数据结构与算法 II 上机实验 (11.3)

中国人民大学　信息学院　崔冠宇　2018202147

注：请使用 C++17 或以上编译器！

上机题 1　实现最长公共子序列 (LCS) 算法。

一、问题描述

实现最长公共子序列算法。

1. 输入：字符串 $X =< x_1, x_2, \cdots, x_m >$ 与字符串 $Y =< y_1, y_2, \cdots, y_n >$。
2. 输出：它们的最长公共子序列 $Z =< z_1, z_2, \cdots, z_l >$。

二、算法基本思路

记 $count[i][j]$ 为 $X[1...i]$ 与 $Y[1...j]$ 的最长公共子序列的长度。根据课上的分析（定理 15-1），总共有下列几种情况：

1. 若两字符串的当前字符相同，即 $X[i] = Y[j]$，则两字符串截至此处的最长公共子序列为"两字符串往前缩一个字符的最长公共子序列加上当前字符"，即

$$count[i][j] = count[i-1][j-1] + 1$$

2. 若两字符串的当前字符不同，即 $X[i] \neq Y[j]$，则两字符串截至此处的最长公共子序列为"X 往前缩一个字符，Y 不动的最长公共子序列"和"Y 往前缩一个字符，X 不动的最长公共子序列"中较大的一个，即

$$count[i][j] = \max\{count[i-1][j], count[i][j-1]\}$$

为了重构解，还需要定义 $dir[i][j]$ 记录每一次计算 $count[i][j]$ 是上述何种情况：

1. 如果是上述第一种情况，则 $dir[i][j] = LEFTUP$；
2. 如果是上述第二种情况，且"X 往前缩一个字符，Y 不动的最长公共子序列"较长，则 $dir[i][j] = UP$；
3. 如果是上述第二种情况，且"Y 往前缩一个字符，X 不动的最长公共子序列"较长，则 $dir[i][j] = LEFT$。

打印解的时候，首先根据 $dir[i][j]$ 的方向递归打印两字符串前缀对应的解，若 $dir[i][j] = LEFTUP$，则还要打印 $X[i](Y[j])$。

综上，得到状态转移方程

$$count[i][j] = \begin{cases} count[i-1][j-1] + 1, & X[i] = Y[j] \\ \max\{count[i-1][j], count[i][j-1]\}, & X[i] \neq Y[j] \end{cases}$$

$$dir[i][j] = \begin{cases} LEFTUP, & X[i] = Y[j] \\ UP, & X[i] \neq Y[j] \text{且} count[i-1][j] \geq count[i][j-1] \\ LEFT, & X[i] \neq Y[j] \text{且} count[i-1][j] < count[i][j-1] \end{cases}$$

下面给出 LCS 的伪代码:

```
// 最长公共子序列
// 参数:
//      X, Y: 字符串
// 返回值:
//      count[i][j]: X[1...i] 与 Y[1...j] 的最长公共子序列的长度
//      dir[i][j]: 上述最长公共子序列由哪个前缀确定
LCS(X, Y):
    m = X.length()
    n = Y.length()
    count[0...m, 0...n], dir[1...m, 1...n]
    // 初始化
    for i = 0 to m:
        count[i, 0] = 0
    for i = 0 to n:
        count[0, i] = 0
    // 填表
    for i = 1 to m:
        for j = 1 to n:
            // 情况一
            if X[i] == Y[j]:
                count[i, j] = count[i - 1, j - 1] + 1
                dir[i, j] = LEFTUP
            // 情况二
            else if count[i - 1, j] >= count[i, j - 1]:
                count[i, j] = count[i - 1, j]
                dir[i, j] = UP
            else:
                count[i, j] = count[i, j - 1]
                dir[i, j] = LEFT
    return count and dir
```

以及重构解的伪代码:

```
// 打印解
// 参数:
//       X: 字符串
//       dir: LCS 使用的记录方向的矩阵
//       i, j: 想求的是 X[1...i], Y[1...j] 的 LCS
// 输出:
//       X[1...i], Y[1...j] 的 LCS
PrintSolution(X, dir, i, j):
    // 递归出口
    if i == 0 || j == 0:
        return
    // 情况一
    if dir[i, j] == LEFTUP:
        PrintSolution(X, dir, i - 1, j - 1)
        print(X[i])
    // 情况二
    else if dir[i, j] == UP:
        PrintSolution(X, dir, i - 1, j)
    else:
        PrintSolution(X, dir, i, j - 1)
```

