**网络实验报告 exp9**

**2020K8009929032 刘耕印**

1. **实验内容**

为了改善大规模路由表的查找性能，避免线性查找，应对路由表的存储结构和查找算法进行优化。在本实验中，我们首先实现最简单的trie树存储以及对应的查找算法，之后在其基础上进行优化，以使得性能进一步改善。

1. **实验思路**
2. 基础实现

按照课件的要求，首先进行基础实现。实现使用二叉的trie树存储整个路由表。树有两种节点，即内部节点和匹配节点。在建立trie树的过程中，对于每个给定的路由表条目，我们将走一条从根节点向下的长度为mask的路径：从给定网络号的最高位开始，逐位处理。若该位为0，则检查当前节点的左孩子，若左孩子不存在则需要创建一个；否则该位为1，检查右孩子，没有则创建之。无论是否创建了对应孩子，将指针移到该孩子上。直到最后一次循环，也就是第mask次循环，此时一定是创建了一个孩子节点。这个孩子节点设为匹配节点，并填入条目的端口号。也就是说，任何符合此路由条目（但不一定是最终结果）的ip地址在查找树的过程中，一定曾走过如上所述的路径，并曾经过如上创建的那个匹配节点。这个匹配节点就代表着这一次处理的路由表条目。

在查找时，当前节点初始化为根节点。首先检查本节点是不是匹配节点，若是，就把结果暂时置为本节点描述的端口。从输入ip的最高位开始，是0就检查左孩子，是1就检查右孩子，不断将当前节点指针指向该孩子，直到对应的孩子为空即可退出搜索。这样的过程结束后，结果一定是在整个路径中曾经过的最低匹配节点所描述的端口。当然，如果一路上从没有经过任何匹配节点，就意味着查找失败，此时返回的将是默认值-1。

基础实现的代码见第三部分。

1. Poptrie

我花费了一个下午阅读课件所推荐的顶会论文，即poptrie。现在我可以介绍一下该论文的设计思想：

一是压缩空间，最小化存储结构的规模，使其尽可能能被整个调入cache中，从而大幅减少访存足迹，优化性能。

二是压缩树的高度，使用一个被称为”direct\_pointing”的数组存储预先求好的若干比特前缀的查询结果，每次查询时可以根据ip的该前缀首先查询该数组，然后从该数组指示的位置开始向下遍历子树。于是该数组越大，常数时间查找的部分就越多。当然，该数组长度设为2^32时将会变为完全的常数时间查找，即哈希表。

论文使用的存储结构将内部节点与匹配节点分开，分别放在两个数组中。内部节点会有一个vector成员，相当于该节点的位图（bitmap），指示该节点的孩子中内部节点和匹配节点的分布情况。有一个leafvec成员，帮助指示压缩后的匹配节点的分布情况。还有两个坐标base1和base0，前者指示本内部节点的孩子节点中首个内部节点的index，后者指示首个匹配节点的index。于是，一个内部节点占据的空间只有2\*2^(每次匹配的bit数)+2\*(下标长度)个bit，比起存储大量指针要省空间，而且每次匹配的bit数越多就越省。

direct\_pointing数组的计算是在建树之后的，以该数组的下标为前缀查找树，将会停在某个节点上，将这个节点填进数组的对应位置中。于是相当于预先进行了若干bit的匹配。

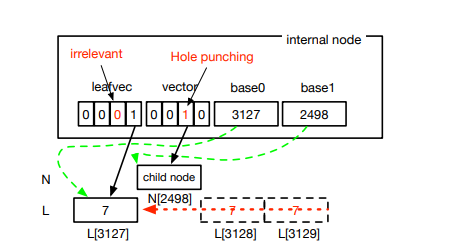


Figure poptrie的存储结构（每层匹配2bit）

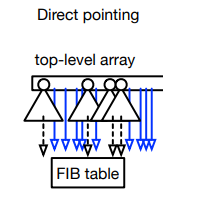


Figure direct\_pointing数组，其中三角形代表内部节点

但是，在尝试实现poptrie的过程中，我在建树这一步失败了。因为在某个内部节点刚刚建立时，无法确定它最终将会有怎样的孩子分布，也就很难为其准确地分配内部节点数组以及匹配节点数组中的空间。如果是C++的话，vector可以很简单地解决这种问题，但在C中相应的内存管理实现起来工作量很可观。因此，我无法实现完整的poptrie。但是，在我后续的实现当中，优化查找性能时，借用到了direct\_pointing的压缩思想。

