基于 TT-ENTAILS 的实现说明

针对待完善代码 check_true_false 函数，本部分给出了一种基于 TT-Entails 算法思路的实现说明。基于 TT-Entails 算法代码详见 tt_entails.py 文件。

一、算法分析

对 TT-ENTAILS.pdf 中提出的 TT-Entails 算法进行分析可知，该算法可以拆解成为以下几个部分：

- TT-Entails? ：它提供了初始化和 TT-Check-All 的入口；
- TT-Check-All：为一个递归程序，它修改 model，并不断地调用自身。从它的行为来看，它按照列表的顺序对每个不确定真值的变量进行了枚举，并在枚举出一个 model 之后调用 PL-True? 将这些值代入到逻辑表达式中进行计算；
- ExtractSymbols：从逻辑表达式中抽取出逻辑变量；
- PL-True? ：把一组逻辑值代入到逻辑表达式中，获得一个布尔值。

这里我们主要讨论 TT-Entails? 、TT-Check-All 和 PL-True? 的实现，即初始化及算法入口、枚举遍历和表达式求值。

二、代码分析

本部分内容，将分析的代码主要为 utils.py 中的 Statement 类及其子类。

通过简单的代码阅读，我们不难发现，Statement 类及其子类承担着构建逻辑表达式的作用。整个逻辑表达式在构建时为一个树形结构（见 TT-Entails.pdf 第 19 页），叶子节点对应于逻辑变量，各类运算符对应于树的非叶子节点。

对每个表达式进行求值时，首先对子表达式进行求值，在获取子表达式的逻辑值后，再根据运算符进行运算。

回到代码，我们观察到 Statement 类预留了一个 eval 方法。该方法的参数为 atom_dict，返回一个布尔值。特别地，当表达式为叶子结点（Atom 类）时，返回值为该叶子结点在 atom_dict 中对应的值。对比其它类型的表达式，可以得出，该方法对应于 PL-True? ：将 atom_dict 视为 model，eval 方法实现了表达式求值。

```python
class Statement(object):
    def __init__(self) -> None:
        self.atom_set = set()
        self.operator = None
        self.statements = []

    def eval(self, atom_dict: Dict[str, bool]):
        raise NotImplementedError
```

```python
class Atom(Statement):
    def __init__(self, name: str) -> None:
        self.operator = 'ATOM'
        self.name = name
        self.atom_set = {name.strip()}

    def eval(self, atom_dict: Dict[str, bool]):
        v = atom_dict[self.name]
        if v is None:
            return None
        return atom_dict[self.name]
```

在给出的代码中，仅有 Atom、Or、And 和 Xor 的 eval 方法给出了具体的实现，其余的运算符需要你自己编写。

三、PL-True? 的实现

在代码分析中，我们利用方法 eval 对 PL-True? 给出了一个实现。该方法将运算符视为算子，在对表达式进行求值的时候，我们可以直接调用 eval 方法，传入一个 model，得到表达式在该 model 下的值。

进一步地观察，可以发现，PL-True? 事实上对整个表达式树进行了一次遍历，对 eval 进行调用则隐式地实现了这样的一次遍历，并在遍历的过程中，自底向上地传递了子表达式的值。

另一方面，我们可以把这种遍历显式地实现，给出 PL-True? 的另一种实现。由于我们需要对树形结构进行遍历，递归函数就是我们天然的选择。在这个实现中，PL-True? 首先递归地对参数中的表达式之子表达式进行计算，然后根据当前表达式的运算符，将其子表达式的值代入，得出计算结果。特别地，若当前表达式不存在运算符（表达式的类型为 Atom 或表达式的运算符为 ATOM），当前表达式为一逻辑变量，则直接从 PL-True? 的参数 model 中取出该逻辑变量的值。伪代码如下：

```
def PL-True?(expression: Statement, model: dict) -> bool:
    if is-atom(expression):
        return model[expression.name]
    sub-values = []
    for sub-expr in expression.statements:
        sub-values.append(PL-True?(sub-expr, model))
    return apply-operator(expression.operator, sub-values)
```

