第九周实验报告

高效IP路由查找实验

2015K8009922021
李一苇

一、实验内容

- 实现基本的单比特前缀树查找实验，生成Golden result
- 实现多比特（2，3，4比特）前缀树及优化（叶推、压缩指针、压缩向量），利用Golden result验证正确性
- 比较上述版本的内存开销和平均单次查找时间

二、实验流程

本实验按照以下流程进行：

1. 实现基本的单比特前缀树查找实验(prefix_2node.c)
    - 用`output_finding_result`函数读取`forwarding-table.txt`里的每一行记录，用`find_node`函数得到IP地址的端口值（注意并不是读到的`real_iface_id`值，因为可能存在更长的前缀能覆盖本条目对应的端口值），将IP地址串（用于查错）、32位IP整数、正确端口值输出到`golden-result.txt`文件中

```
void output_finding_result(pt_node *root) {
    FILE* file = fopen("forwarding-table.txt", "r");
    FILE* o_file = fopen("golden-result.txt", "w+");
    char ip_string[BUF_SIZE];
    int ip_mask;
    int real_iface_id;
    while (!feof(file)) {
        //parse input line
        if (fscanf(file, "%s %d %d", ip_string, &ip_mask, &real_iface_id) < 0)
break;
        u32 random_tail = rand() % ((1 << (32 - ip_mask) - 0)) + 0;
        u32 ip_int = parse_ip_addr(ip_string);
        int iface_id = -1;
        find_node(root, ip_int, 0, &iface_id);
        fprintf(o_file, "%s %x %d\n", ip_string, ip_int, iface_id);
    }
}
```

    - `find_node`函数

```
void find_node(pt_node *node, u32 ip_int, int depth, int *last_match_iface_id)
{
    if (node == NULL) return;
    if (node->interface_id != -1) *last_match_iface_id = node->interface_id;
    int nx_step = (ip_int & (1 << (31 - depth))) > 0 ? 1 : 0;
    pt_node* nx_node = node->child[nx_step];
    find_node(nx_node, ip_int, depth + 1, last_match_iface_id);
}
```

注意这里不能写成int函数返回的形式，因为查找结点时，匹配可能并不会发生在递归的最深层，如果最深层匹配失败，应该返回最后一次匹配的结果。所以，必须写成尾递归的形式，不断修正返回参量，实现功能。

2. 实现多比特（2，3，4比特）前缀树：
    - 编写模式
        - `build_node`生成多比特树
        - `leaf_pushing`叶推修改多比特数
        - `create_pt_internal_tree_from_pt_tree`从多比特数生成压缩指针树
        - `create_cv_tree_from_internal_tree`从压缩指针树生成压缩向量树
        - `output_finding_result`对比Golden Result
        - `destroy_xxtree`销毁生成的树

      这样写保证在一个文件`prefix_nnode.c`中实现所有功能
    - 对比特数进行宏定义，以便重用代码

```
#define BIT_STEP 4
#define CHILD_NUM (1 << BIT_STEP)
```

    - 多比特数的数据结构

```
typedef struct pt_node {
    int interface_id;
    int mask_len;
    struct pt_node* child[CHILD_NUM];
} pt_node; //prefix_tree_node
```

    - 扩展单比特树的`find_node`中的`nx_step`变量，让其根据`ip_int`和当前深度`depth`返回，在n比特下匹配到的下一条路径

```
int get_next_step(u32 ip_int, int depth) {
    u32 mask = 0;
    for (int i = depth; i < min(depth + BIT_STEP, 32); i++) {
        mask |= 1 << (31 - i);
    }
    mask &= ip_int;
    return mask >> (32 - min(depth + BIT_STEP, 32));
}
```

在该函数中，`mask`得到`ip_int`的`depth`位到`depth+BIT_STEP`位的结果并返回，并进行边界条件的处理

- 插入函数`insert_node`

  分为两个部分，如果`depth+BIT_STEP`小于读入条目的掩码长度`mask_len`说明，应插入子节点，并进行内存分配；否则，应插入叶节点，此时应插入多个叶节点（从`initial_nx_step`到`until_nx_step-1`），保证在一个`BIT_STEP`内的所有叶子都分配到这个端口值。同时记录下条目的掩码，以便和之后的条目的掩码值比较，将叶子更新为掩码值大的条目的端口值。

```
void insert_node(pt_node *node, u32 ip_int, int mask_len, int iface_id, int
depth) {
    int nx_step = get_next_step(ip_int, depth);
    pt_node* nx_node = node->child[nx_step];
    if (depth + BIT_STEP < mask_len) {
        if (nx_node == NULL) {
            nx_node = (pt_node*)malloc(sizeof(pt_node));
            init_node(nx_node);
            node->child[nx_step] = nx_node;
        }
        insert_node(nx_node, ip_int, mask_len, iface_id, depth + BIT_STEP);
    }
    else {
        int initial_nx_step = get_next_step(ip_int, depth);
        int until_nx_step = initial_nx_step + (1 << (depth + BIT_STEP -
mask_len));
        for (nx_step = initial_nx_step; nx_step < until_nx_step; nx_step++) {
            nx_node = node->child[nx_step];
            if (nx_node == NULL) {
                nx_node = (pt_node*)malloc(sizeof(pt_node));
                init_node(nx_node);
                node->child[nx_step] = nx_node;
            }
            if (nx_node->mask_len < mask_len) {
                nx_node->interface_id = iface_id;
                nx_node->mask_len = mask_len;
            }
        }
    }
}
```