1. 进阶实现

进阶体现在2-bit匹配和direct\_pointing压缩。我没有考虑优化占用的空间，而是主攻查找性能。

首先，实现基本的2-bit匹配。在基本实现的基础上，考虑每次进行2比特的匹配，以更少的内存访问找到最终结果。这要求内部节点有四个指针，分别对应着该层2bit的00，01，10，11。于是，树的高度可以压缩一半。查找时，从ip的最高位开始每次匹配2bit，按2bit对应的指针向下发展。仍然是遇见空指针即可退出。

针对2bit还有一些额外的改动：

*第一*，路由表可能包含有mask为奇数的条目，在建树和查找时都要考虑到。具体来说，在建树时，相比于mask为偶数（2k）的情况，mask为奇数（2k-1）时要少进行一次一般的循环。偶数时，最后一次循环将创建最低2bit对应的匹配节点，而奇数情况下，第k-1次循环结束后，需要创建最低1bit对应的2个匹配节点。如果最低1bit是0，那么创建子节点00和01；否则创建子节点02和03。总的来说，mask为2k的条目将创建/经过k-1个内部节点（不包括根节点），并创建一个匹配节点（填写其port\_even）；mask为2k-1的条目也将创建/经过k-1个内部节点，并创建两个匹配节点（填写其port\_odd）。（其中port\_even和port\_odd是内部节点不同的成员）而在查找时，在正常的2bit逐级匹配过程中，如果发现了匹配节点，若该节点有port\_even的话，优先将结果设为port\_even，否则才设为port\_odd。这是因为，同一个匹配节点中，port\_odd是由mask为2k-1的条目赋值的，而port\_even是由mask为2k的条目赋值的。如果已经来到了这个匹配节点，就意味着该ip能够匹配mask为2k的条目，因此按最长前缀规则应当选择port\_even。

*第二*，内部节点中指向四个孩子的指针使用一个长度为4的数组组织起来，这样可以直接使用待匹配的2bit作为下标取出对应指针，免去了繁琐的条件判断。

然后，在建好树的情况下，实现direct\_pointing机制。选定数组长度后，就以数组的每个下标为输入，在树中进行遍历查找，将停止的位置填入数组中。实现时还有一些细节的问题，我将在第三部分中介绍。

