基於 R*-Tree 的位元圖交集封包分類演算法

黃鼎峰　　陳 健　　陳盈羽

國立交通大學, 資訊工程學系

artines1.cs97g@nctu.edu.tw　chienchen@cs.nctu.edu.tw　gis91564@cis.nctu.edu.tw

***摘要*—隨著網際網路的蓬勃發展，網際網路服務如網路安全防護、虛擬私有網路、品質保證等的需求也越來越高。為了達到這些服務，網際網路路由器必須針對收到的封包做出快速的分類。封包分類是利用封包標頭中所包含的資訊與路由器中事先定義好的規則做比對，依據比對出來的結果，執行不同的動作；而多重欄位的封包分類是一個相當困難的問題，已經有許多的演算法被提出來解決這個問題。在這篇論文中，我們提出了一個透過結合位元圖交集(Bitmap Intersection)演算法與 R*-Tree 封包分類演算法來達到快速封包分類的演算法，我們所提出的方法無論是在封包分類速度或是所需要的記憶體上，比起這兩種方法都有著顯著的進步。**

***關鍵詞*—Packet Classification, R-Tree, R*-Tree, Bitmap Intersection**

一、簡介

網際網路已經成為我們日常生活中的一部分，許多不同的應用也相繼出現在網際網路上，而針對這些越來越多樣化的應用，網際網路服務也變得越來越重要，例如網路安全防護、虛擬私有網路、品質保證等等。要實現這些網路服務，皆需要路由器有能夠分類封包的能力。但隨著網路速度的增加，一般的封包分類演算法已經無法負荷高速的網路速度，如何能夠加速封包比對的速度是目前路由器設計的重要問題。

路由器在執行封包分類的時候，是利用封包標頭的資訊與路由器內部的規則表作比對，來找出對應的規則。規則表是由多個規則所組成的，一個規則由多個欄位組成，這些欄位可能會是一個 Prefix (如 IP source/ destination IP address) 、一段值域 (如 source /destination port) 、或是一個數值 (如 protocol type) ，另外每個規則各自會有不同的優先權以及所對應的動作，以網路安全防護為例，動作可能是允許或是拒絕接收此封包。

當一個封包到達路由器時，首先會擷取出封包的標頭以取得分類時所需要的資訊，也就是對應到規則的那些欄位。然後去規則表中找尋對應到此封包的規則;所有欄位都必需要符合才是所需要的規則，規則表中可能會有許多規則跟這個封包符合，此時擁有最高優先權的規則才是最後所需要的規則，路由器則依照此規則所預先定義的動作來處理這個封包。

傳統的封包分類問題可以把它視為在多維圖形空間點位置的搜尋問題 [8]。在多維空間中，我們可以把規則視為存在於空間中的多維矩形，也就是說，規則中的各個欄位代表此矩形在不同維度中所涵蓋的區域，而封包的標頭資訊則是在多維空間中的一個點。如此一來，我們就可以把傳統的封包分類問題看成是在多維空間中搜尋包含某個點之所有矩形的問題。圖 1.1 為一個規則表對應到二維空間的範例。此範例中總共包含四個維度為二的規則以及一個封包 P，規則分別被對應到圖形空間中部分的區域。把封包 P 對應到空間後，我們可以得到其所對應的規則，在此範例中即 R3 與 R4，其中又由於 R3 的優先權較高，所以最後會得到到 R3 這個規則。

上述的問題是一個很困難的問題。傳統解決方法不是需要大量的儲存空間，就是比對的時間過長。一個好的封包分類演算法必須同時具備快

速的分類以及少量的儲存空間的特性，因此在本篇論文中我們提出了結合位元圖交集演算法[12]與 R*-Tree 封包分類演算法[2]的方法，能夠同時滿足分類速度快與儲存空間少的需求。

本篇文章接下來的部分包括:第二章會介紹封包分類問題先前的相關研究；第三章將會提出我們的演算法以及演算法的效能分析比較；在第四章中，我們經由模擬比較我們的方法以及其他文獻中方法的效能；最後第五章則是對於此篇文章的結論以及未來展望。

