改良式 Bloom Filter 演算法及其效能分析
An Enhanced Bloom Filter Algorithm and It's Performance Analysis

張盛傑　李政宏[†]　呂紹偉[††]
國立台灣海洋大學
電機工程學系
{M92530069, [†]D93530006, [††]b0119}@mail.ntou.edu.tw

摘要

搜尋過濾器是一種能加速搜尋比對過程的特殊機制。由於能夠被應用在許多需要用到搜尋系統的場合，搜尋過濾器的研究在最近幾年相當受到重視。而單維度搜尋過濾器 Bloom Filter 就是第一個被提出來的搜尋過濾器並且持續作為其他搜尋過濾器發展的基礎，但是基於 Bloom Filter 的搜尋過濾器不僅會有誤判的發生，並且在搜尋目標進入過濾器後，搜尋整個鍵語的步驟耗費太多成本。本文提出了一套新的搜尋過濾器架構，此架構以每個鍵語的最小值做歸納區分，來建立一個查表目錄，讓新的搜尋過濾器只需以其鍵語最小值所分配的位置做查詢，來取代原本的搜尋過濾器，當誤判發生時對整個鍵語集合內所有鍵語做搜尋。而我們由實驗結果可以知道，新的搜尋過濾器架構，不僅能達到過濾動作之要求，且能降低誤判率的發生，大幅減少通過搜尋過濾器後需比對之鍵語的數量，降低整個過濾搜尋動作所須花費的搜尋成本，讓搜尋過濾器有更好的效能。

關鍵詞：搜尋過濾器，鍵語集合，誤判，查表目錄。

Abstract

A search filter is a special mechanism that is used to speed up the operation of a search system. It has received much attention in recent years as it is critical to the efficiency of applications involving the use of search engines. The well-known Bloom Filter is the first search filter to be proposed and has remained as an important foundation for the development of new search filters. However, Bloom Filter based filtering mechanisms have one major drawback, that is, after a query item passes through the filter the entire key set has to be searched even for false positives. The search cost is therefore bound to be higher than necessary. To cope with this problem we have developed in this article a new search filter mechanism which is free from unnecessary increases of search cost upon the presence of false positives. Our algorithm starts from building a lookup table with the minimum value of each search key. Instead of searching the entire key set as many traditional filters do when a false positive happens, the proposed search filter only inquires the location corresponding to the minimum value of the key. Our analyses show that the new search filter not only greatly lowers the possibility of false positives, but also significantly reduces the overall search cost, since a large amount of unnecessary search operations can be avoided.

Keywords: Search Filter, Key Set, False Positive, Lookup Table.

一、導論

由於資訊應用的普及，搜尋技術的研究重新受到廣泛的重視。著名的搜尋方法如Binary Search、Hashing、Game Tree Search等，都能讓搜尋過程更有效率，但是這些方法在搜尋時，若搜尋對象不在其鍵語集合(Key Set)內，則搜尋效率就會變的很差。因此在進入搜尋步驟之前，如果能夠先過濾不屬於鍵語集合的搜尋目標，便能大幅提昇搜尋系統的效率。搜尋過濾器(Search Filter) [1]就是因應此種需求而產生的。

Bloom Filter [2]是第一個被提出的搜尋過濾器，它所使用的核心方法為雜湊編碼(Hash Coding)[3]，其特點是能夠利用少量的儲存空間存放鍵語集合，然後利用雜湊函數的轉換求得所欲搜尋的對象。由於使用較少的儲存空間，誤判(False Positive)[2]的情形是無法避免的，而誤判率(False Rate)的高低將會影響搜尋過濾器的搜尋成本(Search Cost)，因此如何減少誤判率並且降低搜尋成本就成為重要的課題。本論文以Bloom Filter為基礎提出一個新的搜尋過濾器，此過濾器能夠縮小一般搜尋過濾器後續需搜尋比對的範圍，加速鍵語的擷取，因此能降低搜尋成本，提高搜尋過濾器的整體效能。