三、算法复杂性分析

先分析 LCS 的复杂度:

1. 时间复杂度:
   设 $|X| = m$，$|Y| = n$，算法运行时间为 $T(m, n)$。根据算法伪代码，主要的计算步骤在于双层循环，共循环 mn 次，每次循环内部是常数时间，所以 $T(m, n) = O(mn)$。
2. 空间复杂度:
   算法内部的辅助空间是 $count[0...m, 0...n]$ 和 $dir[1...m, 1...n]$，共有 $(m+1)(n+1) + mn = O(mn)$ 个单元，所以 $S(m, n) = O(mn)$。

再分析 PrintSolution 的时间复杂度:

容易看出，在打印解的时候，程序递归走过的路径至多是从 dir 右下角走到左上角的曼哈顿距离 $m + n$，所以 $T(m, n) = O(m + n)$。

四、程序源代码

最长公共子序列: lcs.cpp:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <utility>

// 最长公共子序列问题
// （动态规划法）

// 方向枚举，重构解的时候使用
enum class Direction
{
    // 空方向
    NULLDIR,
    // 左上
    LEFTUP,
    // 左
    LEFT,
    // 上
    UP
};

// 最长公共子序列
// 参数:
//      X[]: 字符串1
//      Y[]: 字符串2
// 返回值:
//      count[i][j]: 保存 X[1...i] Y[1...j] 的最长公共子序列长度
//      dir[i][j]: 保存 X[1...i] Y[1...j] 的最长公共子序列由哪个前缀确定
// 时间复杂度:
//      O(mn)
// 空间复杂度:
//      O(mn)
//
auto LCS(const std::string & X, const std::string & Y)
{
    // 长度
```

```
    size_t m = X.length();
    size_t n = Y.length();

    // 辅助空间: O(mn)
    // count[0...m][0...n]
    // 用于记录最长公共子序列长度
    std::vector<std::vector<size_t>> count;
    // dir[0...m][0...n] (0 忽略)
    // 用于重构解
    std::vector<std::vector<Direction>> dir;

    // 初始化为 (m + 1) * (n + 1) 的全零矩阵
    // 零向量 1 * (n + 1)
    std::vector<size_t> zeroSizeVector;
    std::vector<Direction> zeroDirVector;
    // 构建零向量
    for(size_t i = 0; i < n + 1; i++)
    {
        zeroSizeVector.push_back(0);
        zeroDirVector.push_back(Direction::NULLDIR);
    }
    // 构建零矩阵
    for(size_t i = 0; i < m + 1; i++)
    {
        count.push_back(zeroSizeVector);
        dir.push_back(zeroDirVector);
    }
    // 填表
    // 时间复杂度: O(mn)
    for(size_t i = 1; i <= m; i++)
    {
        for(size_t j = 1; j <= n; j++)
        {
            // 最后字符相同
            if(X[i - 1] == Y[j - 1])
            {
                count[i][j] = count[i - 1][j - 1] + 1;
```

```
                dir[i][j] = Direction::LEFTUP;
            }
            // max{ LCS_len(X_{i-1}, Y_j), LCS_len(X_{i}, Y_{j-1}) }
            else if(count[i - 1][j] >= count[i][j - 1])
            {
                count[i][j] = count[i - 1][j];
                dir[i][j] = Direction::UP;
            }
            else
            {
                count[i][j] = count[i][j - 1];
                dir[i][j] = Direction::LEFT;
            }
        }
    }
    return std::make_pair(count, dir);
}

// 打印最长公共子序列
// 参数:
//       X[]: 字符串1
//       dir[][]: 表示最长公共子序列由谁决定的矩阵
// 输出:
//       重构出的最长公共子序列
// 时间复杂度:
//       O(m+n)
void PrintSolution(const std::string & X,
    const std::vector<std::vector<Direction>> & dir,
    size_t i, size_t j)
{
    // 打印解的时候，走过的路径至多就是
    // 从右下角到左下角的曼哈顿距离 O(m+n)
    if(i == 0 || j == 0)
        return;
    // 左上方，递归打印前边的最长公共子序列
    // 并打印该字符
    if(dir[i][j] == Direction::LEFTUP)
```