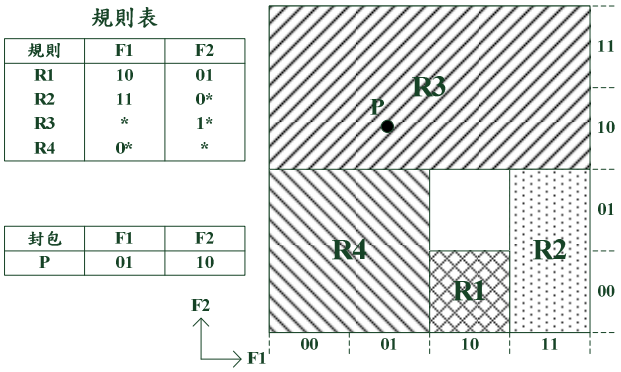

圖 1. 1 規則表對應圖形空間範例

二、相關研究

在封包分類這個領域中，有許多的論文以及演算法被提出用以解決這個問題，大致上可以分作為下面幾類:Trie-based 演算法、圖形空間演算法、探索式演算法、以及 TCAM-based 演算法等四大類 [7]。我們的方法是屬於圖形空間演算法類別，在此我們簡單扼要地描述幾個圖形空間演算法的相關研究，其他更完整詳細的封包分類演算法介紹可以參考下列論文 [4][6][9]。

HiCuts [10]主要是把所需要的空間依照不同的維度做出切割，每一次的切割會把某個維度切成多個等分的區塊，每個區塊中則會包含在這個切割範圍內的規則，而這些區塊如果所包含的規則數過多的話，會再以同樣的方式切格其他維度，分成更多的區塊。以上步驟將重複直到區塊中的規則數小於一定數量或是所有的維度都被切割完為止。最後再使用一個決策樹來索引這些切割的區塊。在做封包分類的時候，它會依照切割的資訊來搜尋這個決策樹，找到對應的樹葉節點。此節點內會包含可能會對應到此封包的規則，由於其包含的數量不會太多，所以這邊會使用線性搜尋的方法來找到對應的規則，而這個方法的效能主要會受到實際規則表的分佈影響，所以它的效能比較難以被評估。

HyperCuts [11]和上面所提到的 HiCuts 非常的相似，主要的差別在於 HiCuts 切割時一次只會依照一個維度作切割，而 Hyper-Cuts 則是於單次切割中會對多個維度作切割。Hyper-Cuts 也同樣使用一個決策樹來索引它所切出來的空間資訊。在 Hi-Cuts 中決策樹內的節點會記錄著單維的切割資訊，而在 Hyper-Cuts 中則是紀錄多維的切割資訊，也就是它會利用多維陣列來紀錄對應多維切割的資訊。在效能上 Hyper-Cuts 有著不錯的速度，也不需要很大的記憶體空間，但是這個方法對於百搭符號的規則(wildcard rule)有較差的效能。

Lakshman et al. [12] 所提出的位元圖交集 (Bitmap Intersection) 封包分類演算法，其主要的精神是 divide–and-conquer，把封包分類的問題切成數個子問題，最後再把所有的解合併得到真正的結果。這個演算法同樣也是基於圖形空間的封包分類法。假設在同型空間中總共包含 N 個規則，它最多會在每個維度上都產生 2N+1 個不重複的圖形區間，每一個區區間都會包含一個 N 位元長度的位元向量(Bit vector)，位元向量中的第 j 個位元在規則 j 有重複到此圖形區塊的情況下會被設定成 1，反之的話會設成 0。如圖 2.1 中，X1 區間中只有包含 R4，所以第四個位元被設定成 1，而其他規則並沒有包含在此區間中，所以其他的位元被設定為 0。

當封包被送來的時候，首先利用二元搜尋把每個維度中對應到這個封包標頭資訊的圖形區間所屬的位元向量取出來，接下來把取出來的這些位元向量作一個交集的動作，產生另一個的位元向量。在此位元向量中所有被設定的位元就代