二、文獻回顧

在1970年代，電腦記憶體還是非常昂貴，因此在使用電腦處理資料處理時，必須仔細考量記憶體空間的需求。著名的Bloom Filter搜尋演算法就是

利用雜湊編碼的方法來減少記憶體的使用，並且能夠達到資料搜尋過濾的目的。1990年，根據Bloom Filter改良的搜尋過濾器Random Filter [4]被提出。上述兩種過濾器都是一維搜尋過濾器。接著在1992年及1995年，學者們又陸續發表三個二維搜尋過濾器，即RR Filter [5]、BB Filter [6]和BR Filter [6]。以下簡介這些搜尋過濾器及其基本原理。

首先介紹Bloom Filter搜尋過濾器。搜尋過濾器的運作機制有兩個要素，一是雜湊空間(Hash Space)，二是雜湊函數(Hash Function)[2]。雜湊空間主要是用來儲存經過雜湊函數轉換過的特徵值的記憶空間；而雜湊函數的功能是將鍵語集合內的鍵語轉換成鍵語值，然後再把鍵語值所對應的雜湊空間內的位置全都設定為1。假設有一大小為N位元的雜湊空間，其值全部設為0。欲存入之鍵語為K，經以雜湊函數$h_{b1}(K)$、$h_{b2}(K)$、$h_{b3}(K)$、$h_{b4}(K)$轉換後，將運算所得的特徵值在雜湊空間內所對應的位置全都設為1。當我們要測試新鍵語K'是否包含於鍵語集合內時，只需以雜湊函數轉換，然後以轉換之後的特徵值去比對雜湊空間中的位置，只要在比對的過程中發現任何一個位置上的位元為0，就表示此新鍵語不在此鍵語集合中；若是所比對的位置皆為1，則新鍵語就會被搜尋過濾器所接受。另一個單維度搜尋過濾器Random Filter的建制過程大部分都和Bloom Filter相同，而這兩種演算法的最大差異在於雜湊函數的碰撞。Bloom Filter在轉換鍵語集合時不允許特徵值的碰撞，而Random Filter則允許鍵語擁有相同的特徵值位置 [4]。如圖1所示，假設我們要為鍵語K'建置雜湊空間，雜湊函數為$h_1, h_2, ..., h_{bd}$，此例中Bloom Filter所轉換的每一個位置都不相同，而Random Filter在$h_{b1}(K')$、$h_{b2}(K')$兩個位置上發生碰撞。根據實驗統計，Random Filter在誤判率較Bloom Filter為低 [4]。

二維搜尋過濾器跟一維搜尋過濾器的最大差異在於雜湊空間的不同，二維搜尋過濾器使用二維的雜湊空間，並且使用兩個獨立的雜湊函數，而建構的方法大致上和一維搜尋過濾器相同 [5][6]。假設有一個二維搜尋過濾器，其雜湊空間為$N \times N$位元。當我們有一鍵語 K 要存入雜湊空間時，我們同樣利用雜湊函數求 K 之特徵值，然後把這些值轉換成相對的位置，再依序存入其雜湊空間，如此即完成二維搜尋過濾器之建置。新鍵語是否包含於鍵語集合內的測試方法也和一維搜尋過濾器完全相同。

RR Filter 搜尋過濾器是 Random and Random Filter 的縮寫，表示此二維搜尋過濾器使用的兩個雜湊函數都是允許碰撞發生的；依此類推，BB Filter 表示所使用的兩個雜湊函數不允許碰撞發生，而 BR Filter 就是其中一個雜湊函數允許碰撞，另一個雜湊函數不允許。在二維搜尋過濾器的效能方面，BB Filter 優於其餘二種搜尋過濾器[5][6]。