1. **代码实现**
2. 基础实现

首先建树，初始化：

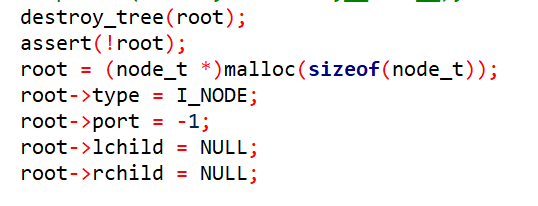


Figure 基础实现——建树初始化

读入所有路由表条目：

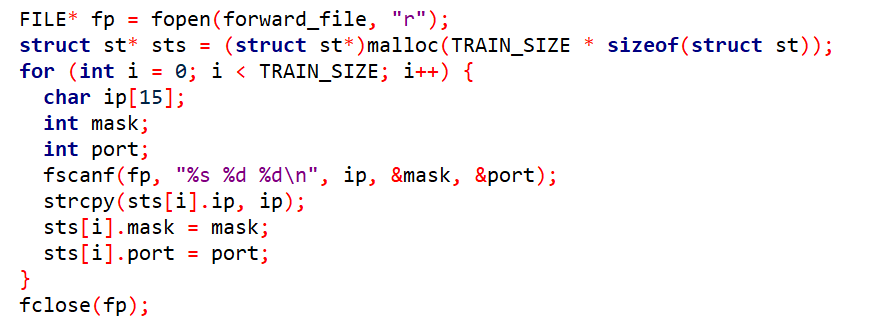


Figure 基础实现——读入路由表

循环处理每个条目。处理过程大致可以概括为向下逐位匹配，一路创建路径上的未创建节点，

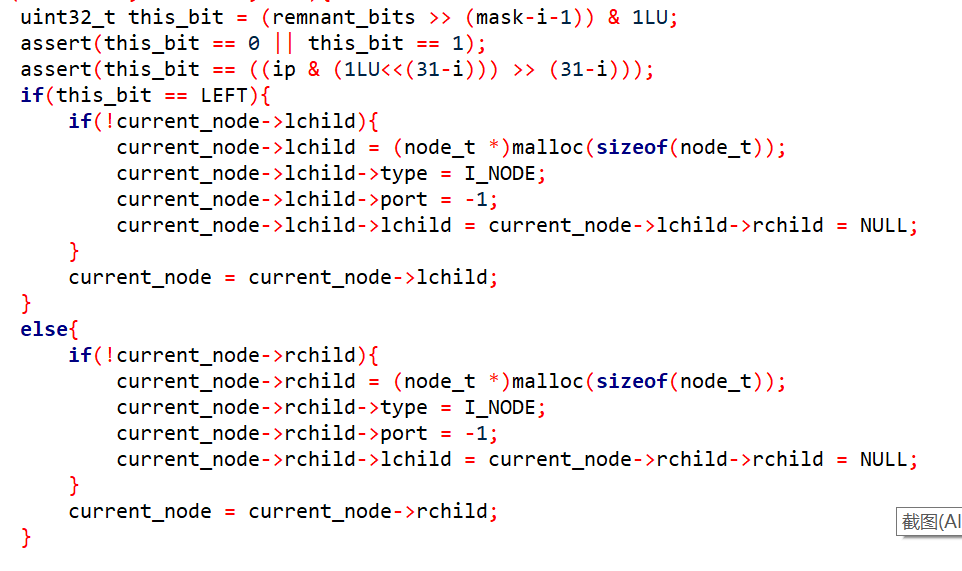


Figure 基础实现——向下匹配并创建节点

最后创建的是匹配节点。

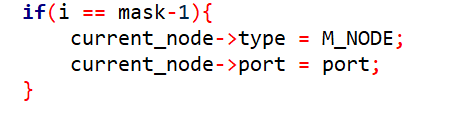


Figure 基础实现——创建匹配节点

然后实现查找，对给定ip只需逐位匹配，向下走一条以空节点为终点的路径即可。过程中时刻更新最低的匹配节点描述的端口。

