Theory and Practice of Science and Technology
2025, VOL. 6, NO. 4, 158-163
DOI: 10.47297/taposatWSP2633-456928.20250604

Analysis and comparison of search algorithms

Haomin Zhao[1]*, Dan Xu[2]

[1]Modern Information Industry Department, Guangzhou College of Commerce, Guangzhou, China

[2]Department of Information Technology and Engineering, Guangzhou College of Commerce, Guangzhou, China

ABSTRACT

This paper deeply studies the sequential search, binary search, hash search, binary search tree, balanced binary search tree (AVL tree and red-black tree), B tree and B+ tree in the search algorithm, and implements and tests their performance on different data sets using Python programming language. By analyzing the time complexity and space complexity, their advantages and disadvantages in practical applications are evaluated. The experimental results show that for static data sets, binary search and hash search are efficient choices; for dynamic data sets, balanced tree structures such as red-black tree and AVL tree are more suitable. Choosing a suitable algorithm can significantly improve the search efficiency. The research results of this paper provide a theoretical basis and practical guidance for the selection of search algorithms in practical applications.

KEYWORDS

search algorithm; time complexity; space complexity; balanced binary tree

1 Introduction

Searching is one of the most basic and commonly used operations in computer science. Efficient search algorithms are crucial to improving program performance. With the continuous expansion of data scale and the increasing complexity of data structures, various search algorithms have emerged, each of which has its specific applicable scenarios and advantages and disadvantages. This paper aims to systematically analyze and compare common search algorithms to provide a reference for algorithm selection in practical applications.

The study of search algorithms began in the early stages of computer science. From the simplest sequential search to efficient hash search, from basic binary search to complex balanced tree structures, the development of search algorithms reflects the unremitting pursuit of efficiency by computer scientists. In recent years, with the rise of big data and distributed systems, search structures such as B-tree and B+ tree suitable for external storage and databases have also received widespread attention[1].

This paper first introduces the basic concepts and evaluation criteria of search algorithms, then analyzes the principles and characteristics of various search algorithms respectively, and finally compares their performance through experiments, and gives suggestions for algorithm selection.

2 Overview and classification of search algorithms

Search algorithms are the process of locating target elements in a specific data structure. According to the different characteristics of data structures and application requirements, search algorithms can be divided into two categories: static search and dynamic search. Static search is suitable for situations where the data set does not change, while dynamic search needs to handle frequent insertion and deletion operations[2].

The main indicators for evaluating search algorithms include time complexity, space complexity, and implementation difficulty. Time complexity measures the relationship between the time required for algorithm execution and the size of the problem, usually expressed in big O notation. Space complexity refers to the additional storage space required during the execution of the algorithm. Implementation difficulty is related to the maintainability and scalability of the algorithm.

The selection of search algorithms needs to consider many factors, including data size, data distribution, memory limitations, and the frequency of search, insertion, and deletion operations. For example, for small static data sets, sequential search may be efficient enough; for large dynamic data sets, more complex balanced tree structures or hash tables are required.

2.1 Common Search Algorithms

Search algorithms are basic operations used in computer science to locate specific elements in a data set. The most common search algorithms include sequential search and binary search. Sequential search is the simplest search method, which starts from the first element of the data set and compares one by one until the target element is found or the complete set is traversed. This method is suitable for unordered data sets, but it is less efficient and has a time complexity

of O(n). As shown in Figure 1, a function named linear_search is defined to implement the sequential search algorithm. Linear search is a simple search algorithm used to find a specific target value in a list (or array)[3].

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # 返回索引
    return -1
```

Figure 1 Implementation of sequential search algorithm

Binary search is a more efficient algorithm that is suitable for sorted data sets. It quickly locates the target element by repeatedly splitting the search range in half. The time complexity of binary search is O(log n), which is much better than sequential search. However, binary search requires the data set to be ordered, which may add additional sorting overhead in some cases. As shown in Figure 2, a function named binary_search is defined to implement the binary search algorithm. Binary search is an efficient algorithm for finding a specific element in an ordered array.

```
def binary_search(arr, target):
    left, right = 0, len(arr)-1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Figure 2 Implementation of the binary search algorithm

In addition to these two basic algorithms, there are more advanced search methods, such as interpolation search and Fibonacci search. Interpolation search optimizes the search process by estimating the location of the target value and performs well on uniformly distributed data sets. Fibonacci search uses the Fibonacci sequence to divide the search range and is suitable for systems that cannot perform multiplication or division operations. These algorithms have their own advantages in different scenarios and provide more efficient solutions to specific problems[4].

2.2 Hash-based search algorithm

Hash search is a fast search method based on hash tables. Hash tables use hash functions to map keys to specific locations in the table, thereby achieving fast access. An ideal hash function should evenly distribute keys and minimize conflicts. Common hash functions include division hash, multiplication hash, and global hash. As shown in Figure 3, a simple hash table (HashTable) class is defined to implement basic hash search functions. A hash table is a data structure that converts input (such as a string or a number) into an index value through a hash function, and uses the index value as a key to access the corresponding position in the table.

```
class HashTable:
    def __init__(self, size=1000):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):  2 用法
        return key % self.size

    def insert(self, key):
        idx = self._hash(key)
        self.table[idx].append(key)

    def search(self, key):
        idx = self._hash(key)
        return key in self.table[idx]
```

Figure 3 Hash search algorithm

Handling hash collisions is a key issue in hash search. The main solutions are the chain address method and the open address method. The chain address method stores the conflicting elements in a linked list, while the open address method searches for the next available position by probing the sequence. Each method has its advantages and

disadvantages: the chain address method is simple but may produce a long linked list, and the open address method has high space utilization but is prone to clustering.