表比對到的規則，其中優先權最高的規則就是我們想要的規則。由於規則事先以依照其優先權作遞減順序的排列，所以在這位元向量中第一個被設定的位元所對應的規則就是優先權最高的規則。圖 2.1 呈現一個簡單的例子，假設存在一個封包 P，其對應到的區間分別為 X6 以及 Y3，它們所屬的位元向量分別為 0011 與 1111，則我們得到的交集結果位元向量為 0011，所比對到的規則是 R3 和 R4，因為 R3 的優先權比較高，所以得到的規則就為 R3。

這個方法因為把不同維度的比對分開執行，所以可以同時的對所有維度作比對的動作，這可以利用實作在硬體的方式來達到高速的封包分類。此方法在規則數量變大的時候則會有所需的儲存空間變大以及比對所需要的時間變多等缺點，原因是因為當規則數變大的時候就代表需要更長的位元向量以及更多的圖形區間，所以所需要的儲存空間會變大;而由於位元向量變長，所以做比對的時候取出位元向量的時間會變長，導致比對的時間變長。

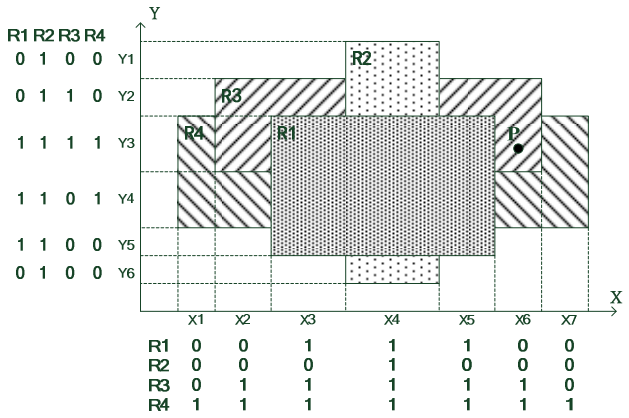

圖 2. 1 位元圖交集法

R*-Tree 演算法 [2] 是一個用來索引圖形空間中物件的一個演算法， Maindofer et al. [3]提出了把 R*-Tree 使用在圖形空間的封包分類上的想法。R*-Tree 本身的架構跟 R-Tree [1]十分的類似，它用階層式架構來索引圖形空間的物件。這些物件必須要是矩形的，在 R-Tree 中圖形物件會被視為最小包含區塊(Minimal Bounding Region，MBR)，數個 MBR 會由一個更大的 MBR 包含，這些 MBR 又會再被一個更大的 MBR 包含，直到最大的 MBR 包含到整個圖形空間為止。以圖 2.2 中左邊圖為例，MBR B 包含 a、b、c、d 四個子 MBR。R-Tree 中的節點記錄著某個 MBR 的資訊以及它所包含的子 MBR 對應的節點在記憶體中的位置。若一個 MBR 內所包含的子 MBR 的個數為 X，則 $m \le X \le M$，其中 m 表示節點中包含 MBR 數量的最少個數，M 表示最大的個數。圖 2.2 為節點的架構圖，節點 B 記錄下一層的 MBR，包括 a、b、c、與 d 所包含空間的範圍資訊以及所對應節點的位置。當我們想要搜尋空間中某個區塊的資訊時，從根節點開始找尋與搜尋區塊有重疊到的 MBR，再到這些 MBR 對應的節點中搜尋，重複這些動作直到到達樹葉節點為止，最後在這個節點中與搜尋區塊重疊到的 MBR 就是我們所找到的物件。例如在搜尋圖 2.2 中的 B 節點時，在此會使用線性搜尋的方式，依序地讀取節點中 MBR 的空間資訊以及比對是否與搜尋的區域重疊。假設搜尋的空間與 c 重疊的話，則會繼續往 c 對應的節點作搜尋。

