中山大学数据科学与计算机学院

程序设计数据结构综合实践

Boggle 实验报告

(2018-2019 学年秋季学期)

学　　号: 16337113

姓　　名: 劳马东

专　　业: 超算

一. Boggle 游戏

Boggle 是一个单词游戏，它由一个 4*4 的棋盘组成，棋盘的每个位置是一个骰子，骰子的 6 面是不同的字母。初始时，16 个骰子随机初始化为一个面的字母，玩家通过点击不同的骰子切换骰子的字母，找出合法的单词，从而得到一定分数。

1、 单词规则

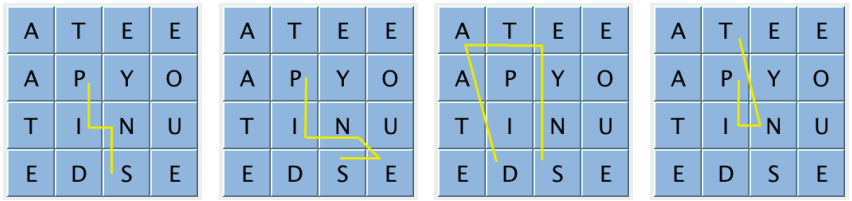
(a) 单词由一系列相邻骰子的字母按顺序连成（考虑行、列、对角线）；

(b) 单词至多使用每个骰子一次；

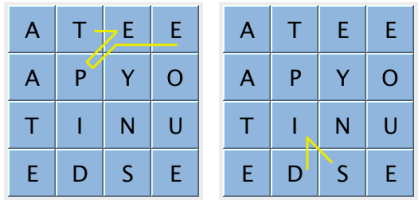
(c) 单词需要包含至少 3 个字母；

(d) 单词需要在给定字典中出现；

图1给出了一些合法和不合法的例子。



(a) pins：合法 (b) pines：合法 (c) dates：D 和 A 不相邻 (d) pint：N 和 T 不相邻



(e) tepee：E 被使用了两次 (f) sid：单词不在字典中

图 1. boggle 例子

2、 积分规则

不同长度的单词分数不同，如表1。在计算单词长度时，字母 Q 算 2，因为在英语中 Q 后面总是跟着 u，在骰子中省略了 Q 后面的 u。例如 Qeen 实际上是 Queen，它的长度时 5。

单词长度	分数
0-2	0
3-4	1
5	2
6	3
7	5
8+	11

表 1. 单词分数

二. 实验内容

（一）问题分析

对于一个给定的棋盘，最简单的方法是从 16 个不同的位置开始，运行一遍 DFS，每次递归都判断访问过的序列是否出现在给定字典中。这个暴力方法的缺点是时间复杂度太高。假设字典中单词个数为 N，棋盘大小为 $m \times n$，每个位置邻居节点平均个数为 b，则总的节点数为 $1 + b + b^2 + ... + b^{m \times n - 1}$，每次递归检索单词的复杂度上界为 $O(N)$，故总的时间复杂度为 $O(b^{m \times n - 1} N)$。这种指数级的时间复杂度显然不是我们想要的，更何况棋盘种类有 $6^{m \times n}$ 种。

（二）初步优化

考虑在字典中查单词 apple 的过程，肯定不会从头到尾遍历每一个单词。普遍的做法是，我们先翻到字母 A 出现的第一页，然后在所有以字母 A 开头的部分，翻到 P 开头的第一页，不断重复，直到找到单词 apple。这就引出了实验中所用到的第一种数据结构——Trie 字典树。

Trie 树是一棵多叉树，树的节点值是一个前缀（这个前缀也可能是一个单词），边是单个字母，表示子树的开始字母。图2是字典 A、i、to、in、tea、ted、ten、inn 对应的 Trie 树，它具有以下特点：

1、 根节点不包含字符，除根节点外的每一个节点都包含一个字符。

2、 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。叶子节点必定对应字典中的一个单词，而内部节点可能对应字典的一个单词，如单词 ad 和 ba 都是在内部节点。

3、 每个节点的所有子节点包含的字符互不相同。

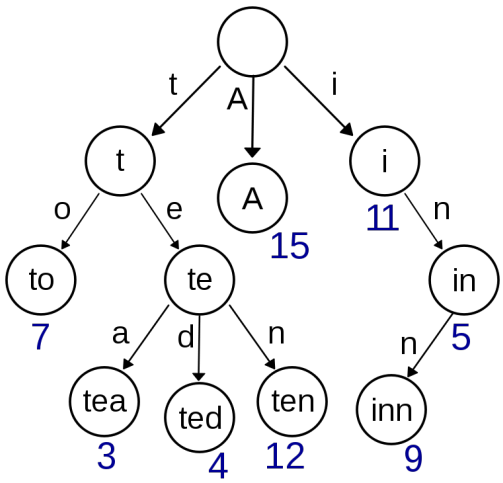


图 2. Trie 树的例子

显然，Trie 树的查找效率取决于最长单词的长度。假设 Trie 树的平均子节点树为 c，单词最大长度为 h，则其查找时间复杂度为 $O(c) \times O(h) = O(ch)$。这样，我们就把 DFS 的时间复杂度从 $O(b^{m \times n - 1} N)$ 降低到了 $O(b^{m \times n - 1} ch)$，显然通常情况下 $N \gg ch$。但是，这似乎并没有多大的优化效果？因为时间瓶颈在于 DFS 的状态空间太大了，有 $b^{m \times n - 1}$ 种。