The average time complexity of hash search is O(1), but it may degenerate to O(n) in the worst case. Its performance depends largely on the quality and load factor of the hash function. Choosing a suitable hash function and conflict resolution strategy is crucial to optimizing hash search performance. In practical applications, hash search is often used to implement fast dictionaries, caches, and database indexes[5].

2.3 Tree-structured search algorithm

Tree structure provides another idea for efficient search. Binary search tree (BST) is a basic structure in which the left subtree of each node contains values less than the node and the right subtree contains values greater than the node. The time complexity of BST search, insertion, and deletion operations is O(h), where h is the tree height. However, ordinary BST may degenerate into a linked list, resulting in performance degradation. As shown in Figure x-x, a simple binary search tree is implemented, where the TreeNode class is used to create tree nodes and the BST class implements the function of searching for a specific value in the tree. The characteristic of a binary search tree is that for any node, the values of all nodes in its left subtree are less than the value of the node, and the values of all nodes in its right subtree are greater than the value of the node. This structure allows search, insertion, and deletion operations to be completed in logarithmic time.

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

class BST:
    def search(self, root, target):  2 用法
        if not root: return False
        if root.val == target: return True
        elif target < root.val: return self.search(root.left, target)
        else: return self.search(root.right, target)
```

Figure 4 Binary search tree algorithm

To overcome this problem, balanced binary trees came into being. AVL trees and red-black trees are two common balanced binary search trees. AVL trees keep the tree height to a minimum through strict balance conditions, providing optimal search performance, but the cost of maintaining balance is high. Red-black trees relax the balance conditions and achieve approximate balance through color marking and rotation operations, which are more advantageous in insertion and deletion operations. As shown in Figure 5, some basic operations of AVL trees are shown, including obtaining node heights, calculating balance factors, performing right rotation operations, and searching operations. These operations are the key to maintaining balance and efficiency of AVL trees.

```
class AVLTree:
    def _get_height(self, node):  6 用法
        return node.height if node else 0

    def _balance_factor(self, node):
        return self._get_height(node.left) - self._get_height(node.right)

    def _rotate_right(self, z):
        # 右旋操作（完整实现需补充左旋和平衡调整逻辑）
        y = z.left
        z.left = y.right
        y.right = z
        z.height = 1 + max(self._get_height(z.left), self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left), self._get_height(y.right))
        return y

    def search(self, root, target):  2 用法
        # 搜索逻辑与BST相同，但树始终保持平衡
        if not root: return False
        if root.val == target:
            return True
        elif target < root.val:
            return self.search(root.left, target)
        else:
            return self.search(root.right, target)
```

Figure 5 Search algorithm of AVL tree

B-tree and B+-tree are multi-way balanced search trees designed for disk storage. They reduce the tree height by increasing the branching factor of each node, thereby minimizing disk I/O operations. B+-tree is optimized on the basis of B-tree, storing all data in leaf nodes, which is more suitable for range query and sequential access. These tree structures are widely used in databases and file systems, providing strong support for the efficient management of large-scale data. As shown in Figure 6, a simple B-tree is implemented, in which the BTreeNode class is used to create the nodes of the tree,

and the BTree class implements the function of searching for specific key values in the tree. B-tree is a self-balancing tree data structure that can keep data in order and support efficient insertion, deletion and search operations. The characteristic of B-tree is that each node can have multiple child nodes, which makes it particularly useful in disk I/O intensive applications because it can reduce the number of disk accesses[6].

```
class BTreeNode:
    def __init__(self):
        self.keys = []
        self.children = []