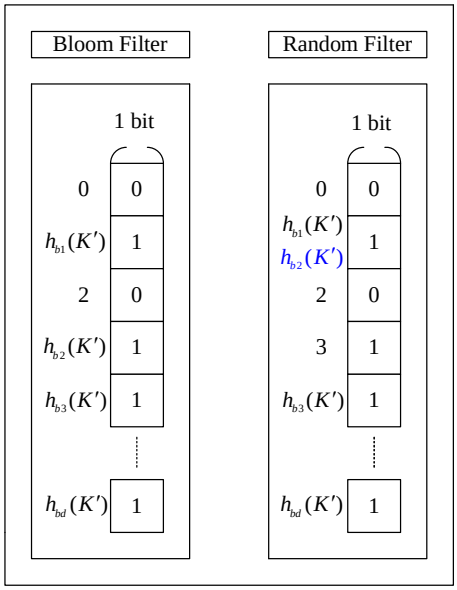

圖 1 Bloom Filter 和 Random Filter 雜湊空間的差異

三、改良式 Bloom Filter 搜尋過濾器

3.1 Bloom Filter 演算法[8]

考慮一個擁有 n 個元素的鍵語集合 $A = \{a_1, a_2, ..., a_n\}$。Bloom Filter 使用一含有 m 個一位元元數的向量 V，來描述 A 集合的成員資訊，並以 k 個雜湊函數 $h_1, h_2, ...h_k$ 來做運算，其中 $h_i : X \to \{1, 2, ..., m\}$。

Phase1: 建立過濾器
Procedure BloomFilter(set *A*, hash_functions, integer *m*, *V*) **returns** filter
    Initialize *V* to 0(m bits)
    foreach a_i in *A*:
        foreach hash function h_j:
            $V[h_j(a_i)] = 1$
        end foreach
    end foreach
    return filter

首先以雜湊函數 $h_1, h_2, ...h_k$ 建立一個對應到鍵語集合 A 且包含 m 個一位元元數的向量 V。如果 a_i 是集合 A 的成員之一，則 Bloom Filter 執行後相對應的 V 元素應該全部設為 1。

Phase2: 鍵語比對階段
Procedure MembershipTest (*elm*, filter, hash_functions, *V*) **returns** Yes/No
    foreach hash function h_j:
        if $V[h_j(elm)]$!= 1 **return** No
    end foreach
    return Yes

第二階段測試 *elm* 是否為鍵語集合 *A* 的成員之一。測試值 *elm* 進入搜尋過濾器做檢驗，如果 *elm* 的特徵值不符合就回傳 No，表示 *elm* 不屬於這個鍵語集合。如果符合就傳回 Yes，然後進入存取階段繼續做搜尋，直到找到符合的值為止，如果沒有找到則表示這是一個誤判。

3.2 改良式 Bloom Filter 演算法

我們提出了一個新的演算法，此演算法以**Bloom Filter**為基礎，在比對步驟中以鍵語的特徵來建立一查表目錄，此查表目錄是以每個鍵語特徵的最小值來做歸納區分，當誤判發生時，只需以其鍵語最小值所分配的位置做查詢並判別是否正確，因而減少搜尋時間與成本，並且提高搜尋過濾器的整體效能。以下是完整的演算法步驟說明：

考慮一含有 n 個元素的鍵語集合 $B=\{a_1,a_2,...,a_n\}$。我們使用一含有 m 個一位元元素的向量 V，來描述集合 B 的成員資訊，並使用 k 個雜湊函數 $h_1,h_2,...,h_k$，其中 $h_i: X \rightarrow \{1,2,...,m\}$。

Phase1: 建立過濾器[8]
Procedure Search Level(set *B*, hash_functions, integer *m*, *V*) **returns** filter
Initialize *V* to 0(m bits)
foreach b_i in *B*:
foreach hash function h_j:
$V[h_j(b_i)] = 1$
end foreach
end foreach
return filter

