启发式搜索实验报告

武自厚 20336014 保密管理

2022 年 4 月 6 日

本次实验对 A* 算法以及 IDA* 算法进行分析并实现.

1 原理分析

1.1 启发式函数

本次实验采用当前状态和目标状态对应的数字的 Manhattan 距离和作为启发函数, 这是通过对于原问题进行松弛 (即退化为两个数字无论如何都可以交换) 而得来. 即

$$h(n) = \sum_{k=1}^{15} |x_n(k) - x_{\text{dest}}(k)| + |y_n(k) - y_{\text{dest}}(k)|$$

需要注意的是 “0”(即空位) 的距离不应该加入结果, 否则将会存在 $h(n) > h^*(n)$, 也就是不保证找到最优解的情况.

1.2 A* 搜索

在该算法中, 每一个状态 n 都对应着权值 $f(n)$, 其满足

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

其中 $g(n)$ 定为从初始状态到状态 n 所需的步长,$h(n)$ 即是前文提到的启发式函数.

该算法使用优先队列作为 `Open` 表维护, 在算法开始时将初始状态推入优先队列, 另外准备一个集合作为 `Closed` 表. 当 `Open` 表不为空时, 不断完成以下过程:

1. 从 `Open` 表中弹出 n_k, 满足 $f(n_k) = \min_{n \in \texttt{Open}} f(n)$
2. 如果 n_k 就是目标状态, 直接结束算法并返回结果.
3. 遍历 n_k 所有可能的相邻状态 (即行动一次就可以得到的状态)$n_k^{(i)}$. 如果 $n_k^{(i)} \notin$ `Closed`, 那么就计算出 $f(n_k^{(i)})$, 并将 $n_k^{(i)}$ 推入 `Open` 表.

1.3 IDA* 搜索

在该算法中, 依旧存在映射 $f : n \mapsto f(n)$ 表示状态的行动代价, 其意义与 A* 算法中的 f 相同.

该算法使用栈作为 `Open` 表维护, 在算法开始时将初始状态 n_0 压入栈, 将初始成本边界定为 $b \leftarrow f(n_0)$. 当 `Open` 表不为空时, 不断完成以下过程:

- 从 `Open` 表中弹出 n_k. 如果 n_k 就是目标状态, 返回 "已找到" 信息; 如果没有则继续.
- 如果 $f(n_k) > b$, 那么直接返回 $f(n_k)$
- 遍历 n_k 所有可能的相邻状态 (即行动一次就可以得到的状态)$n_k^{(i)}$. 如果 $n_k^{(i)} \notin$ `Closed`, 那么就将 $n_k^{(i)}$ 压入栈, 执行递归算法. 找出 $n_k^{(l)}$ 使得 $f(n_k^{(l)}) = \min_i f(n_k^{(i)})$, 返回 $f(n_k^{(l)})$.

  如果最终没有返回 "已找到" 信息, 则将返回值定为新的 b, 再次执行算法.

2 效果分析

代码文件为父文件夹内 `*.py` 文件, 会在标准输出中打印算法的处理器占用时间以及处理的状态数量.

2.1 标准样例

样例 1(小型):

```
From:
 1  2  4  8
 5  7 11 10
13 15     3
14  6  9 12

To:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

A* algorithm completed with 187 states detected.
It took 0.009882000000000002 secs.
IDA* algorithm completed which 197 states scanned, bound=22
It took 0.008309999999999998 secs.
```

样例 2(小型):

```
❯ python3 main.py
From:
 5  1  3  4
 2  7  8 12
 9  6 11 15
   13 10 14

To:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

A* algorithm completed with 32 states detected.
It took 0.0016639999999999988 secs.
IDA* algorithm completed which 34 states scanned, bound=15
It took 0.0013979999999999965 secs.
```

样例 3(大型):

```
❯ python3 main.py
From:
14 10  6
 4  9  1  8
 2  3  5 11
12 13  7 15

To:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

A* algorithm completed with 832767 states detected.
It took 26.15428 secs.
IDA* algorithm completed which 11114134 states scanned, bound=49
It took 232.89722 secs.
```

样例 4(大型):

```
❯ python3 main.py
From:
 6 10  3 15
14  8  7 11
 5  1     2
13 12  9  4

To:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

A* algorithm completed with 7285026 states detected.
It took 890.048974 secs.
IDA* algorithm completed which 23582429 states scanned, bound=48
It took 565.7612330000001 secs.
```

可见, 在深度不太大的情况下,A* 算法的时间复杂度优于 IDA* 算法, 而当深度很大时则是 IDA* 算法更快. 而 IDA* 算法占用的空间远小于 A* 算法.