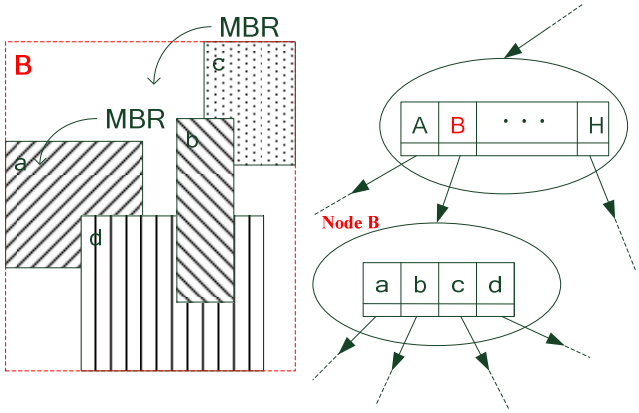

圖 2. 2 R*-Tree 節點架構

R*-Tree 與 R-Tree 兩者十分的相似，它們的基本架構相同。兩者相較，R-Tree 在搜尋上效能較差，是因為 R-Tree 中 MBR 之間重疊比率較高，搜尋時需要走較多的路徑，如圖 2.2 中，若我們想要搜尋的物件屬於 b，並且它的位置位於 b 與 d 重疊的區塊中，在搜尋時我們並不知道物件是位於 b 還是 d，因此這兩個 MBR 我們都需

要搜尋，但事實上 d 中並沒有我們想要的物件，所以搜尋 d 會造成多餘的花費，這個問題會讓搜尋速度降低。R*-Tree 使用強制重新插入以及較好的建樹演算法改善此缺點，所以 R*-Tree 在搜尋的效能上比 R-Tree 要好。

把分類規則對應到 R*-Tree 上被索引的圖型物件，封包標頭資訊對應到搜尋的區塊，如此一來我們就可以使用 R*-Tree 這個演算法來解決圖形空間式的封包分類問題。封包到達時利用原本 R* -Tree 的演算法搜尋，找到的物件就是適合的規則，之後再從適合的規則中找出優先權最高的規則，就是符合的規則。R*-Tree 這個方法的優點在於只需要很小的儲存空間，但是它的比對速度就比較慢。

三、提出方法

(一) 基本想法

在前面的章節有提到，位元圖交集法有著快速的封包比對速度，並且它可以使用硬體平行處理的特性加速比對的速度。然而，它在規則數量多的情況下會有記憶體需求過高的缺點，它的空間複雜度是 $O(dN^2)$，其中 d 代表維度的個數，N 代表規則的數目。也就是說，當規則數量成長時，所需要的儲存空間會以二次方的方式往上成長。除此之外，規則數變多也就代表位元向量會隨之變長，這會導致比對時所需要的記憶體讀寫次數增加，進而讓比對速度變慢。

另外一種 R*-Tree 的演算法，在空間利用度上有著不錯的效能。當規則數量成長時，所需要的空間只會以線性的方式成長。然而在比對的速度上這個方法就比較慢了，主要的原因是因為 R*-Tree 在節點中找尋重疊的 MBR 時是以線性搜尋的方式來搜尋。舉例來講，考慮一個樹高為 4，維度為 5 的 R*-Tree，其中每個節點包含的 MBR 數量為 10 個，假設在理想狀況下，搜尋的動作必須要經過四個節點，總共需要 200 次的記憶體讀寫才能得到結果，所以在比對上的速度相對就比較慢了。

為了解決上面所提到的問題，我們提出一個結合這兩種方法的演算法，希望可以利用 R*-Tree 在空間利用度的優點去克服位元圖交集法的問題，以及利用位元圖交集法比對速度上的優勢來加速 R*-Tree 演算法。