首先以k個雜湊函數 $h_1,h_2,...,h_k$ 為集合*B*建立一個含有*m*個一位元元數的Bloom Filter。

Phase2: 建立查表目錄 [9]
Procedure Access Level(set *B*, hash functions, *T*) **returns** *T*
foreach b_i in *B*:
foreach hash function h_j:
Allocate each **min[**$h_j(b_i)$**] into** *T* with all b_i's generating the same **min[**$h_j(b_i)$**]** put in the same entry.
end foreach
end foreach
return *T*

第二階段的目的是建置一個Search Mechanism，其輸出以符號*T*表示，*T*即為後來搜尋階段所用到的查表目錄，而Search Mechanism則是一個存放鍵語集合*B*的資料庫。在存取步驟中，以鍵語的特徵值來建立一查表目錄，此查表目錄是以每個鍵語特徵的最小值來做歸納區分，把擁有相同最小鍵語特徵值的成員排列在一起。設定查表目錄*T*的第一格為這些最小的鍵語值，以備查詢之用。

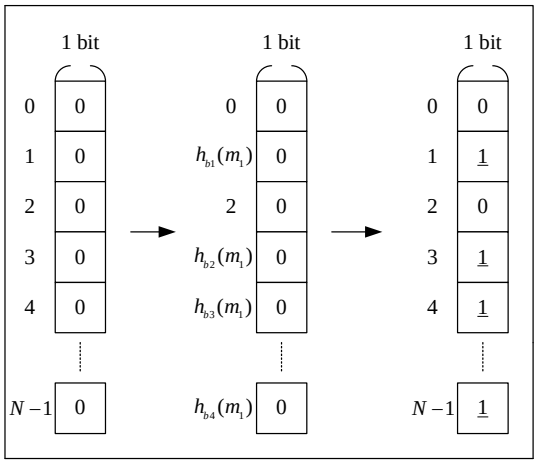

圖 2 新搜尋過濾器 Hash Space 的建制

Phase3: 新鍵語比對階段
Procedure MembershipTest(*elm*, filter, *T*, hash functions) **returns** Find/Error/No
foreach hash function h_j:
if $V[h_j(elm)]$!= 1 **return** No
if $V[h_j(elm)]$ = 1**then** search *T*
if min[$h_j(elm)$] !=*T* **return** Error
if min[$h_j(elm)$] =*T* **return** Find
end foreach

第三階段為測試 *elm* 是否為鍵語集合 *B* 的成員之一，測試值 *elm* 一開始先進入搜尋過濾器做檢驗，如果不符合就回傳 No，如果測試符合就進入第二步驟做比對。第二步驟一開始先找尋 *elm* 特徵值的最小值是否存在查表目錄中，如果沒有則回傳 Error 表示誤判發生，如果有則傳回 Find 表示 *elm* 為這個鍵語集合的其中一員。

3.3 新的搜尋過濾器技術

本節所提出的搜尋過濾器演算法，是以每個鍵語經雜湊運算後得到的最小值為依據，建立一個查表目錄，在搜尋過程中只需以其鍵語最小值所分配的位置做查詢，來判別測試值是否存在，不僅能達到過濾動作之基本要求且能大幅減少通過搜尋過濾器後需比對之鍵語的數量，降低整個過濾搜尋動作所須花費的搜尋成本，讓搜尋過濾器有更好的效能。接下來我們用一個圖例來說明。假設有一個鍵語集合 $m_1,m_2,...,m_n$，我們使用雜湊函數 h_{b1}、h_{b2}、h_{b3} 和 h_{b4} 來對鍵語集合做轉換，如圖 2 所示。New key file 是我們以鍵語集合所建構的查表目錄，其中每一個 entry 都紀錄鍵語的最小特徵值。由於 m_1 和 m_5 的最小特徵值都為 4，所以歸類在一起，其餘鍵語都只擁有一個最小的特徵值，因此在目錄中各自單獨佔用一列。現在有三個測試值 m_1'、m_2' 及 m_3' 要接受測試，我們已經知道 m_1' 的最小特徵值為

