Bait 游戏——搜索算法实现报告

摘　要: 本次实验是一个推箱子游戏（GVGAI-BAIT），目标是实现三种搜索算法，使得 Agent 能够控制小人先吃到钥匙，再进入门，吃到蘑菇有额外加分。我实现了一个算法框架，通过给定三种算法参数（实际实现了四种算法，还有 BFS）和搜索深度来调节算法的行为。三种算法按任务次序分别是不限深度的深度优先搜索、限制深度并使用启发式函数的深度优先搜索（深度参数设置为 43）、限制深度的 A*搜索算法（深度参数设置为 11）。三种算法均能快速通过 0,1,2,4 关，均不能通过第 3 关。下面为详细分析。

关键词: 推箱子;人工智能;深度优先搜索;深度受限搜索;启发式函数;A*

//中图法分类号: TP??? 文献标识码: A

1 Agent 结构

由于设计了一个通用算法框架（下面会讲到），通过调参来指定算法和搜索深度，所以三个 Agent 结构都很简单。下面给出了 A*算法的代码，其他两种参数不同：

```
public class Agent extends AbstractPlayer {
    private final Algorithms algorithms;
    public Agent(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
        algorithms = new Algorithms(stateObs, Algorithms.aStarAlgorithm, Algorithms.MIN_DEPTH);
    }
    @Override
    public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
        return algorithms.act(stateObs, elapsedTimer);
    }
}
```

类 Algorithm 来自包 controllers.Algorithms，构造函数可指定参数如下：

	参数 2：算法	**参数 3：搜索深度**
任务 1	Algorithms.depthFirstAlgorithm	Algorithms.MAX_DEPTH
任务 2	Algorithms.heuristicAlgorithm	Algorithms.MIDDLE_DEPTH
任务 3	Algorithms.aStarAlgorithm	Algorithms.MIN_DEPTH

搜索深度可指定任意正整数，MAX_DEPTH 的值为 Integer. MAX_VALUE，即不限深度，MIN_DEPTH 值为 11，MIDDLE_DEPTH 值为 43。为什么要用这两个数字呢？它们来源于调参，是使用当前算法能够通过除了倒数第二关的其他关卡所需要的最小搜索深度。这四个关卡中有一个关卡需要这么大的搜索深度，而并不意味着其他关卡也需要，比如第一关不需要很深的深度。搜索深度越大，决策速度越慢，而所做决策几乎不会改变（测试范围内未发现，除非深度达到能看到目标，则会直接停止搜索前往目标）。

参数 3 除了表格中的外还可以设置为 breadthFirstAlgorithm，宽度优先搜索，作业中没有要求。

可以发现，我的设计是必须要某一层搜索深度达到指定深度，即一次 act 调用操作的限制时间计算所得的信息可能不足以进行一次决策，而这个游戏是没有后悔药可以吃的，走错一步就不能补救。所以信息不足的情况下算法返回 NIL，并设计为可恢复的，下一次调用继续上次的任务。我称一次决策所需要的信息的搜索深度为“决策深度”，下文也将使用这一名词。

2 Algorithm 分析

2.1 数据结构设计

该搜索问题实际非常复杂，通过前期对框架代码的各种方法进行测试，以及通过阅读 Javadoc，我发现所给数据结构并不能满足我的需求。本实验中我对数据结构设计比较仔细，中心思想是尽可能减少时间开销，空间开销暂不考虑。

在 openSet 和 closedSet 中测试一个元素是否存在，从中取出一个元素的时间复杂度都是 O(n)，虽然是线性的，但存在更快的方式。分析 StateObservation 的 equalPosition 方法，可知 avatar 的位置如果不相同，则一定不相等。更重要的是，可以通过坐标产生全序规则，这就使得状态是可以比较大小的，使用 TreeSet 封装（该数据结构使用红黑树实现）将时间复杂度降低到 O(log n)。

另一个重要的数据结构是 class StateEstimate。考虑 TreeSet 的泛型参数，StateObservation 没有可定制的空间。而许多属性都要聚合在一起方便使用。如 Avatar 的位置在该状态不会变化，Avatar 离目标的曼哈顿距离不会变化，从开始计算路径到该状态的距离不会变化，所以其总和也不会变化。上述三种不会变化的量实际上就是 A*算法评估局面的 h()，g()和 f()。