```cpp
    {
        PrintSolution(X, dir, i - 1, j - 1);
        std::cout << X[i - 1];
    }
    // 上边或左边，说明最后字符不同，需要递归打印
    else if (dir[i][j] == Direction::UP)
    {
        PrintSolution(X, dir, i - 1, j);
    }
    else if(dir[i][j] == Direction::LEFT)
    {
        PrintSolution(X, dir, i, j - 1);
    }
    else
    {
        std::cout << "Error";
    }
}

int main()
{
    // 测试字符串
    std::string X = "ABCBDAB";
    std::string Y = "BDCABA";
    std::cout << "LCS of " << X << " and "
        << Y << " is: ";
    // 求解矩阵
    auto result = LCS(X, Y);
    // 打印解
    PrintSolution(X, result.second,
            X.length(), Y.length());
    std::cout << std::endl;
    std::cout << "Length: "
        << result.first[X.length()][Y.length()] << std::endl;
    return 0;
}
```

五、运行结果截图

运行 lcs.cpp，程序以课件上的例子做测试

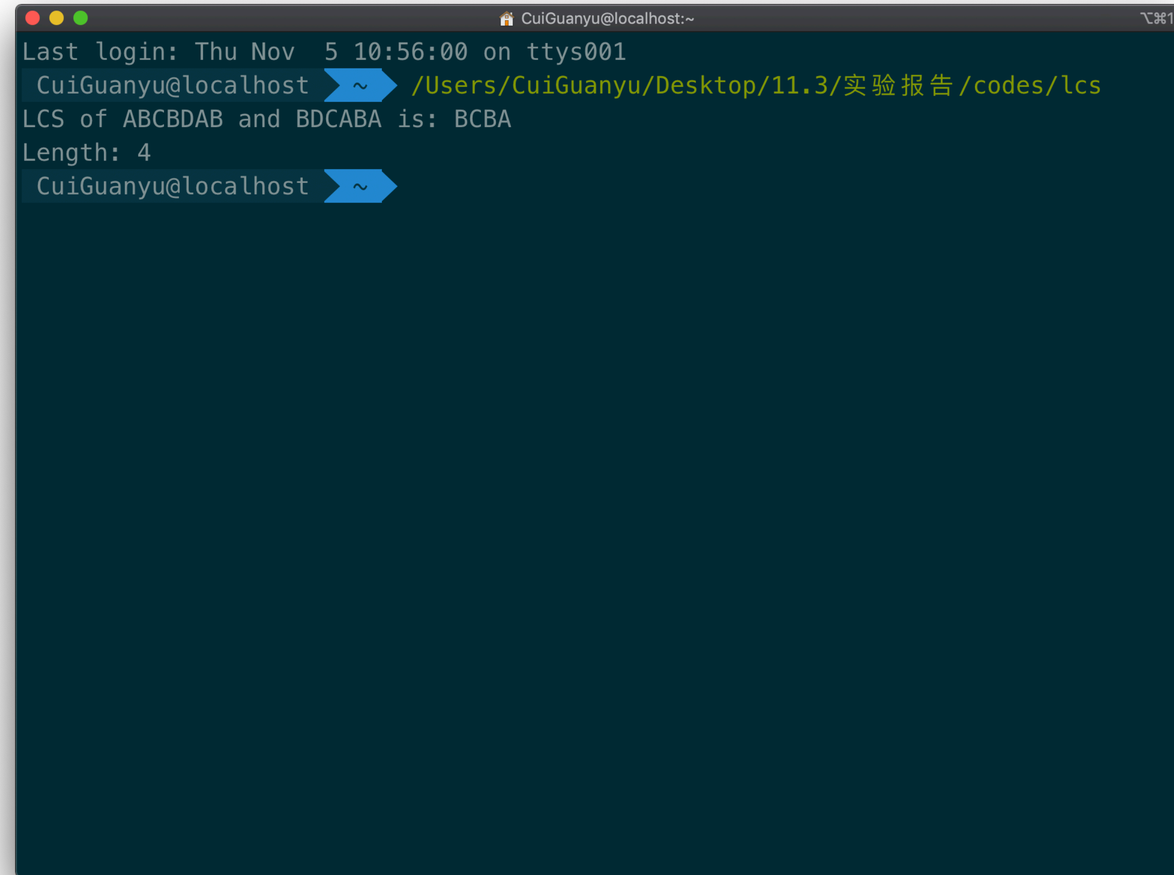

图 1: 最长公共子序列测试

可见程序运行正确。

上机题 2 实现最优二叉搜索树 (OBST) 算法。

一、问题描述

实现最优二叉搜索树算法。

1. 输入：排好序的关键词序列 $K=<k_1,k_2,\cdots,k_n>$，各关键词被查询到的概率数组（可按比例扩大为整数）$P=<p_1,p_2,\cdots,p_n>$，各关键词“之间”[1]被查询的概率数组（可按比例扩大为整数）$Q=<q_0,q_1,q_2,\cdots,q_n>$。
2. 输出：一棵最优二叉搜索树（按先根序打印）。

二、算法基本思路

使用动态规划的方法，多阶段决策当前子树的根节点。

首先，定义 $w[i,j](i=1...n,j=i-1...n)$[2] 为增加一层时，由 $K[i...j]$ 构成的子树增加的代价，根据二叉搜索树外节点比内节点多一个的特点，有

$$w[i,j]=Q[i-1]+\sum_{l=i}^{j}(P[l]+Q[l])$$

接下来定义 $e[i,j]$ 为 $K[i...j]$ 构成最优二叉搜索树时的最小代价。假若 $K[r]$ 为 $K[i...j]$ 的根时，

$$\begin{aligned} e[i,j] &= P[r]+(e[i,r-1]+w[i,r-1])+(e[r+1,j]+w[r+1,j]) \\ &= e[i,r-1]+e[r+1,j]+w[i,j] \end{aligned}$$

因为我们要使树代价 $e[i,j]$ 最小，所以有

$$e[i,j]=\begin{cases} Q[i-1], & j=i-1 \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\}, & i\le j \end{cases}$$

为了重构解，我们需要保存 $K[i...j]$ 的根的位置为 $root[i,j]$，即：

$$root[i,j]=\arg\min_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w[i,j]\}$$

重构解时，只需要打印出 $root[i,j]$ 对应的关键字，并分别打印左子树和右子树即可。

下面给出 OBST 算法的伪代码：