2.2 不同启发函数的区别

这里采用两种启发函数进行对比:$h_1(n)$ 为 Manhattan 距离和,$h_2(n)$ 为错位牌的数量.

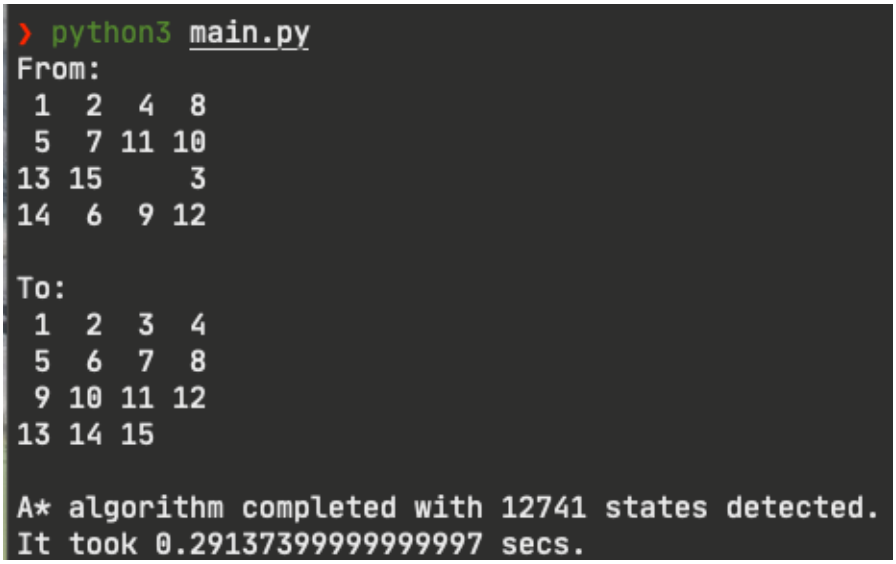
$h_1(n)$:

```
From:
 1  2  4  8
 5  7 11 10
13 15     3
14  6  9 12

To:
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15

A* algorithm completed with 187 states detected.
It took 0.009882000000000002 secs.
IDA* algorithm completed which 197 states scanned, bound=22
It took 0.008309999999999998 secs.
```

$h_2(n)$:



由于 $h_2(n)$ 中对问题的松弛程度比 $h_1(n)$ 大, 导致 $h_2(n) < h_1(n)$, 算法需要更多步骤才能找到目标.

2.3 算法性能差距的原因

假设我们能够找到最优的启发式函数 $h^*(n)$, 那么可以借助 $h^*(n)$ 以最少的步数找到目标状态, 因为采用 $h^*(n)$ 时, 算法只会按照最优路径进行扩展. 然而在现实问题中我们很难直接找到 $h^*(n)$. 只能通过各种情况对 $h^*(n)$ 进行估计得到 $h(n)$. 如果 $h(n) = 0$ 也就是不采用启发式函数, 那么算法将会退化为 UCS 甚至是 BFS 算法, 虽然保证找到最短路径, 但效率很低. 如果 $h(n) > h^*(n)$, 那么启发函数造成的效果将会被高估, 将不会满足最优性.

因此合适的启发式函数应当控制在 $0 < h(n) \leq h^*(n)$ 范围内. 在这个范围内启发式函数都是可采纳的且最优的. 但是如果太小, 启发信息不足, 算法需要扩展更多节点, 造成性能下降; 如果太大, 算法需要花费更多资源去估计启发函数值, 也会造成性能下降. 因此一个合适的启发式函数更能优化算法.