通过上面的考虑，可以将 h(), g()和 f()的计算函数或方法整合到该数据结构的构造函数中。显然，g()值在初始状态为 0，其他状态均有上一状态，都为其上一状态加 1；h()值虽然是估计值，但直接使用到目标的曼哈顿距离是合理的，另外，h()函数还有一个过滤作用，如 avatar 掉坑里了或 avatar 推箱子和目标重合了，都可认为这样的路径是没有希望的，直接返回 Integer.MAX_VALUE/2，这个值被定义为 INFINITY，算法看到这样的距离直接忽略该节点。f()为两者之和。

考虑游戏的玩法，先要吃钥匙，才能进门。所以将目标设为两段，吃钥匙前的目标为吃钥匙，吃到后转为进门。虽然未考虑吃蘑菇，但蘑菇摆放位置恰好容易被吃到，能通过的关卡中，只有任务 2 的 2 关（0 开始）的蘑菇没有吃到。

class StateEstimate 在包 controllers.Algorithms 中，详情请查看该文件。

2.2 算法设计

算法需要解决的几个问题：通用算法结构、对时间限制“免疫”（计算过程可恢复）、搜索深度可控、减少计算不可能状态、减小数据结构的时间复杂度、重构路径

通用算法结构：使用 AStar 的算法执行框架，将待计算的节点放入优先级队列 openSet，计算过的放入 closedSet，每次从 openSet 取一个出来，分别对该节点的四个方向进行计算。如果子状态不满足一些判断条件则直接被舍弃（到目标无穷远、是走过的节点、是计算过的节点、是不可移动到的节点）。如果没有被舍弃则会判断 openSet 中是否存在相同节点，若存在且该节点的 g()值更低，则舍弃当前节点，否则舍弃 openSet 中的该相同节点（可以证明 openSet 中不会有两个相同节点）。请查看 Algorithm 的 act()方法。

考虑 openSet，是一个优先级队列，这个优先级是自己定的，所以计算节点的方向是自己控制的。对于 DFS 来说，需要一个栈来存储节点，即走得越深越应该得到计算。对于 BFS，需要一层一层计算，即普通的优先级队列。那么将深度越大（在本实现中深度等同于 g()）的优先级设置越高，则优先级队列 openSet 被改造为栈；反之可设置 BFS。对于启发式函数驱动即让 h()的优先级越高就行，A*让 f()的优先级越高。这样就是一个通用的算法结构。严格意义上，由于一些判断条件会使得这个 DFS 和 BFS 搜索的范围更小，算法性能更好，算法表现出来的性质（或所走路径）与重写为通用结构前的纯粹 DFS 相同。下面为自设优先级接口函数：

```
class AStarComparator implements Comparator<StateEstimate> {
    @Override
    public int compare(StateEstimate stateEstimate, StateEstimate t1) {
```

```
        return stateEstimate.distanceTotal - t1.distanceTotal;
    }
}

class HeuristicComparator implements Comparator<StateEstimate> {
    @Override
    public int compare(StateEstimate stateEstimate, StateEstimate t1) {
        return stateEstimate.distanceToGoal - t1.distanceToGoal;
    }
}

class BreadthFirstComparator implements Comparator<StateEstimate> {
    @Override
    public int compare(StateEstimate stateEstimate, StateEstimate t1) {
        int difference = stateEstimate.distanceSoFar - t1.distanceSoFar;
        return difference == 0 ? stateEstimate.sequence - t1.sequence : difference;
    }
}

class DepthFirstComparator implements Comparator<StateEstimate> {
    @Override
    public int compare(StateEstimate stateEstimate, StateEstimate t1) {
        int difference =  t1.distanceSoFar - stateEstimate.distanceSoFar;
        return difference == 0 ? stateEstimate.sequence - t1.sequence : difference;
    }
}
```

优先级接口函数	原理
深度优先搜索	深度之差（右减左）
广度优先搜索	深度之差（左减右）
启发式搜索	距离评估值之差
A*搜索算法	总距离之差

计算过程可恢复和搜索深度可控是相结合的。当搜索时间快到时，就会停止搜索，并判断搜索深度是否满足给定值，如果不满足，则该轮计算是未完成的，用一个 boolean 标识符 shouldContinue 和一个枚举类型状态机 stateMachine 表明下次应该继续搜索。如果深度足够，即满足决策深度，可进行一次决策，并在下次计算前情况所有状态，使得下次计算以当前位置为起始位置进行计算。时间控制的方法模仿 sampleRandom。