```
// 最优二叉搜索树
// 参数:
```

[1]“伪关键字”，即 q_i 为查询词落到 (w_i,w_{i+1}) 的概率。

[2]当 $j=i-1$ 时，表示子树为空，也即只有外节点。

```
//      P[1...n]: 各关键词被查询的概率数组
//      Q[0...n]: 各伪关键词被查询的概率数组
//      n: 关键词数量
// 返回值:
//      e[i][j]: 表示关键词i-关键词j的OBST的搜索代价
//      root[i][j]: 表示关键词i-关键词j的OBST的根
OBST(P, Q, n):
    e[1...n + 1, 0...n], w[1...n + 1, 0...n], root[1...n, 1...n]
    // 初始化
    for i = 1 to n:
        e[i][i - 1] = Q[i - 1]
        w[i][i - 1] = Q[i - 1]
    // l 长的数组, 类似矩阵链一样斜着填表
    // n 次
    for l = 1 to n:
        // n - l + 1 次
        for i = 1 to n - l + 1:
            // 终点
            j = i + l - 1
            e[i, j] = +INFINITY
            w[i, j] = w[i, j - 1] + P[j] + Q[j]
            // 计算根
            for r = i to j:
                t = e[i, r - 1] + e[r + 1, j] + w[i, j]
                if t < e[i, j]:
                    e[i, j] = t;
                    root[i, j] = r;
    return e and root
```

再给出 PrintSolution 的伪代码:

```
// 打印解
// 参数:
//      words: 关键词数组
//      root: OBST 算出的根的位置
//      i, j: 关键词 i-j 构成的树
// 输出:
//      最优二叉树, 以先根序给出
```