伪代码中的 is-atom 函数即为一个测试当前表达式是否是简单逻辑变量的函数，apply-operator 则使用将当前表达式对应的运算符作用在子表达式求出的值上。考虑到一些逻辑运算存在短路的特性，例如：AND 运算在我们知道其子表达式中存在一个是 False 的时候，就不用去计算其它的表达式，直接返回 False。利用这种性质，我们能够对 PL-True? 进一步地优化，请同学们自行探索。

四、TT-Entails? 的实现

观察 TT-Entails? 的伪代码（见 TT-Entails.pdf 第 15 页），我们了解到 TT-Entails 抽取了 KB 和 alpha 中的符号，并提供了一个 TT-Check-All 的入口。

对于抽取 KB 和 alpha 中的符号 ExtractSymbols，Statement 类事实上已经提供了相关的属性 atom_set。atom_set 中给出了当前表达式中的所有逻辑变量的名称的集合，因此直接将 KB 的 atom_set 和 alpha 的 atom_set 取并集就完成了这一部分的实现。

作为 TT-Check-All 的入口，我们需要四个参数：KB、alpha、symbols 和 model。其中 KB、alpha 已经给出，对于 symbols 和 model 的选择，一种方法为直接根据伪代码，将 symbols 设为全符号集、alpha 为空的 dict。在这种情况下，对于给定的题目，我们将要枚举 45 个变量的真值表，共 2^{45} 种情况，这种枚举是不可忍受的，因此我们有必要根据

题目的一些良好的性质做一些优化。

根据题目的描述，我们的 KB 包含两部分：Wumpus World 的规则和单变量逻辑表达式。这些单变量逻辑表达式或为简单的逻辑变量，或为简单逻辑变量的否定。对于那些使得 KB 为 True 的 World，这些单变量逻辑表达式的取值必然为固定的值，并由其是否为简单逻辑变量给出。有了这个约束，我们可以首先对单变量逻辑表达式进行处理，得到一个部分取值的 model，并在 symbols 中去掉我们已经赋值了的逻辑变量。

由于 KB-b 中的表达式或为简单逻辑变量，或为简单逻辑变量之否定，代码中的 expr.statements[0].name 即提取了简单逻辑变量之否定中的简单逻辑变量。由题目的信息，经过这样的处理后，我们需要确定的真值表的行数下降到了 2^{20} 左右，这保证了直接枚举来进行推理的时间复杂度能够落在合理的范围内。

五、TT-Check-All 的实现

TT-Check-All 为整个推理程序的核心，它负责对逻辑变量的真值表进行枚举，从而进行推理。观察 TT-Check-All 的伪代码（见 TT-ENTAILS.pdf 第 19 页），TT-Check-All 为一个递归程序，它的终止条件为 symbols 空，即对所有符号的逻辑值都进行了枚举。在 symbols 为非空时，它将取出 symbols 的第一个元素，并尝试将它复制为真和假这两种情况。

注意到在 TT-Check-All 的相邻的两次真值表枚举中，在大多数的情况下，多数的逻辑变量的取值没有发生改变，此处我们给出另一个优化 PL-True? 的思路：对表达式构成的树上，我们保留每个节点在上一次求值中取得的值，每次求值前于上一次求值的 model 进行比较，找出取值不同的变量，并在求值的过程中只计算那些包含（或部分包含）了这些变量的表达式，从而达到节省计算量的目的。特别地，当我们对逻辑变量的取值的枚举按照二进制格雷码的顺序进行时，相邻两次求值的 model 的仅有一个变量存在差别。同时，我们应当注意到，若存在短路计算的情况，某些子表达式的历史取值可能并不是上一次求值得到的，此处留作思考。

另一方面，考虑到 Wumpus World 对某些逻辑变量的约束非常强，例如 wumpus 只存在一个、左下角不可能存在 wumpus 或 pit。我们可以通过调整 symbols 列表中的变量的顺序，在 model 还是部分确定的情况下，提前测试 KB 是否为真，从而减少枚举次数。具体实现留作思考。