前面已经完整实现了整个算法，现在要提升算法性能。不必计算的节点前面已经探讨了。数据结构的设计上，增加的专门用于测试的 log n 复杂度的多个 TreeSet，如 visitedSet, openSetForTesting 等。

最后需要重构路径，这个比较简单，只要在 StateEstimate 中加一个 previousState 域即可。这是一个链表。

3 性能评估

上文已经对算法进行了详细阐述，算法究竟执行性能如何？本节进行实际测试并给出测试结果。

实际在写这样的通用算法框架前，我的任务 1 和任务 2 都是单独写的，由于性能和通关率都太差，且各种情况考虑太过 dirty，控制方法不能调节等。所以被舍弃了。可以查看源文件相关 controller，那些代码没有删去都被注释了。

从运行 log 中获取的信息如下：

深度优先搜索：搜索深度：2147483647，执行时间限制：10000

	输赢	分数	时间步
第 0 关	Win	5	9
第 1 关	Win	7	98
第 2 关	Win	11	261
第 3 关	Lose	0	1000
第 4 关	Win	11	61
总计	4:1	34	429

深度受限启发式搜索：搜索深度：43，执行时间限制：100

	输赢	分数	时间步
第 0 关	Win	5	9
第 1 关	Win	7	205
第 2 关	Win	7	124
第 3 关	Lose	0	1000
第 4 关	Win	11	549
总计	4:1	30	887

A*搜索：搜索深度：11，执行时间限制：100

	输赢	分数	时间步
第 0 关	Win	5	9
第 1 关	Win	7	38
第 2 关	Win	11	155
第 3 关	Lose	2	1000
第 4 关	Win	11	72
总计	4:1	**36**	**274**

可见，A*搜索算法在时间步上明显优于前面两种算法。不过这只是粗略估计，因为每种搜索都使用满足要求的最小深度来搜的，深度受限搜索的最小搜索深度为 43，会使有些关卡浪费了一些时间步返回 nil。而深度优先搜索没有限制执行时间，也就是说没有返回过 nil。

可以发现有的情况下 A*是不如深度受限的启发式搜索的，如第 2 关，由于局面比较对称，启发式搜索更专注于眼下较近范围进行搜索，而 A*更倾向于宽度的，就会出现反复跑来跑去的现象。而纯粹的深度优先搜索耗时较少，但非最优路径，跑了很多弯路。对于第四关，墙壁限制少，造成宽度很广，而深度并不深，深度优先就捡了便宜很快搜到，而启发式函数搜索了大量的宽度范围，搜索深度 43 也是因为这个关卡而定的，所以性能很差，A*搜索，时间步应该比深度优先少，但是由于时间限制可能返回了 nil 使得时间步稍微虚高。

第三关为什么会失败？失败在算法中定义为 openSet 为空。显然，如果能够穷尽所有状态，最终总能到达目的，只是时间问题。而算法没有穷尽所有状态，这与框架有关。有一些状态被认为冗杂的、不具有希望的被淘汰，如走过的路。所以算法不允许回到一个到达过的状态。所以算法尽其所能之后放弃搜索。

References

[1] A* search algorithm - Wikipedia https://en.wikipedia.org/wiki/A*_search_algorithm

[2] Information Technology Gems: Java Collections – Performance (Time Complexity) http://infotechgems.blogspot.com/2011/11/java-collections-performance-time.html

[3] Disha Sharma, Sanjay Kumar Dubey. Anytime A* Algorithm – An Extension to A* Algorithm. International Journal of Scientific & Engineering Research Volume 4, Issue 1, January-2013

附录 A：屏幕录制

请查看ScreenRecord目录。

附录 B：运行 log

请查看ouput.txt文件

附录 C：Algorithms.java 源文件

请查看gvgai-assignment1/src/controllers/Algorithms.java文件

附录 D：运行方式

```
cd gvgai-assignment1/
java -cp out/production/gvgai-assignment1/ Assignment1
```

感谢阅读！