```
PrintSolution(words, root, i, j):
    // i > j 表示空节点
    if i > j:
        print("NULL")
    // 打印根
    print(root[i, j])
    // 左子树
    PrintSolution(words, root, i, root[i, j] - 1)
    // 右子树
    PrintSolution(words, root, root[i, j] + 1, j)
```

三、算法复杂性分析

先分析 OBST 的复杂度:

1. 时间复杂度:
   类似于矩阵链乘法，OBST 也是在两个循环中填写一个 $O(n) \times O(n)$ 的矩阵，其中每个单元确定树根时都要从 i 迭代至 j，共有 $O(n)$ 次计算，所以平凡方法的总时间复杂度为 $O(n^3)$[3]。
2. 空间复杂度:
   内部辅助空间 e、w 与 $root$ 都是 $O(n) \times O(n)$ 的矩阵，所以空间复杂度 $S(n) = O(n^2)$。

再分析 PrintSolution 的时间复杂度:

设打印节点为基本操作。因为二叉搜索树有 n 个内节点和 $n+1$ 个外节点，所以共会打印 $O(n)$ 次，即时间复杂度为 $T(n) = O(n)$。

四、程序源代码

最优二叉搜索树: OBST.cpp:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include <limits>

// 最优二叉搜索树
// 参数:
//       P[]: 各关键字被检索的概率
```

[3]根据课本 15.5-4，Knuth 改进了算法，使得时间复杂度降低到了 $\Theta(n^2)$。

```
//      Q[]: 落在各关键字“之间”的概率
//      n: 关键字个数
// 返回值:
//      e[i][j]: a_i...a_j 构成的最优二叉树的COST
//      root[i][j]: a_i...a_j 构成的最优二叉树的根的位置
// 时间复杂度: (认为填表是基本操作)
//      T(n)=O(n^3), 若确定根时使用平凡方法
//      T(n)=O(n^2), 若确定根时使用 Knuth 改进的方法
// 空间复杂度:
//      S(n)=O(n^2)
//
auto OBST(const std::vector<double> & P,
    const std::vector<double> & Q, size_t n)
{
    // 辅助空间: O(n^2)
    // 补偿 w[0...n+1, 0...n]
    std::vector<std::vector<double>> w;
    // 代价 e[0...n+1, 0...n]
    std::vector<std::vector<double>> e;
    // 为了重构解表示 a_i...a_j 的根的位置
    // root[0...n, 0...n]
    std::vector<std::vector<size_t>> root;

    // 初始化
    // 零向量 1 * (n + 1)
    std::vector<double> zeroDoubleVector;
    std::vector<size_t> zeroSizeVector;
    for(size_t i = 0; i < n + 1; i++)
    {
        zeroDoubleVector.push_back(0.0);
        zeroSizeVector.push_back(0);
    }
    // w、e、root赋初值
    // (n + 2) * (n + 1)
    for(size_t i = 0; i < n + 2; i++)
    {
        w.push_back(zeroDoubleVector);
```

```cpp
        e.push_back(zeroDoubleVector);
        root.push_back(zeroSizeVector);
    }

    // 动态规划填表
    // 初始化
    for(size_t i = 1; i <= n + 1; i++)
    {
        e[i][i - 1] = Q[i - 1];
        w[i][i - 1] = Q[i - 1];
    }
    // l 节点的树
    // 循环 l 次
    for(size_t l = 1; l <= n; l++)
    {
        // 迭代树的起点
        // 循环 n - l + 1 次
        for(size_t i = 1; i <= n - l + 1; i++)
        {
            // 终点
            size_t j = i + l - 1;
            e[i][j] = std::numeric_limits<double>::infinity();
            w[i][j] = w[i][j - 1] + P[j] + Q[j];
            // 找根的位置
            for(size_t r = i; r <= j; r++)
            {
                double t = e[i][r - 1] + e[r + 1][j] + w[i][j];
                if(t < e[i][j])
                {
                    e[i][j] = t;
                    root[i][j] = r;
                }
            }
        }
    }
    return std::make_pair(e, root);
}
```

```cpp

// 打印解
// 参数:
//      words[]: 关键字
//      root[i][j]: OBST算出的根位置
//      i, j: 从 i 开始到 j 结束的树
// 输出:
//      最优二叉树(按数组的形式先根打印)
// 时间复杂度: (认为打印节点是基本操作)
//      T(n) = O(n)
void PrintSolution(const std::vector<std::string> & words,
    const std::vector<std::vector<size_t>> & root,
    size_t i, size_t j)
{
    // i > j, 表示空节点
    if(i > j)
    {
        std::cout << "NULL ";
        return;
    }
    // 因为会打印 n 个内节点和 n+1 个外节点
    // 所以打印次数为 O(n)
    // 打印根
    std::cout << words[root[i][j] - 1] << " ";
    // 打印左子树
    PrintSolution(words, root, i, root[i][j] - 1);
    // 打印右子树
    PrintSolution(words, root, root[i][j] + 1, j);
}

int main()
{
    // 关键字
    std::vector<std::string> words = {"do", "if", "read", "while"};
    // P[0] 补0
    std::vector<double> P = {0, 3, 3, 1, 1};
    // Q
```

```
    std::vector<double> Q = {2, 3, 1, 1, 1};
    auto result = OBST(P, Q, words.size());
    std::cout << "BST is: " << std::endl;
    PrintSolution(words, result.second, 1, words.size());
    std::cout << std::endl;
    std::cout << "Cost: " << result.first[1][words.size()] << std::endl;
    return 0;
}
```