我們的方法是基於 R*-Tree 封包分類法，把位元圖交集法使用在各個節點上。我們將 R*-Tree 中的每個節點想成一個獨立的圖形空間，裡面包含著對應此節點的子 MBR。將這些 MBR 當作規則，我們就可以在節點上使用位元圖交集法，這個方法可以用來替代原本 R*-Tree 演算法中搜尋重疊 MBR 的方法。此外，由於節點中所包含的 MBR 數量並不多，這表示位元圖交集法並不會使用到過多的記憶體空間，雖然整體 MBR 的數量會超過原本規則的數量，但是它們會被切分成較小的群體，每個群體使用極小的位元圖來代表，整體來看所需要的記憶體空間比起位元圖交集法來小得多。

(二) 實際作法

我們的方法基本上分成兩個部分，建立 R*-Tree 以及處理封包比對。第一部分主要是把所有的規則建立成一個 R*-Tree，而第二部分是利用這個建立好的 R*-Tree 來搜尋封包所對應的規則。

在建立 R*-Tree 這部分，我們會先把規則轉換成 MBR 的資料格式，其中包含規則在每個維度中對應到圖形空間的開始位置與結束位置以及規則的相關資訊(如優先權、比對後採取的動作)。之後利用原本 R*-Tree 的演算法把這些 MBR(即 規則)依序的插入到樹當中，直到規則被完全插入為止。接下來我們使用 Depth-First-Search 或是 Breadth -First-Search 的方式依序拜訪樹中的節點。每到一個節點時，就把 MBR 資訊轉換成位元圖交集法中所使用的位元向量，並且重新記錄到此節點之中。在拜訪完所有的節點後，建立 R*-Tree 的步驟就正式完成。圖 3.1 中顯示了我們方法對於圖 2.2 中的範例所建立的節點架構，我們使用位元圖交集法把圖 2.2 中四個 MBR 圖形資訊轉換成位元向量，並且

把這些位元向量紀錄在節點 B 中。

在處理封包比對這部分，我們把封包標頭中用來比對的欄位資訊取出，然後利用此資訊詢問我們所建立的 R*-Tree 來找出規則。一開始從樹的根節點開始，注意在這邊與 R*-Tree 方法不同的是，這邊是利用位元圖交集法來找出與封包所代表的詢問點重疊的 MBR。如圖 3.1，我們使用位元圖交集法快速地搜尋節點 B 中與詢問點重疊的 MBR，它的搜尋速度比起使用線性搜尋要快，所以我們能夠快速地得到結果。在結果位元向量中被設定為 1 的位元所對應的 MBR 就是我們要找的 MBR。找到所有重疊的 MBR 後，繼續拜訪這些 MBR 所對應的節點，重複以上動作直到拜訪到樹葉節點為止。在樹葉節點也是使用相同的方式找出重疊的 MBR，而這些 MBR 就會對應到適合的規則，其中優先權最高的就是最後的比對結果。

