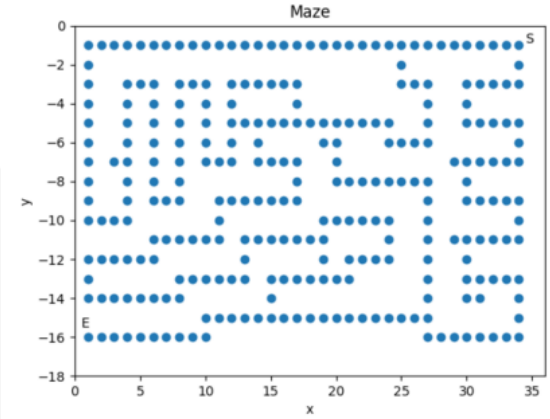
```
class Node:
    def __init__(self, location, estimate=0, cost=0, father=None):
        self.location = location    # tuple: [x,y]
        self.cost = cost    # g(x)
        self.father = father
        self.estimate = estimate    # h(x)

    def __lt__(self, other):        # redefine '<'(used in heappush)
        return self.cost + self.estimate < other.cost + other.estimate
```

- 计算启发式函数：这里计算了曼哈顿距离和欧式距离

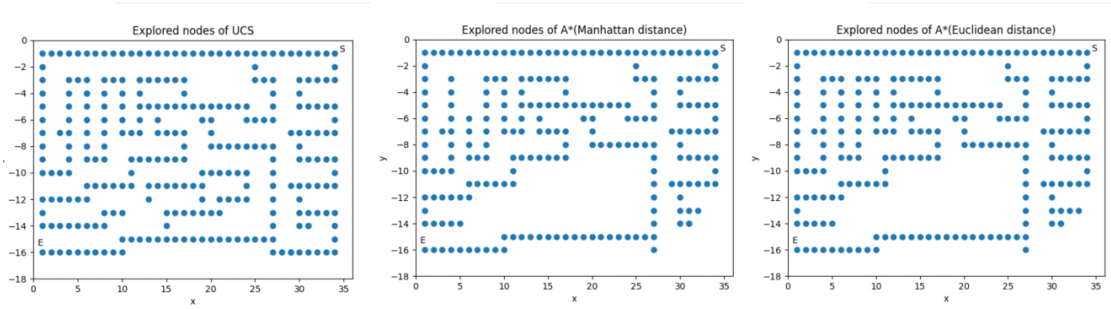
```
def dis_estimate(S, E, func_type):
    if func_type == 'Manhattan distance':
        return abs(S[0] - E[0]) + abs(S[1] - E[1])      # manhattan distance
    elif func_type == 'Euclidean distance':
        return np.sqrt((S[0] - E[0])**2 + (S[1] - E[1])**2)
        #return np.sqrt(np.square(S[0] - E[0])+np.square(S[1] - E[1])) 两种写法都可以!
```