其实，同样利用 Trie 树，我们可以对 DFS 剪枝。在 DFS 遍历的过程中，如果发现经过的字母序列不是 Trie 树的一个前缀，就可以回溯，因为继续往下搜索到的单词肯定不在单词中。

算法1是使用 Trie 树优化后的 DFS 搜索伪代码。与普通的 DFS 伪代码相比有两处不同。一是判断当前 $prefix$ 是不是 Trie 树的一个前缀，以进行剪枝；二是用 Trie 树检索 $prefix$ 是不是字典的一个单词，加快查找效率。

算法 1 Trie 树优化的 DFS

输入: $node$ 当前节点状态，is_max 是否极大节点

输出: 博弈树的值

function DFS-TRIE($board, cur\_pos, trie, prefix, visited, words$)
    add cur_pos into $visted$
    $prefix \leftarrow prefix + board[cur\_pos]$
    if $prefix$ is a prefix in $trie$ **then**
        if $prefix$ is a word in $trie$ **then**
            add $prefix$ into $words$
        end if
        for $n\_pos \in$ neighbor(cur_pos) **do**
            if n_pos is not in $visted$ **then**
                dfs-trie($n\_pos, prefix, trie, words$)
            end if
        end for
    end if
end function

三. 关键代码

1、 Trie 节点类

```
class TrieNode {
    // char表示边，bool表示子节点是不是对应一个单词
    map<char, pair<bool, TrieNode*>> _children;
public:
    // 获取对应边上的子节点
    pair<bool, TrieNode*> get_child(char c);
    // 获取所有子节点的字符
    string keys() const;
    // 获取所有子节点
    vector<TrieNode*> children() const;
    // 插入一个字符串序列
    bool insert(string::const_iterator first, string::const_iterator last);
};
```

代码清单 1. 使用 Trie 树的 DFS 搜索

2、 使用 Trie 树的 DFS 搜索

代码的基本框架与算法1相同。这里做了实现上的一些优化。首先，不用每次都从根节点开始往下搜索，以判断是不是前缀或者单词。TrieNode 指针也随着 DFS 往下递归时，向下指到子节点中；其次，TrieNode 节点有一个 bool 变量以表示它是否对应一个单词，而不是每次从头开始遍历。

```
void BoggleSolver::dfs(const BoggleBoard& board, int i, int j,
                       string prefix, TrieNode* node,
                       set<string> &all_words,
                       vector<vector<int>>& visited) {
    visited[i][j] = 1;
    char letter = board.getLetter(i, j);
    prefix += letter;

    bool is_word;
    TrieNode* child;
    // is_word表示child节点是否对应一个单词
    tie(is_word, child) = node->get_child(letter);
    if (is_word)
        all_words.insert(prefix);

    // get_child的结果为NULL指针
    // 说明当前prefix不是Trie树的前缀
    if (child) {
        for (int k = -1; k < 2; ++k) {
            for (int l = -1; l < 2; ++l) {
                if (k == 0 && l == 0)
                    continue;
                int ii = i + k, jj = j + l;
                if (isvalid(ii, jj) && visited[ii][jj] == 0)
                    // 这里传递child指针
                    // 直接在child指针判断当前prefix是不是Trie树的前缀
                    // 而不用从根节点开始
                    dfs(board, ii, jj, prefix, child, all_words, visited);
            }
        }
    }
    visited[i][j] = 0;
}
```

代码清单 2. 使用 Trie 树的 DFS 搜索

3、 Trie 树的插入

由于 Trie 树是递归定义的，因此 Trie 树的插入算法也可以用递归。把 $[first+1, last)$ 区间的字符串传递给 *child* 节点，让它建立一棵 Trie 子树，*first* 处的字符是到这棵子树的边。*insert* 函数的返回值表示它是否对应一个单词，TRUE 表示不对应，FALSE 表示对应。显然，叶子节点对应一个单词。一个内部节点在插入它对应的单词时，它也是一个叶子节点，是因为后来插入了以它为前缀的单词，才使它变成了内部节点。因此，*is_word* 使用或赋值判断。

```
bool TrieNode::insert(string::const_iterator first,
                      string::const_iterator last) {
    if (first == last) {
        return false;
    }
    else {
        bool is_word;
        TrieNode *child;
        tie(is_word, child) = _children[*first];
        if (child == nullptr)
            child = new TrieNode;
        bool flag = child->insert(first + 1, last);
        // 要么之前是叶节点，要么现在是叶节点
        is_word = is_word || !flag;
        if (!flag) {
            delete child;
            child = nullptr;
        }
        _children[*first] = make_pair(is_word, child);
        return true;
    }
}
```

代码清单 3. 使用 Trie 树的 DFS 搜索

四. 实验结果

由于完全搜索全部可能的棋盘效率极低，实验中使用随机生成棋盘的方法，结果表明该方案确实能获得很好的效率提升。在小数据集上，寻找一个得分为 19 的棋盘 YVES EDOO AXCE LLNO，用时 1.279 秒；在大数据集上，寻找一个得分为 20 的棋盘 RRNE GYOC ALAE VRIJ，用时 0.967 秒。

字典	分数	时间/秒
dictionary-algs4	19	1.279
dictionary-yawl	20	0.967

表 2. 问题求解时间