3. 实现叶推

叶推函数严格执行：对子节点是中间节点，且有空子节点的节点，将端口值下放到子节点中（即使端口值为空），递归执行。

这样保证了新生成的前缀树的边界时都是叶子节点；每个节点或者为叶子结点或者为内部结点。

```
void leaf_pushing(pt_node *node) {
    if (is_node_leaf(node)) return;
    int iface_id = node->interface_id;
    node->interface_id = -1;
    for (int i = 0; i < CHILD_NUM; i++) {
        pt_node *child_node = node->child[i];
```

```
            if (child_node == NULL) {
                child_node = (pt_node*)malloc(sizeof(pt_node));
                init_node(child_node);
                node->child[i] = child_node;
            }
            if (child_node->interface_id == -1)
                child_node->interface_id = iface_id;

            leaf_pushing(child_node);
        }
    }
```

4. 实现压缩指针

- 压缩指针和压缩向量共享一个数据结构

  `leaf`可看成`pt_childleaf_node`数组，存储一系列的`iface`和`mask_len`

  `child`可看成指针数组，存储一系列下一个内部结点的地址

```
typedef struct {
    int iface_id;
    int mask_len;
} pt_childleaf_node;

typedef struct {
    struct pt_internal_node* node;
} pt_childinternal_node;

typedef struct pt_internal_node {
#if BIT_STEP == 4
    u16 bitarr;
#elif BIT_STEP == 3
    u8 bitarr;
#else
    unsigned bitarr : CHILD_NUM;
#endif
    pt_childleaf_node* leaf;
    pt_childinternal_node* child;
} pt_internal_node; //compressed vector/pointer tree node
```

- 生成压缩指针树

  如果多比特前缀树的某结点为叶子结点，则直接把`iface_id`赋给压缩指针树的`leaf`相应位上；如果为非叶子结点，则修改压缩指针树的`bitarr`相应位为1，且添加子树，并递归执行。

  在`init_inode`函数中，进行内存分配，直接为每个节点分配`CHILD_NUM`个叶子结点指针和`CHILD_NUM`个内部结点指针。

```
void create_pt_internal_tree_from_pt_tree(pt_internal_node *inode, pt_node
*node) {
    if (is_node_leaf(node)) return;
    for (int i = 0; i < CHILD_NUM; i++) {
        if (is_node_leaf(node->child[i])) {
```

```
            inode->leaf[i].iface_id = node->child[i]->interface_id;
        }
        else {
            inode->bitarr |= 1 << i;
            inode->child[i].node =
(pt_internal_node*)malloc(sizeof(pt_internal_node));
            init_inode(inode->child[i].node);
            create_pt_internal_tree_from_pt_tree(inode->child[i].node, node-
>child[i]);
        }
    }
}
```

5. 实现压缩向量前缀树
   - 生成压缩向量前缀树

```
void create_cv_tree_from_internal_tree(pt_internal_node *cvnode,
pt_internal_node *inode) {
    cvnode->bitarr = inode->bitarr;
    int ichild_num = popcount((u32) inode->bitarr);
    cvnode->child =
(pt_childinternal_node*)malloc(sizeof(pt_childinternal_node) * ichild_num);
    cvnode->leaf = (pt_childleaf_node*)malloc(sizeof(pt_childleaf_node) *
(CHILD_NUM - ichild_num));
    int child_idx = 0, leaf_idx = 0;
    for (int i = 0; i < CHILD_NUM; i++) {
        int is_child_node = inode->bitarr & (1 << i);
        if (is_child_node) {
            cvnode->child[child_idx].node =
(pt_internal_node*)malloc(sizeof(pt_internal_node));
            init_cvnode(cvnode->child[child_idx].node);
            create_cv_tree_from_internal_tree(cvnode->child[child_idx].node,
inode->child[i].node);
            child_idx++;
        }
        else {
            cvnode->leaf[leaf_idx].iface_id = inode->leaf[i].iface_id;
            leaf_idx++;
        }
    }
}
```