11，m_2' 是 8，m_3' 是 2，我們可以先比對查表目錄，在查表目錄裡只發現 11 這個值，所以 m_1' 會被接受，而 m_2' 及 m_3' 會在過濾階段前就直接被刪除。被接受之 m_1'，如果通過過濾階段，在後續搜尋動作時，直接進入 11 這個位置，對該位置所含的鍵語做搜尋比對動作，以判斷過濾結果是否為誤判情況，如圖 3 所示。

3.4 效能比較與分析

為了驗證 New Search Filter 的效能，我們與現存的兩種單維度搜尋過濾器進行誤判率和效能上的比較。實驗參數如下：

1. 選用五組雜湊函數。
2. 使用2048位元的Hash Space。
3. 每一個鍵語集合使用亂數產生器（Random Number Generator）產生1000個鍵語。
4. 每次實驗均執行100個不同對象之過濾搜尋，且該100個搜尋對象均不屬於鍵語集合。

實驗結果是以二種數據呈現，即誤判率和搜尋成本。誤判率之計算是依下式 [10]：

$$\frac{\text{通過過濾器之總次數}}{100} \times 100\%$$

而搜尋成本則為

$$\frac{\text{所需搜尋的鍵語次數}}{\text{鍵語總數量}} \times 100\%$$

我們以三個過濾器測試 10 組不同的搜尋對象，每一組均測試 100 個值，且三個過濾器同一組的測試值皆為相同。圖 4-6 分別是三個過濾器的誤判率長條圖，而圖 7 是條據比較。三個過濾器的平均誤判