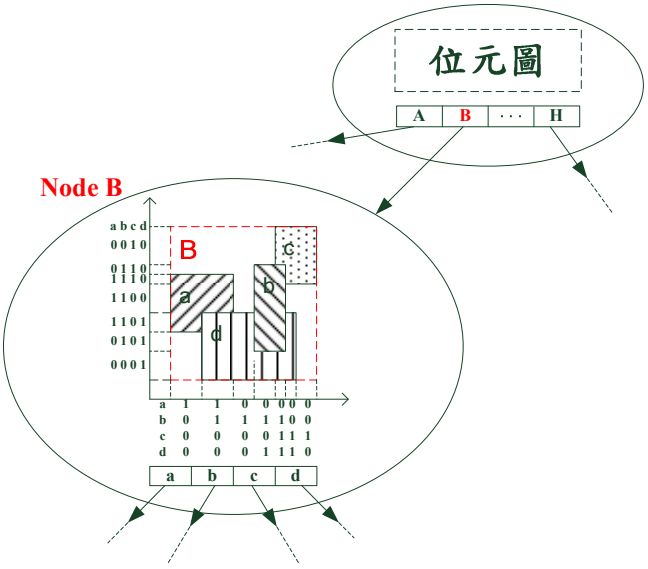

圖 3. 1 節點架構

(三) 效能分析

這邊我們將分析並比較我們的方法、位元圖交集演算法、以及 R*-Tree 封包分類演算法，在時間複雜度與空間複雜度上的差異。

在比對速度方面，我們所提出的演算法封包比對速度的複雜度如下：

$$O(\log N \times d \times (\log M + M/w)) \quad (1)$$

式子中的 d 表示維度的個數，N 表示規則的數目，w 表示記憶體通道的寬度，M 表示一個節點中所包含的 MBR 數量。做一次封包比對的過程中，必須要拜訪 $\log N$ 個節點(即 樹高)才能夠找到規則。並且每個節點中要花費 $\log M$ 的時間找到封包對應的區間以及花費 M/w 的時間取出位元向量，這兩個動作總共要做 d 次。其中 M 可以視為常數。而下面是位元圖交集法比對的複雜度：

$$O(d \times \log N + d \times N/w) \quad (2)$$

在這個方法中，要針對每個維度作出總共 d 次搜尋位元向量的動作，每一次都需要 $\log N$ 的時間找出區間的位置以及 N/W 的時間取出位元向量。而 R*-Tree 的複雜度如下：

$$O(\log N \times d \times M) \quad (3)$$

此方法中，同樣也必須拜訪 $\log N$ 個節點，每個節點中，搜尋重疊 MBR 的動作總共需要 $d \times M$ 的時間。可以看得出來我們的方法複雜度相對於另外兩個方法是比較好的。

在空間複雜度方面，我們提出的方法所的空間複雜度如下：

$$O((N/M) \times d \times M^2) \quad (4)$$

其中 N/M 為樹中節點個數，$d \times M^2$ 為節點內位元向量所需要的空間。而 R*-Tree 演算法的空間複雜度如下：

$$O((N/M) \times d \times M \times C) \quad (5)$$

在這式子中 N/M 也同樣表示節點數目，節點內紀

錄 MBR 資訊則需要 $d \times M \times C$ 的空間，C 代表每個維度中，儲存每個 MBR 的空間資訊所需記憶體的大小，以 source/ destination 位址為例，總共需要 64 位元儲存開始與結束的位置。在此 $M \times C$ 也同樣能夠視為常數，所以這兩種方法的空間複雜度相當。位元圖交集法的空間複雜度如下:

$$O(d \times N^2) \qquad (6)$$

相較於前兩種方法，此方法的空間複雜度比較差，當規則數量上升時，需要的記憶體空間會以二次方的方式成長。在規則數量大的時候，需要大量的記憶體空間，這會影響封包分類的效能。

四、實驗結果

本篇論文中我們使用 ClassBench [5]這個專門使用在封包分類問題上的工具來測試我們方法的效能。此工具包含規則產生器以及流量產生器兩部分，其中規則產生器可以產生出反映出真實情況的規則表，而流量產生器會產生用來測試分類方法的流量。我們我使用的平台規格如表 4.1 中所示，並且我們使用 C 來實作。

表 4. 1 平台規格

參數	數值
中央處理器	Intel C2D T7300
處理器時脈	2.00GHz
記憶體容量	2.00GB
作業系統	WindosVista

表 4. 2 實驗參數

參數	數值
規則數量	1k~10k
測試封包數量	100k
規則的維度	5
節點中 MBR 最大數量	32
節點中 MBR 最小數量	16
封包長度	40 Bytes

在這邊我們所用來測試的規則包含五個維度，分別對應到封包標頭中的五個欄位(即 Source/Destination address、 Source/ Destination Port、 Protocol)。在此章節中我們實驗四種不同的演算法，分別是位元圖交集法、R*-Tree 分類法、R*-Tree 與位元圖混合法、和 R-Tree 與位元圖與混合法。其中 R-Tree 為 R*-Tree 的前身，兩者在架構上非常相似，差別只是 R*-Tree 效能比較好，其他如節點的架構或是封包比對的方法都相同，這邊我們拿它與其他方法一起做比較。其他細部的模擬實驗參數列在表 4.2 中。

