人工智能基础第一次编程

自02 彭程 2020011075

一、无信息搜索的实现

无信息搜索有宽度优先、一致代价、深度优先、深度受限、迭代加深等算法，在本次作业中，我采用了宽度优先搜索。按照老师讲义上的伪代码实现如下：

```
def search_without_info(problem):
    """
    YOUR CODE HERE
    """
    print("无信息搜索")
    # 思路，广度优先搜索，维护一个队列和一个列表
    qu = Queue()
    close = []

    # 维护一些初始化的信息
    parking = problem
    node_start = parking.init_state
    node_end = parking.goal_state
    qu.push(node_start)

    # 开始广度优先搜索
    while not qu.empty():
        node = qu.pop()
        state = node.state
        close.append(state)
        actions = parking.actions(state)
        for action in actions:
            target, empty_pos = action[0], action[1]
            new_node = node.child_node(parking, empty_pos, target)
            if new_node == node_end:
                return new_node
            if new_node.state not in close and qu.find(new_node) is None:
                # print(f'depth:{new_node.depth}')
                qu.push(new_node)

    return 0
```

二、有信息搜索的实现

有信息搜索有贪婪最佳优先搜索、A*搜索等，这里我实现了A*算法。

首先，需要定义启发函数，我将启发函数定义为所有数码到其最终位置的曼哈顿距离之和。但是多目标曼哈顿函数的精准计算情况较多，时间代价高。为了提高效率，我将多目标的曼哈顿函数简化为相同车辆的坐标和与该类车辆目标位置的坐标和之间的曼哈顿距离。可以看出该近似小于等于真实的曼哈顿距离，小于等于真实代价，保证了启发函数的正确性。

具体代码实现如下：

```
def h(self,state):
    """
        cost predict function
        input: two matrix
        output: a predicted distance
    """
    goal = self.goal_state.state
    bike = np.argwhere(np.array(state) == 1)
    bike_g = np.argwhere(np.array(goal) == 1)
    h_bike = np.absolute(np.sum(bike,axis=0)-np.sum(bike_g,axis=0))
    h_bike=np.sum(h_bike)

    ebike= np.argwhere(np.array(state) == 2)
    ebike_g = np.argwhere(np.array(goal) == 2)
    h_ebike = np.absolute(np.sum(ebike, axis=0) - np.sum(ebike_g, axis=0))
    h_ebike = np.sum(h_ebike)

    return h_bike+h_ebike
```

有信息搜索的代码部分：

```
def search_with_info(problem):
    """
    YOUR CODE HERE
    """
    print("有信息搜索。")
    # 思路 A*搜索，用优先队列和一个列表
    # 优先级： F = G（移动代价） + H（预估代价）

    # 维护一些初始化的信息
    parking = problem
    node_start = parking.init_state
    node_end = parking.goal_state
    pqu = PriorityQueue(node_start, parking)
    close = []

    # 开始A*搜索
    while not pqu.empty():
        node = pqu.pop()
        state = node.state
        close.append(state)
        actions = parking.actions(state)
        for action in actions:
            target, empty_pos = action[0], action[1]
            new_node = node.child_node(parking, empty_pos, target)
            if new_node == node_end:
                return new_node
            if new_node.state not in close and pqu.find(new_node) is None:
                # print(f'depth:{new_node.depth}')
                pqu.push(new_node)

    return 0
```

相较于无信息搜索的代码，此处差异仅在于将原本的队列换为优先队列。

三、两种求解方法实现效果讨论

首先，从原理上分析，两种方法都具有完备性和最优性。但由于 A*算法不会扩展 f(n) > C*的节点，有效地实现了剪枝，在所有从开始状态搜索的算法中效率最优，在时间和空间复杂度上要比一致代价搜索更胜一筹。

其次，从程序运行效果来看，对于给定的样例，采用无信息搜索（一致代价搜索）很难在短时间内解出答案，程序最终运行了超过一小时才得到最终结果。而采用有信息搜索则可以在很短的时间内完成解答，具体实现结果如下：

无信息搜索结果如下：

```
depth:15
[[1, 1, 1, 0], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 0, 1], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 0, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [0, 2, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [1, 2, 0, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [1, 2, 2, 0], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [2, 0, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [0, 1, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 0, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 0, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 2, 0], [2, 2, 2, 2]]
```

有信息搜索结果如下：

```
有信息搜索。
time cost:2.4032135009765625
nodes change is as shown:
[[1, 1, 1, 0], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 0, 1], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 0, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 2, 0], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [2, 0, 2, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 2, 2, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [2, 0, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [0, 1, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 0, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 0, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 2, 0], [2, 2, 2, 2]]
depth:15
```