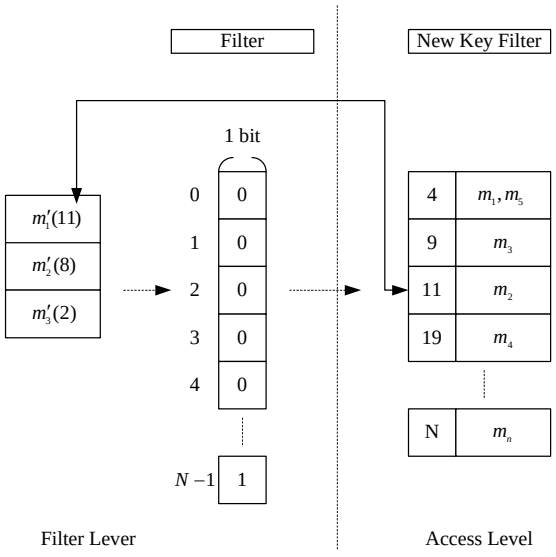

圖 3 新搜尋過濾器的架構

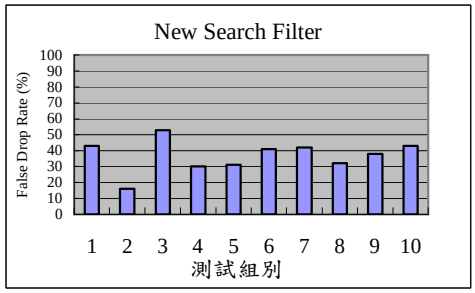

圖 4 New Search Filter 誤判率的測試結果

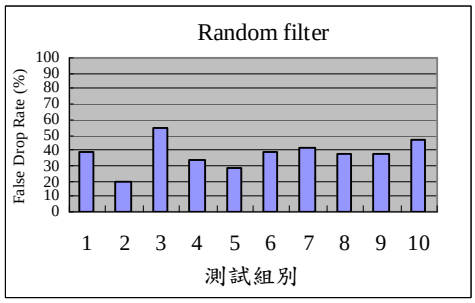

圖 5 Random Filter 誤判率的測試結果

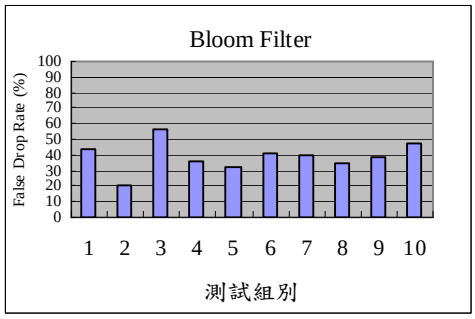

圖 6 Bloom Filter 誤判率的測試結果

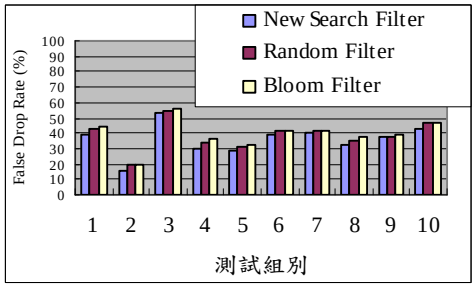

圖 7 三種過濾器誤判率的比較

值列於表 1 中。從表中可以看出，新的搜尋過濾器誤判率大約比 Random Filter 小 3%，而比 Bloom Filter 小 4%。由此可知，我們所提的演算法的確能夠降低誤判的發生，進而減少誤判發生的比率。

表1 三種過濾器的平均誤判值

組別	New Search Filter	Random Filter	Bloom Filter
1	39 %	43 %	44 %
2	16 %	19 %	20 %
3	53 %	55 %	56 %
4	30 %	34 %	36 %
5	28 %	31 %	32 %
6	39 %	41 %	41 %
7	40 %	42 %	42 %
8	32 %	35 %	38 %
9	38 %	38 %	39 %
10	43 %	47 %	47 %
Average	35.8 %	38.5 %	39.5 %