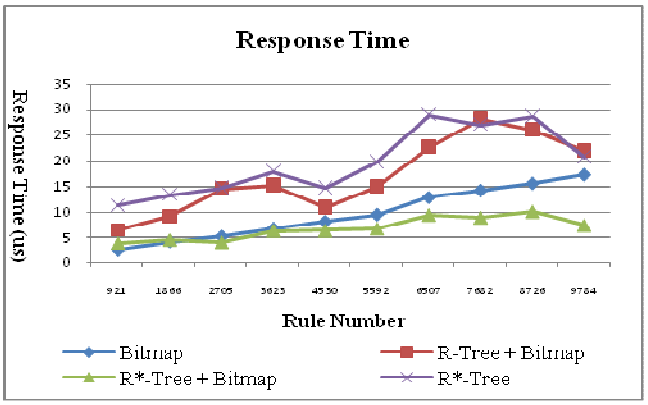

圖 4. 1 Response Time

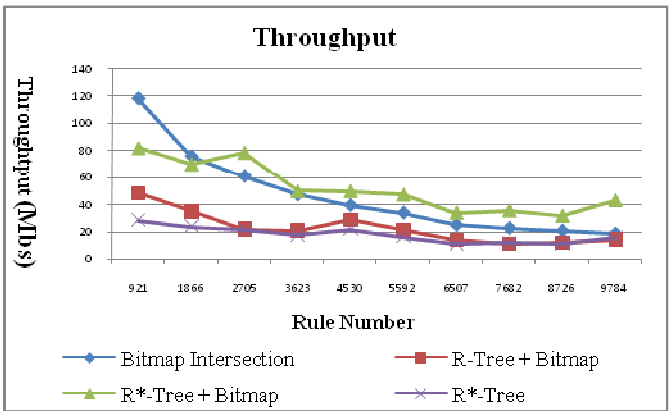

圖 4. 2 Throughput

首先我們測試封包比對的反應時間以及整體的輸出效能，這裡的實驗數據都是處理十萬個封包後，其結果平均所得到的數值。

圖 4.1 以及圖 4.2 顯示兩個實驗的結果，分別為反應時間以及輸出量，反應時間代表的是一次封包分類所需要的時間，而輸出量的計算方法

為反應時間的倒數乘上封包長度。從圖 4.1 我們可以觀察到在規則數量比較小的時候位元圖交集法的反應時間是最短的，而我們所提出的方法雖然比較差但是差距非常小。當規則的數量成長時，可以看得出來位元圖交集法的效能開始下降，原因是因為位元向量變長，存取位元向量所需要的記憶體存取次數變多所導致。反觀我們所提出的方法，雖然反應時間有所上升，但是比起位元圖交集法，我們方法的反應時間上升的幅度比較小，在規則數為一萬筆時，我們的方法效能大約是位元圖交集法的 2.3 倍，R*-Tree 的 2.8 倍。至於 R-Tree 與位元圖與混合法在規則數少的情況下，雖然效能比 R*-Tree 好，但是在規則數量大時，因為搜尋時需要走的路徑變多，導致兩者效能相當。另外我們可以觀察到一個有趣的現象，就是在我們的方法中會有規則數上升可是反映時間反而下降的情形發生。這種情況的發生是因為在我們的方法中影響封包分類效能的因素並不是規則的數目，而是被規則在圖形空間中分佈的情況所影響。若規則與規則間重疊的比率很高的話，就代表 R*-Tree 中 MBR 的重疊率也會上升，這會導致在執行封包比對時會多走一些多餘的路徑，進而使得封包比對時間變長，所以在這才會有這種情形發生。我們觀察圖 4.2 個實驗結果也可以看當相同的趨勢，在規則數量小時也是位元圖交集法效能較好，而規則數量大時則是我們方法效能較好。