可见在此处A*搜索要比一致代价搜索要快得多。从这两个角度来看，有信息搜索要比无信息搜索更优。但是这只是我们所遇到的情况，虽然在很多时候有信息搜索可以节省大量的时间和空间，但是对于大规模问题坚持找最优解仍然不可行，这时候我们可能会通过深度优先搜索等无信息搜索等去替代，所以对于这两种搜索的优劣我们应该辩证地看待。

四、补充：关于代码效率的优化

在完成了基本任务后，我依然百思不得其解，为何代码效率如此之低下，经过对各部分进行计时我发现，时间代价最大的地方出现在对新节点是否在close和frontier里的判断上，为此我们考虑将close和frontier的查找维护成hash的，这样搜索复杂度将从O(n)下降到O(1)。在经过了这样的改动后，以下是代码和结果：

无信息搜索（采用BFS）：

代码如下(search_without_info_pro)：

```
def search_without_info_pro(problem):
    """
    相比于#pro版本，此版本的查找都是hash的
    """
    print("无信息搜索")
    # 思路，广度优先搜索
    qu = Queue()
    close = Set()
    open = Set()
    # 维护一些初始化的信息
    parking = problem
    node_start = parking.init_state
    node_end = parking.goal_state
    qu.push(node_start)
    open.add(np.array(node_start.state).tobytes())
    # 开始广度优先搜索
    while not qu.empty():
        node = qu.pop()
        state = node.state
        strstate = np.array(node.state).tobytes()  # tobytes enables hash
        open.remove(strstate)
        close.add(strstate)
        actions = parking.actions(state)
        for action in actions:
            target, empty_pos = action[0], action[1]
            new_node = node.child_node(parking, empty_pos, target)
            if new_node == node_end:
                return new_node
            if not close.find(np.array(new_node.state).tobytes()) and not open.find(np.array(new_node.state).tobytes()):
                qu.push(new_node)
                open.add(np.array(new_node.state).tobytes())
    return 0
```

结果如下：

```
请分别使用有信息和无信息搜索方法求解单车整理问题。
无信息搜索
time cost:14.257017612457275
nodes change is as shown:
[[1, 1, 1, 0], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 0, 1], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 0, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [0, 2, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [1, 2, 0, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [1, 2, 2, 0], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [2, 0, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [0, 1, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 0, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 0, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 2, 0], [2, 2, 2, 2]]
depth:15

Process finished with exit code 0
```

有信息搜索（采用A*）：

代码如下(search_with_info_pro)：

```
def search_with_info_pro(problem):
    """
    相比于#pro版本，此版本的查找都是hash的
    """
    print("有信息搜索-hash")
    # 思路 A*搜索，用优先队列和一个列表
    # 优先级： F = G（移动代价） + H（预估代价）
    # 维护一些初始化的信息
    parking = problem
    node_start = parking.init_state
    node_end = parking.goal_state
    pqu = PriorityQueue(node_start, parking)
    close = Set()
    open = Set()
    open.add(np.array(node_start.state).tobytes())
    # 开始A*搜索
    while not pqu.empty():
        node = pqu.pop()
        state = node.state
        str_state = np.array(state).tobytes()  # tobytes enables hash
        open.remove(str_state)
        close.add(str_state)
        actions = parking.actions(state)
        for action in actions:
            target, empty_pos = action[0], action[1]
            new_node = node.child_node(parking, empty_pos, target)
            if new_node == node_end:
                return new_node
            if not close.find(np.array(new_node.state).tobytes()) and not open.find(np.array(new_node.state).tobytes()):
                pqu.push(new_node)
                open.add(np.array(new_node.state).tobytes())

    return 0
```

结果如下：

```
请分别使用有信息和无信息搜索方法求解单车整理问题。
有信息搜索-hash
time cost:0.4189789295196533
nodes change is as shown:
[[1, 1, 1, 0], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 0, 1], [2, 1, 1, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 0, 2], [2, 0, 1, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 0, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 2], [2, 0, 2, 0], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [2, 0, 2, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 2, 2, 2], [1, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [0, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 2, 2, 2], [2, 0, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [1, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [2, 1, 1, 0], [0, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [0, 1, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 0, 1, 0], [2, 1, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 0, 2, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 0, 2], [2, 2, 2, 2]]
[[1, 1, 1, 1], [1, 1, 1, 0], [2, 2, 2, 0], [2, 2, 2, 2]]
depth:15

Process finished with exit code 0
```

根据结果可以看出，我们将查找复杂度降低后，成功使BFS在可接受时间范围内得出结果，同时有信息搜索的效率也被大幅度提高。这启发我们，python提供了很多看似便捷的接口，但在实际使用中我们需要仔细考虑其实现代价。