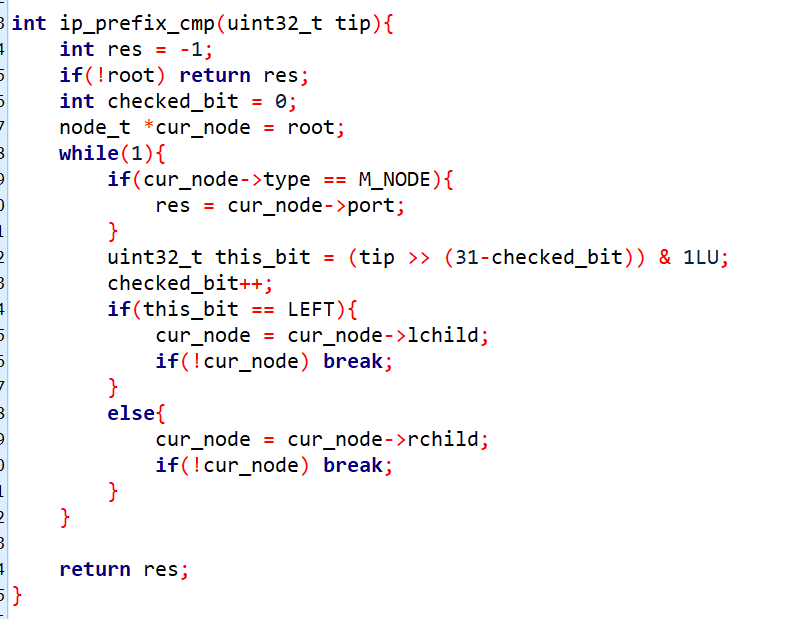


Figure 基础实现——查找

2．进阶实现

建树时，根据mask的奇偶性，决定最后创建几个匹配节点，以及匹配节点里填写哪个端口。

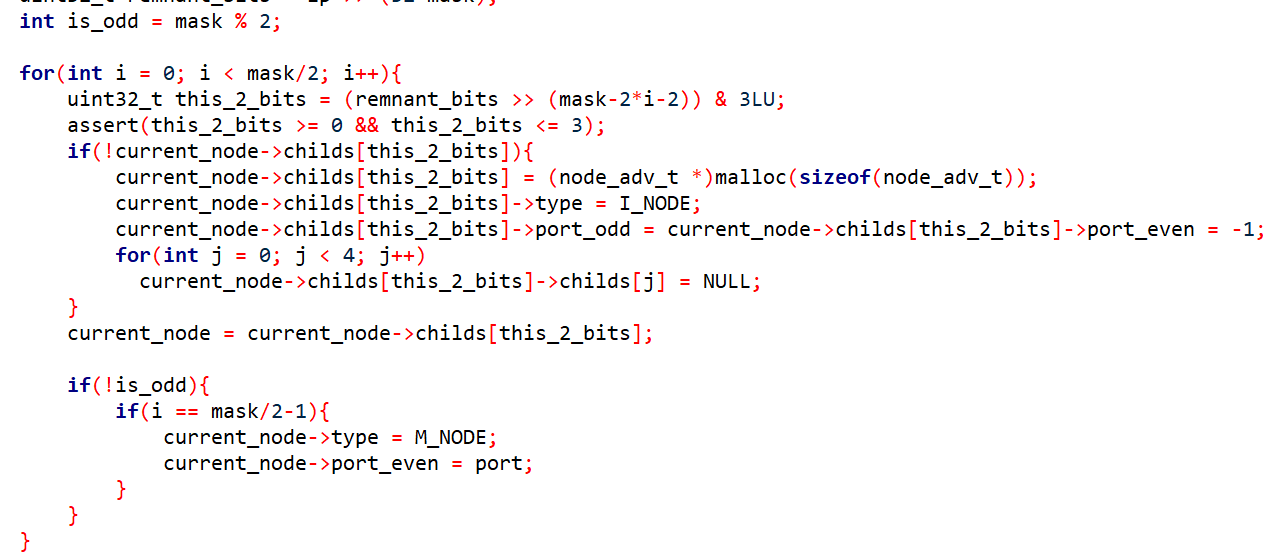


Figure 进阶实现——向下行走

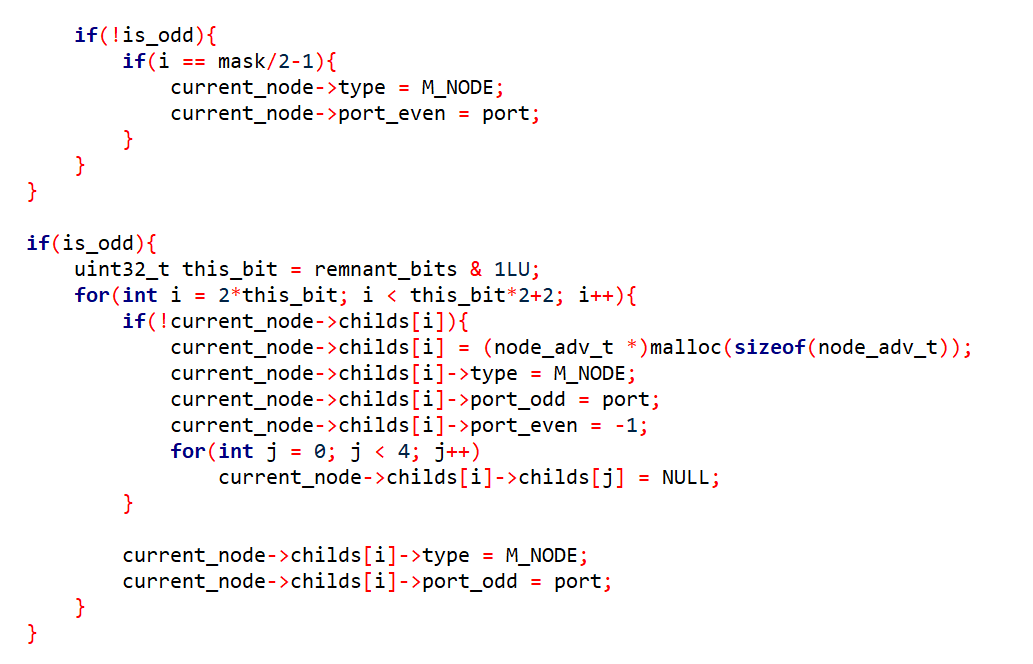


Figure 进阶实现——创建匹配节点，正确填写端口号

在建树完成后，还需要计算direct\_pointing数组。除了上面提到的过程外，还有一个要点要注意：搜索是有可能在达到direct\_pointing数组描述的指针之前就结束（后面路径上不再有匹配节点）的。于是，需要在数组中保存建立数组过程中遇到的最优端口号，供后续查找使用。实现方法就是在数组中加一个描述端口的成员，建立数组时把它更新为最优值。