五、实验结果及分析

- **迷宫**maze上所有可以走的点如下：

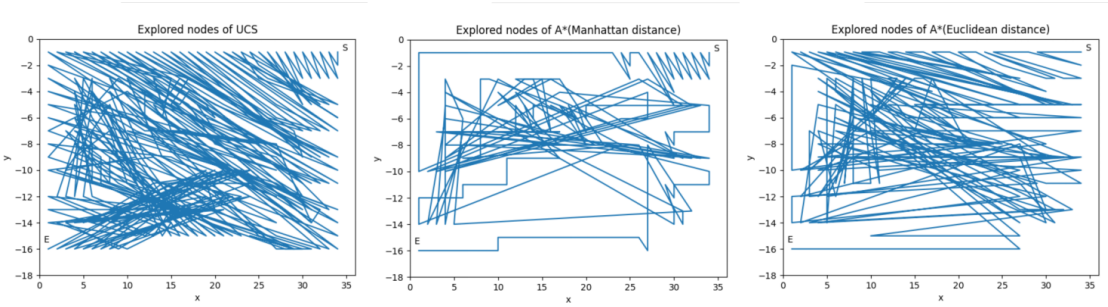


- 三种方法（UCS, A* search with Manhattan/ Euclidean distance）在遍历过程中会**探索的节点**如下：



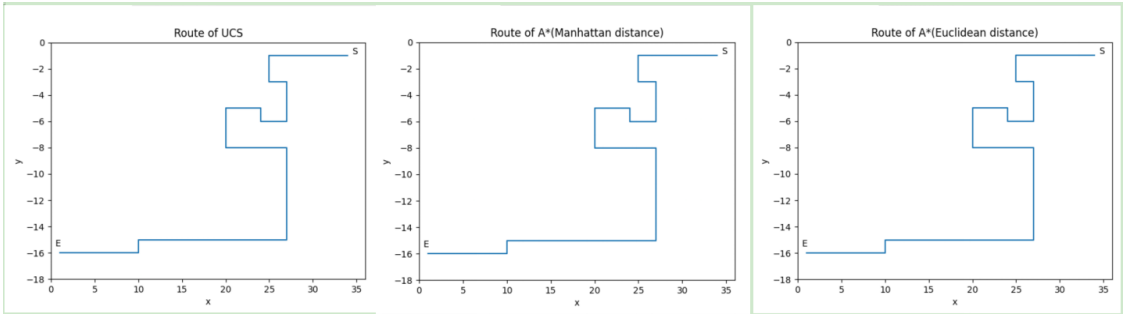
分析：观察可以发现，**UCS**除了在倒数第四行有几个点没有遍历之外，基本上所有迷宫中可走的点都遍历探索过；而使用$A*$ **算法**探索的节点要明显少于UCS探索的节点和迷宫的节点，且使用**曼哈顿距离**作为启发式函数的$A*$算法探索的节点在第3、4、5行（左上角部分）要少于以**欧氏距离**作为启发式函数的A*算法。因此，可以初步看出曼哈顿距离要优于欧氏距离。

- 三种方法在遍历探索过程中的**节点顺序**如下：（使用直线连接相邻的探索节点）



分析：通过此图可以明显发现，由于**UCS**是只考虑了$g(n)$，因此会“反复横跳”，探索间接且缓慢；而$A*$算法考虑了$g(n)$和$h(n)$的影响，效果和速度明显优于UCS的探索速度。且与**欧拉距离**相比，可以看出使用**曼哈顿距离**探索更加直接快速（少了开始的反复横跳过程）。笔者猜测，这与迷宫只能沿“上、下、左、右”四个方向行走的原因有关，因此使用曼哈顿距离作为启发式函数会有更好的结果。（欧拉距离可以走8个方向）

- 三种方法得到的**最优路径**如下：



分析：可以发现，使用UCS或者$A*$算法得到的最优路径都相同。

- 对比三种方法的时间复杂度和空间复杂度：

指标	**UCS**	**$A*$搜索(with Manhattan distance)**	**$A*$搜索(with Euclidean distance)**
时间复杂度	277	224	231
空间复杂度	9	8	8

  - 其中，**时间复杂度**以遍历`frontier`队列一次（出栈pop一次）为一次时间衡量指标。**空间复杂度**使用`frontier`遍历过程中的最大长度。
  - 比较**时间复杂度**，发现两种$A*$搜索的效率都要明显优于UCS搜索。且由于迷宫问题的设置，使用曼哈顿距离得到的效率要稍微优于欧氏距离。
  - 比较**空间复杂度**，发现使用启发式函数确实降低了空间复杂度，但由于搜索的深度并不大，每个点可选分枝数也不是很多，所以差距没有明显体现。

六、思考题

这些策略的优缺点是什么？ 它们分别适用于怎样的场景？

- **一致代价搜索：**本质上是BFS的变种，具有最优性，能确保搜索第一次到某一个点是沿着最优的路径搜索到的。但和BFS一样，边界队列需要经常保存指数级的节点数量， 空间复杂度高。由BFS的特点，可知对于终点深度小的情况下比较快。因此，适合解决最短或最少问题。
- **迭代加深搜索：**迭代加深搜索本质上是一个DFS，DFS的好处是空间复杂度小，因为不需要边界队列来保存所有的节点，适用于对空间复杂度要求较高的环境。但DFS的缺点是DFS不具有最优性，且深度很大的时候，搜索效率很低（如果恰好进入了一条很深的路，走到尽头再回溯的时间复杂度较高）。迭代加深搜索是对DFS缺点的改进，它限制了搜索深度，搜索到一定深度后就不再往更深的地方去搜索。然而，这个方法的缺点是深度不好设置，可能需要在实验过程中一步步增加深度。这可能会导致已经访问的点被反复重复访问，在搜索深度不高，指数级不大的时候效率会显得很慢。
- **双向搜索：**双向搜索适用于已知起点和终点位置状态的情况，从起点和终点两个方向同时开始搜索，可以明显提高单向BFS的搜索效率（搜索深度变成了$d/2$层）。缺点是需要维护两个边界队列，空间复杂度较高，而且必须要知道终点位置才能使用。
- **$A*$搜索：** 本质上也是BFS的变种，加入了启发式函数$h(n)$对边界序列进行排序，它的适用场景和一致代价搜索一样，都比较适合解决最短路径问题。它还融入了启发式函数对终点的已知信息，将信息用在搜索上，加快搜索进程。但如果$h(n)$都为0，即没有有效的信息，或者启发式函数设置不好，也就是信息无用甚至误导搜索，反而可能让搜索效率降低。
- **$IDA*$搜索：** 本质是对迭代加深搜索的再进一步改进，融入了启发式函数对终点的已知信息，适用于对终点位置有已知有效的信息的时候，能将信息用在搜索上，加快搜索进程。它的优缺点和迭代加深搜索都一样，优点是空间复杂度较低，缺点是不具有最优性。