在为每个节点分配内存时，根据压缩指针建好的树，用`popcount`函数得知各结点的叶子数和子节点数，分配相应的紧凑内存，用`child_idx`和`leaf_idx`指向最新的子节点和叶子。

   - 对比Golden result

在`output_finding_result`函数中：

```
if (fscanf(file, "%s %x %d", ip_string, &ip_int, &real_iface_id) < 0) break;
if (!DEBUG_FLAG) {
    if ((rand() % 10000) < PROB_TEST) continue;
```

```
}
int iface_id = -1;

start = getSystemTime();
for (int i = 0; i < RETRY_TIME; i++) {
    find_node(root, ip_int, 0, &iface_id);
}
end = getSystemTime();
sum_tree_time += (end - start);
if (real_iface_id != iface_id) {
    printf("%d %s %x expect: %d got: %d\n", cnt, ip_string, ip_int,
real_iface_id, iface_id);
    break;
}
```

读取Golden result文件，读取`ip_int`和`real_iface_id`，用`ip_int`得到压缩指针树得到的值`iface_id`，对两者比较。

在非DEBUG模式下，随机选择一些条目挑选，记录时间，用于比较最后的结果比较。

-

三、实验结果和分析

正确性：用最基本的前缀树查找结果，在`output_finding_result`中比对，并没有报错。

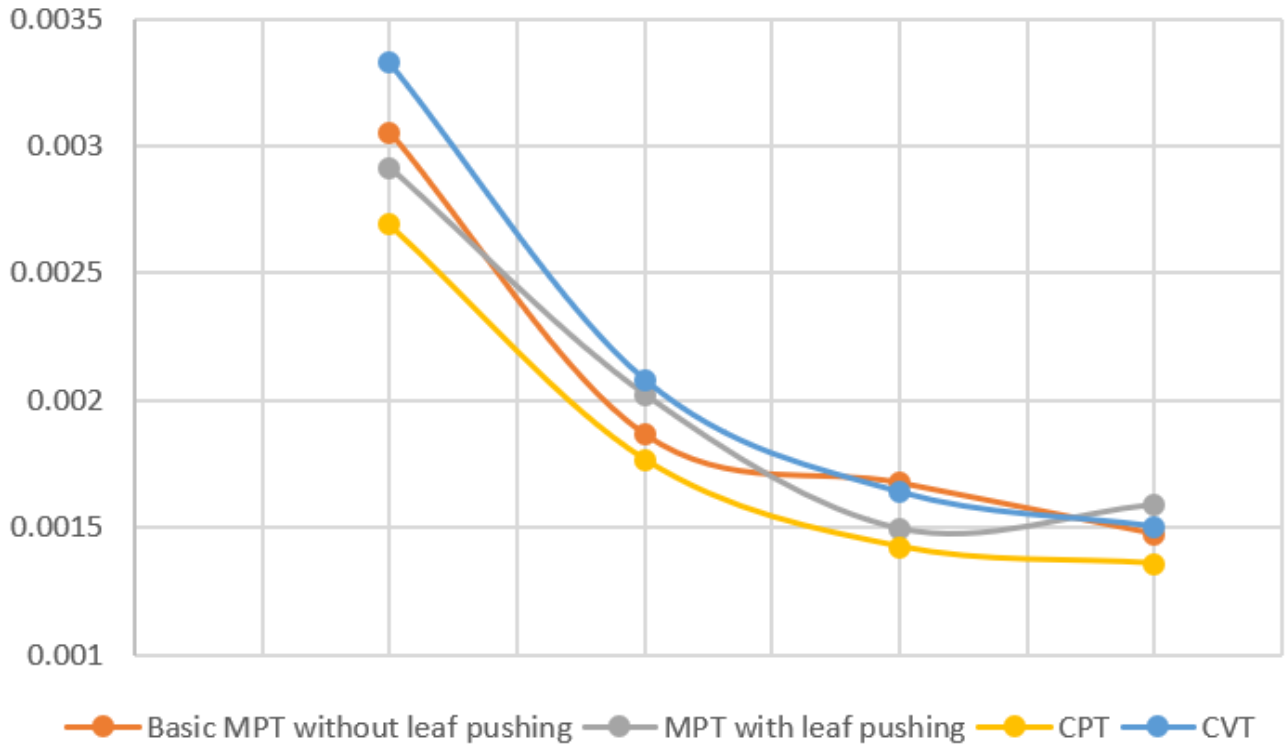
性能对比：

对内存开销与时间进行分析，方法：

- 时间分析：对每个条目的查找重复`RETRY_TIME`(10000)次，保证一个条目的查找时间 大于1ms，最后除以总查找次数`RETRY_TIME * cnt`，得到平均查找时间。
- 内存开销：在每一个`malloc`之后，用类似语句`mem_node_use += sizeof(pt_node);`累加记录新增的内存消耗。注意，在压缩向量中，若单纯累加`sizeof(pt_node)`记录的值是节点个数*单节点的大小，并没有记录指针指向的新结构体的大小。