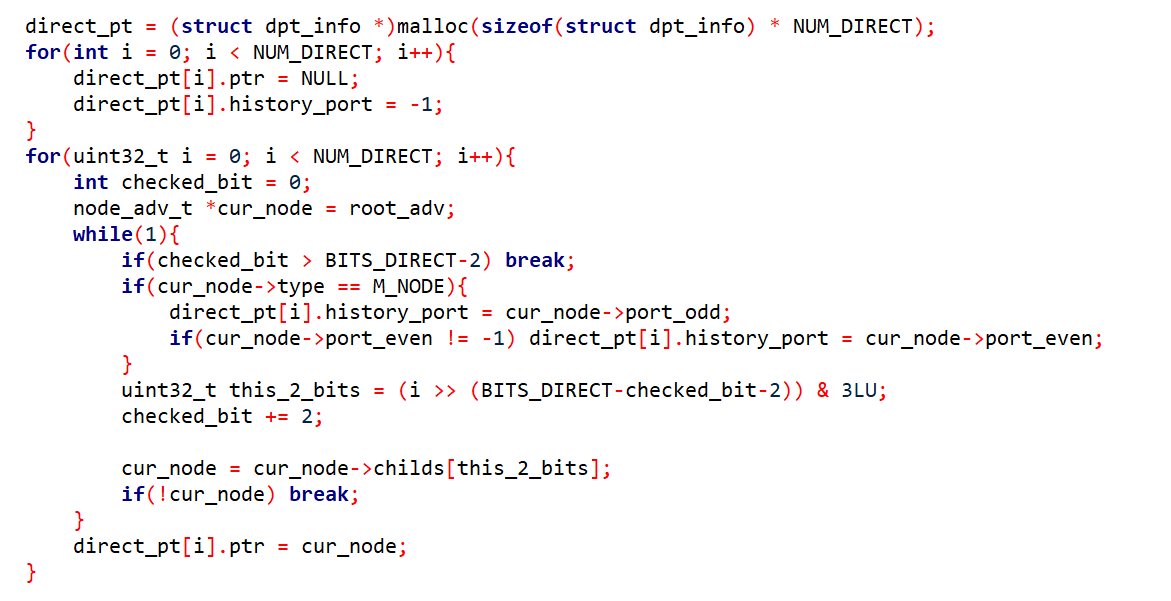


Figure 进阶实现——建立direct\_pointing数组。类似一个以数组下标为ip的较短暂查找过程，查找结果保存在数组中，停止节点也保存在数组中。

然后是查找过程。与基本实现的区别只在于每次匹配2bit，并且初始节点从direct\_pointing数组中取得。需要注意的是，res的初值设为direct\_pointing的记录端口值，以保证两段搜索（建立direct\_pointing数组时和实际搜索时）的结果能够衔接上。另外，可能在建立数组时就已经遇见空节点而终止了，这时直接返回数组记录的端口号即可。代码如下。

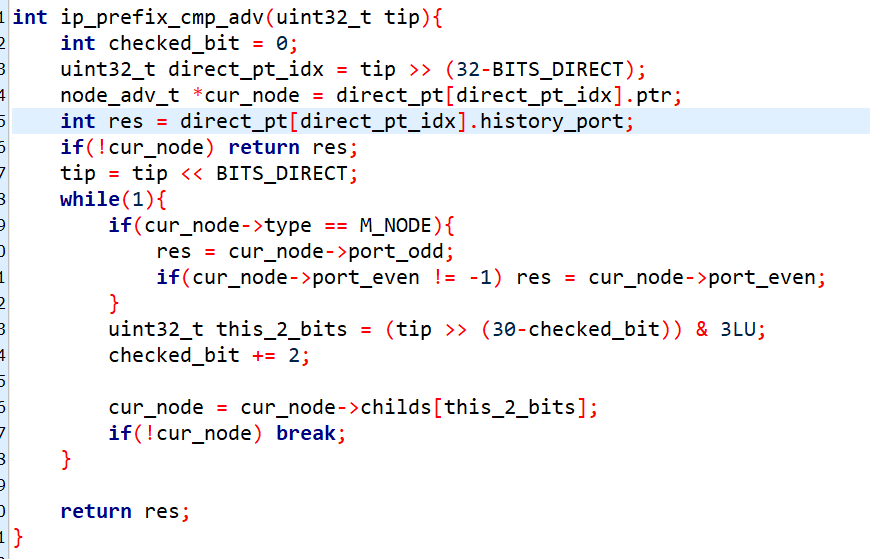


Figure 进阶实现——查找

1. **性能评估**

由于我没有进行任何专门的空间优化，因此进阶实现的空间消耗并不比基础实现少很多。但是，进阶实现减少了树的高度，内部节点的数量也大大减少，因此空间上也可以说有优化。

当将direct\_pointing数组的长度设为2^16时，执行效果如下：

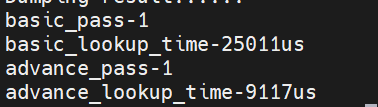


Figure 时间优化

可以看到，效率有约3倍的优化，达到了要求。实际上，把数组长度设得更大时，效率还可以更高，本质上是一个用空间换时间的操作。

1. **实验结果**

****