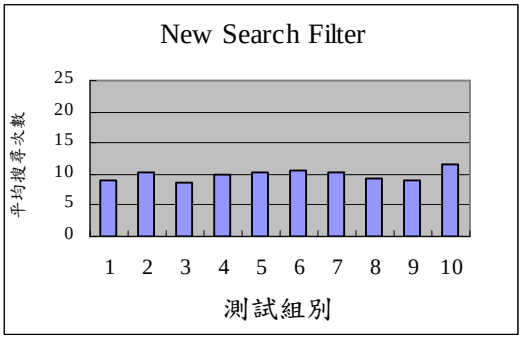

圖 8 新搜尋過濾器的平均搜尋次數

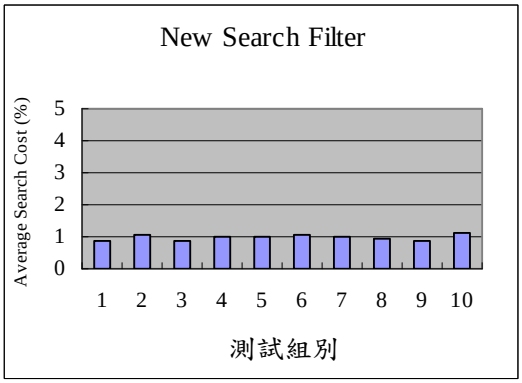

圖 9 新搜尋過濾器的平均搜尋成本

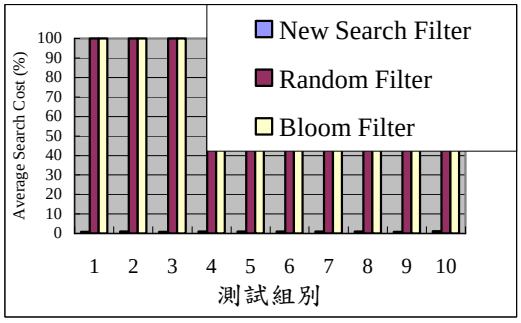

圖 10 三種過濾器平均搜尋成本比較

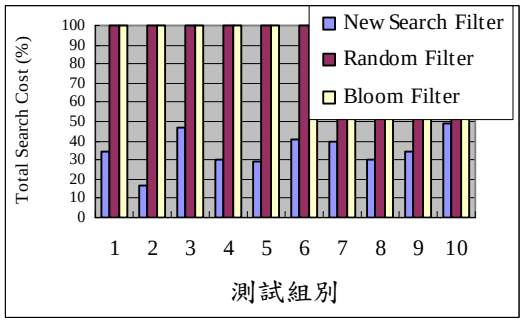

圖 11 總搜尋成本比較

在搜尋成本的測試方面，圖 8 為新搜尋過濾器測試 10 組不同的搜尋對象，每一組均測試 100 個值而得的。圖 9 是新搜尋過濾器的平均搜尋成本，由圖可以知道新搜尋過濾器的搜尋成本幾乎全部低於百分之一。圖 10 為新搜尋過濾器跟另外兩種搜尋過濾器的比較，當 Random Filter 及 Bloom Filter 接受搜尋對象後，均需搜尋整個鍵語集合才能確定是否為誤判，所以搜尋成本均為 100%。

針對這次實驗，我們可以計算出每一種搜尋過濾器因誤判所需付出的總搜尋成本為其誤判率×平均搜尋成本。最後結果如圖11所示，可以看出New Search Filter對於整個過濾搜尋過程中所需付出的搜尋成本最少，換句話說New Search Filter的過濾搜尋效能優於其他兩種搜尋過濾器甚多。

四、結論

本文提出一個新的搜尋過濾器架構，此架構以每個鍵語的最小值為依據，建立一個查表目錄，搜尋過濾器只需對其鍵語最小值所分配的位置查詢，而無須如傳統搜尋過濾器，在誤判發生時仍須對整個鍵語集合內的所有鍵語做搜尋。由模擬驗證結果可知，新的搜尋過濾器架構不僅能達到過濾動作的要求，且能降低誤判率，大幅減少通過搜尋過濾器後需比對之鍵語的數量，降低整個過濾搜尋動作所須花費的搜尋成本，讓搜尋過濾器有更好的效能。未來希望能將此架構應用於實際的搜尋系統上，並配合其他搜尋相關技術，提升搜尋系統的整體效能。

五、參考文獻

[1] D. G. Severance, G. M. Lohman, “Differential files: their application to the maintenance of large databases,” *ACM Transactions on Database Systems*, vol. 1, no. 3, pp. 256-267, 1976.

[2] B. H. Bloom, “Space time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 420-426, 1970.

[3] T. H. Cormen, C. Stein, R. L. Rivest, and C. E.

Leiserson, *Introduction to Algorithms*. McGraw-Hill, 2001.

[4] C. Y. Wang, W. P. Yang, J. C. R. Tseng, and M. Hsu, "Random filter and its analysis," *International Journal of Computer Mathematics*, vol. 33, pp. 181-194, 1990.

[5] J. C. R. Tesng and W. P. Yang, "2D random filter and analysis," *International Journal of Computer Mathematics*, Vol. 42, pp 33-45, 1992.

[6] C. C. Chang and J. J. Leu, "Two 2D Search filters and their performance analyses," *International Journal of Computer Mathematics*, vol. 60, pp. 183-203, 1995.

[7] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," in *Proceedings of the 11th Symposium on High Performance Interconnects*, pp. 44-51, Stanford, CA, Aug. 2003.

[8] Matei Ripeanu and Adriana Iamnitchi, "Bloom filters – short tutorial," <people.cs.uchicago.edu/~matei/PAPERS/bf.doc>

[9] V. M. Malhotra, B. Srinivasan, and S. Kulkarni, "Storage-efficient data structure for large lookup dictionaries," *Information Processing Letters*, vol. 58, pp. 201-206, 1996.

[10] 朱麒仲,"植基於吸收器之篩檢搜尋技術研究," 朝陽科技大學資訊管理系碩士學位論文, July 2002.