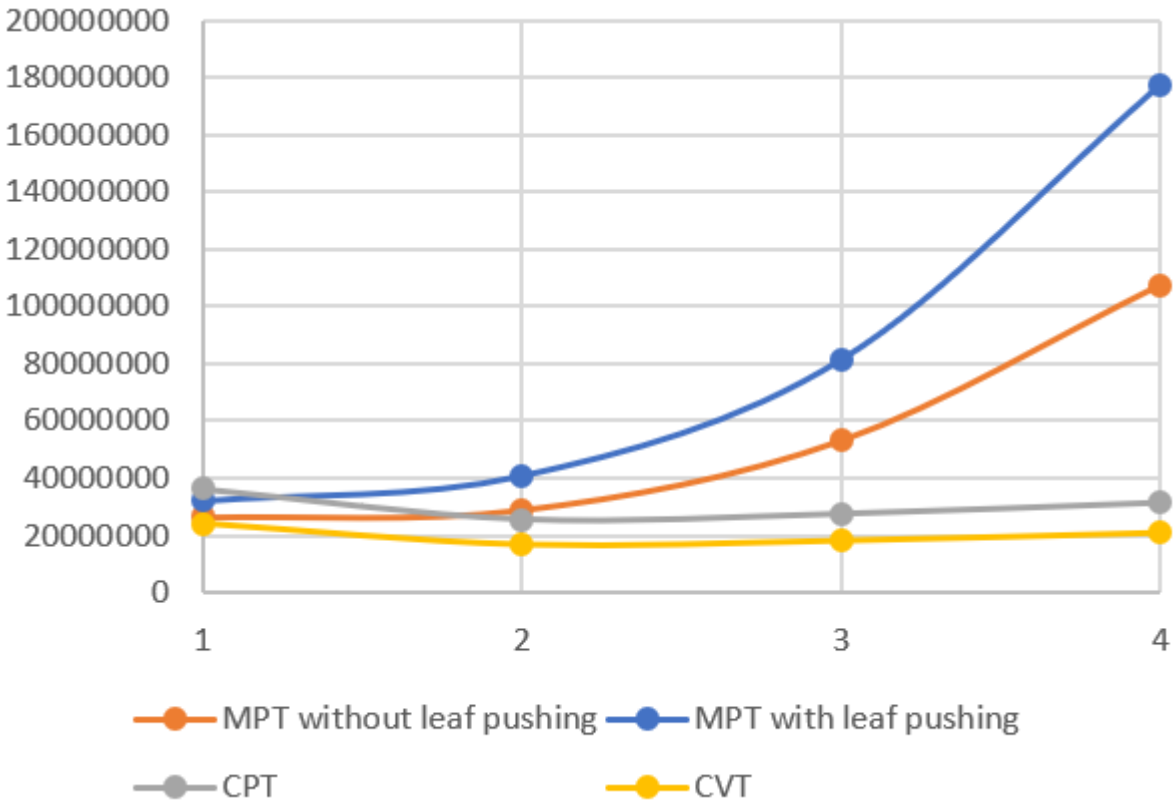
	Bit	Average Finding Time(ms)	Memory Use(Byte)
基本前缀树	1	0.003052778	26345360
叶推优化	1	0.002912778	32160432
压缩指针	1	0.002691667	36180468
压缩向量	1	0.003325556	24120316
基本前缀树	2	0.001864444	28865952
叶推优化	2	0.002021667	40915416
压缩指针	2	0.001767778	25572120
压缩向量	2	0.002080556	17048084
基本前缀树	3	0.001673889	53154640
叶推优化	3	0.001495	81483880
压缩指针	3	0.001425	27500796
压缩向量	3	0.001638889	18333868
基本前缀树	4	0.001473333	107552520
叶推优化	4	0.001589444	177520968
压缩指针	4	0.001358889	31435992
压缩向量	4	0.001503333	20957332

查找时间分析如下：



可见：

- 1. 随着bit数增加，查找时间迅速减小，因为树的结点少了，IO次数少了
  2. CVT(压缩向量树)的查找时间比较高，因为`popcount`的开销
  3. CPT(压缩指针树)的查找时间最低



可见：

- 1. 随着bit数增加，存储空间迅速提高，多比特是空间换时间的算法
  2. 基本的前缀树空间占用提升很快，压缩前缀/压缩指针的树提升较慢，比特数越多，优势越明显
  3. 单比特数下，压缩指针树因为多了指针的空间，内存开销稍高