五、运行结果截图

运行 OBST.cpp，程序以课件上的例子做测试（注：老师给的代码与书上的伪代码略有不同。如果按照书上的计算方式，则代价应该为 40；而老师的方法计算结果是 32，没有考虑最底层的外节点。两种方法得到的树是一致的。）

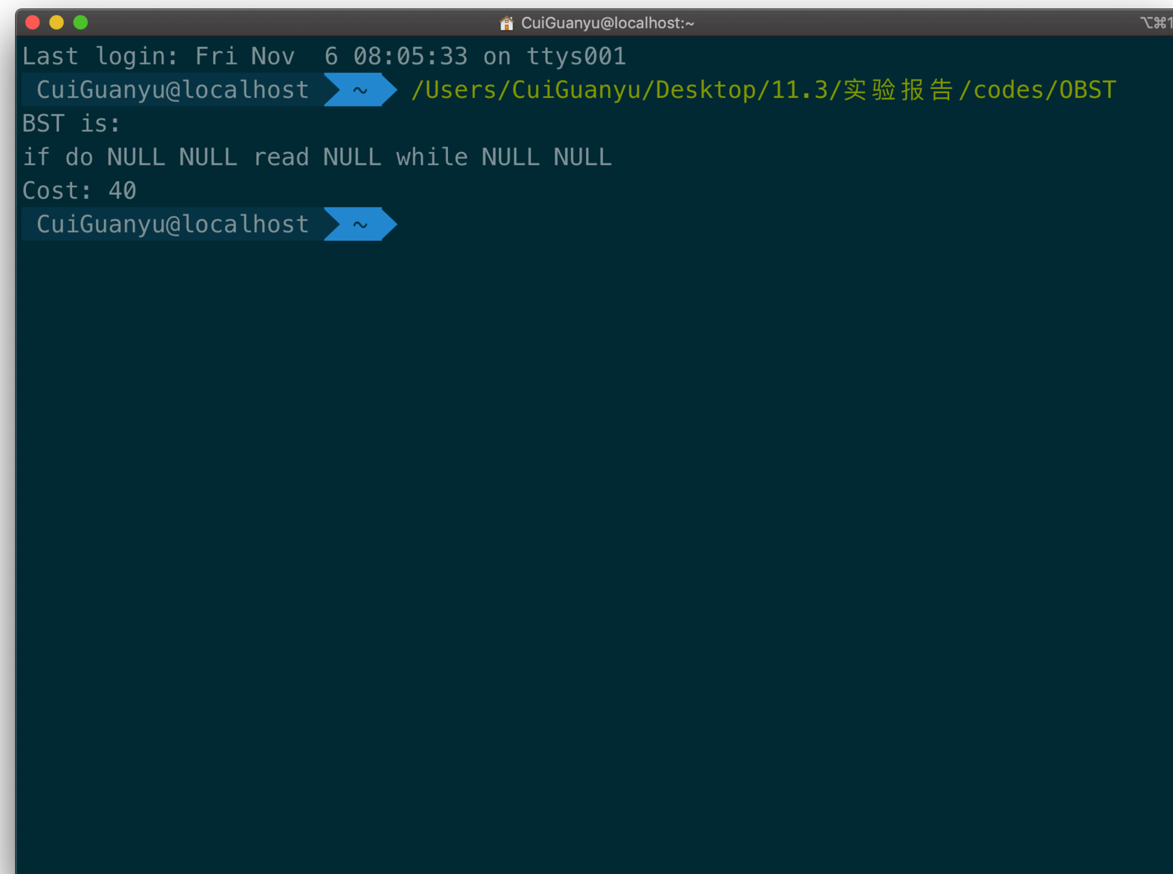

图 2: 最优二叉搜索树测试

可见程序运行正确。