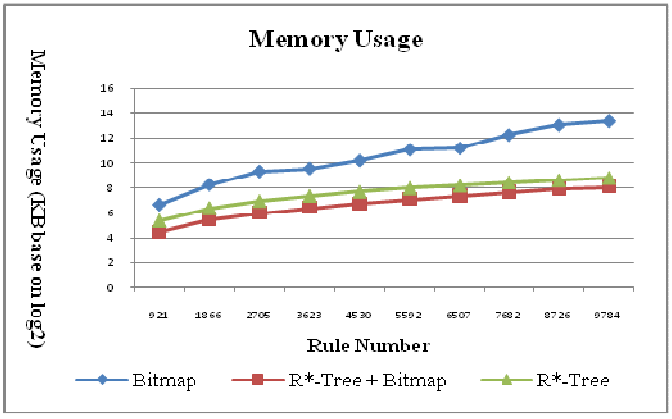

圖 4. 3 Memory Usage

圖 4.3 是記憶體使用量的實驗結果，由於結果差距比較大，所以這裡的結果取了一個 log 讓實驗數據方便觀察。可以很明顯的看得出來，我們的方法在記憶體的使用上比起位元圖交集法有效率的多。在規則數量為一萬筆的情況下，位元圖交集法約莫需要 10MB 的記憶體空間，R*-Tree 為 439KB，而我們的方法大約只需要 267KB 的記憶體，我們的方法需要的記憶體空間最少，至於 R-Tree 與位元圖混合法的記憶體需求與 R*-Tree 與位元圖混合法相當，所以在此就沒有列出。

綜合以上結果，雖然我們的方法在規則數量小時比對速度稍慢於位元圖交集法，但是整體來看當規則數量增加時我們的方法無論是在比對速度或是記憶體的使用上都優於其他演算法。

五、結論與未來展望

本篇論文中，我們提出了一個能夠同時解決位元圖交集法空間複雜度太高與 R*-Tree 封包分類法比對速度慢這兩個問題的演算法。主要的方法是透過結合以上兩種演算法，利用彼此的優點來克服缺點。我們利用 ClassBench 這套能反映現實情況的工具來實驗我們的方法。實驗結果指出，我們的方法無論是在比對速度或是空間複雜度上都展現出超越其他方法的效能。未來我們希望夠利用硬體平行處理的方式來加速我們的方法，例如節點內搜尋重疊 MBR 以及拜訪 R*-Tree 都能夠使用平行處理的方式來加速，這能夠提升我們的方法的效能。

六、參考文獻

[1] A. Cuttman, “R-Tree: A Dynamic Index Structure for Spatial Searching,” in proceedings of SIGMOD, pages 47-57, 1984.

[2] B. Seeger, H.-P. Kriegel, N. Beckmann, and R. Schneider, “The R*-tree: An Efficient and

Robust Access Method for Points and Rectangles," in proceedings of SIGMOD, pages 322-331, 1990.

[3] C. Maindorfer and T. Ottmann, "Is the Popular R*-Tree Suited for Packet Classification?," in proceedings of IEEE NCA, pages 168-176, 2008.

[4] D. Taylor, "Survey & Taxonomy of Packet Classification Techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pages 238–275, 2005.

[5] D.E. Taylor and J.S. Turner, "ClassBench: A Packet Classification Benchmark," in proceedings of IEEE INFOCOM, 2005.

[6] G. Varghese, "Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices," Morgan Kaufmann publishers, 2005.

[7] H. J. Chao and B. Liu, "High Performance Switches and Routers," Wiley-IEEE Press, 2007.

[8] M.H. Overmars and A.F. van der Stappen, "Range Searching and Point Location Among Fat Objects," *Journal of Algorithms*, vol. 21, no. 3, pages 629-656, November 1996.

[9] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pages 24-32, March/April 2001.

[10] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in proceedings of *IEEE Micro*, vol.20, no.1, pages 34-41, January /February 2000.

[11] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification using Multidimensional Cutting," in proceedings of SIGCOMM, pages 213-224, 2003.

[12] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," in proceedings of SIGCOMM, pages 191-202, 1998.