class BTree:
    def search(self, node, key):  1 个用法
        i = 0
        while i < len(node.keys) and key > node.keys[i]:
            i += 1
        if i < len(node.keys) and key == node.keys[i]:
            return True
        elif not node.children: return False
        else: return self.search(node.children[i], key)
```

Figure 6 B-tree search algorithm

3 Comparison and analysis of search algorithms

3.1 Experimental design

Eight common search algorithms were selected in the experiment: Sequential, Binary, Hash, BST, AVL, RBTree, BTree, and B+Tree. These algorithms were selected because they represent different categories and characteristics of search algorithms, including unordered and ordered search, exact search, and range search. Small, medium, and large data sets with data sizes of 10^4, 10^5, and 10^6 were selected. The performance changes of the algorithms under different data sizes were observed to comprehensively evaluate the applicability and efficiency of the algorithms.

Experimental environment: Using the python programming language, the search algorithm's search time and memory usage performance were tested by randomly generating integer sets of 10^4, 10^5, and 10^6. The test environment is Intel i5-1245U, 32GB DDR4.

The experimental steps are as follows:

Prepare three data sets of different sizes for each algorithm.

Apply the corresponding search algorithm to each data set and record the execution time of the search operation.

Record the memory usage of each algorithm when processing each data set.

Repeat the experiment several times to ensure the accuracy and reliability of the results.

Collect and organize data for statistical analysis.

The experimental performance indicators include search time (in milliseconds) and memory usage (in MB). The search time reflects the efficiency of the algorithm in the search operation, while the memory usage reflects the algorithm's demand for system resources during execution.

3.2 Experimental Results

查找算法性能对比表：

Algorithm	10^4 (ms)	10^5 (ms)	10^6 (ms)	Memory (MB)
Sequential	0.15	1.42	14.70	0.8
Binary	0.03	0.04	0.05	1.2
Hash	0.01	0.01	0.01	25.6
BST	0.04	0.05	0.06	12.3
AVL	0.02	0.03	0.04	16.8
RBTree	0.02	0.03	0.04	18.4
BTree	0.03	0.04	0.05	14.2
B+Tree	0.03	0.04	0.05	15.7

Figure 7 Search algorithm performance comparison table

The experimental results are shown in Figure 7. By testing the performance of eight search algorithms, including sequential search, binary search, hash search, binary search tree (BST), AVL tree, red-black tree, B tree and B+ tree, we obtained comprehensive performance comparison data. In terms of time performance, hash search performs best, maintaining a stable search time of 0.01ms in the data scale of 10^4 to 10^6, fully demonstrating its O(1) time complexity

advantage; the search time of balanced tree structure (AVL tree, red-black tree, B tree, B+ tree) and binary search is in the range of 0.02-0.05ms, which is in line with the theoretical complexity of O(log n), among which red-black tree and AVL tree perform best; sequential search shows a linear growth with the expansion of data scale, from 0.15ms for 10^4 data to 14.7ms for 10^6 data, an increase of 100 times, which is consistent with the growth multiple of data scale, verifying its O (n) time complexity. In terms of memory usage, sequential search and binary search only require 0.8-1.2MB of memory, which is the most efficient; hash tables require pre-allocated hash buckets and require 25.6MB of memory; among various tree structures, balanced trees (AVL, red-black trees) require 16.8-18.4MB of memory because they need to store pointers and balance information, and B-trees (BTree, B+Tree) control memory usage to 14.2-15.7MB by aggregating data at nodes. Experimental results show that there is a significant time-memory trade-off: hash tables trade high memory for optimal search speed, which is suitable for scenarios with sufficient memory; balanced trees achieve performance close to O(1) with lower memory usage, which is suitable for dynamic data operations; when the data size increases from 10^4 to 10^6, the sequential search time increases by 100 times, while the tree/binary search time only increases by 1.5-2 times, which verifies the superiority of logarithmic complexity.

Based on the experimental results, the algorithm selection recommendations for different application scenarios are as follows: binary search is recommended for static large data sets, which has high memory efficiency and does not require dynamic maintenance; red-black trees are suitable for high-frequency dynamic operation scenarios, with the best overall performance; hash tables can be selected when extreme query speed is required and high memory consumption is acceptable; and B+ trees are recommended for disk storage or database index scenarios, because of their high node utilization, which is particularly suitable for external memory environments. This experiment ensures the reliability of the results through the control variable method and multiple measurements, providing data support and theoretical basis for algorithm selection in actual engineering.

Table 1 Multi-dimensional comparison of search algorithms

Algorithm Type	Average time complexity	Space complexity	Data requirements	Dynamic data support	Range query support	Implementation complexity	Typical application scenarios
Sequential search	O(n)	O(1)	Unordered	Yes	No	Low	Small-scale unordered data
Binary search	O(log n)	O(1)	Ordered	No	No	Medium	Static ordered data
Hash search	O(1)	O(n)	None	Yes	No	Medium	Precise search (dictionary, cache)
Binary search tree	O(log n)	O(n)	None	Yes	Partial support	Medium	Dynamic data but no strict balance required
AVL tree	O(log n)	O(n)	None	Yes	Partial support	High	High-frequency search, low-frequency update scenarios
Red-black tree	O(log n)	O(n)	None	Yes	Partial support	High	General dynamic data (such as Java TreeMap)
B tree	O(log n)	O(n)	None	Yes	Partial support	High	File system, database index
B+ tree	O(log n)	O(n)	None	Yes	Support	High	Database range query (such as MySQL index)

Table 1 compares eight common search algorithms in detail, including sequential search, binary search, hash search, binary search tree, AVL tree, red-black tree, B-tree, and B+ tree. The table analyzes these algorithms from multiple dimensions, including average time complexity, space complexity, data requirements, dynamic data support, range query support, implementation complexity, and typical application scenarios. Average time complexity uses greater than O to represent the average time consumed by the algorithm during search, such as O(n) for sequential search and O(log n) for binary search. Space complexity uses greater than O to represent the additional space required during the execution of the algorithm. Data requirements indicate the algorithm's need for data sorting, such as binary search and AVL tree require ordered data, while hash search does not. Dynamic data support describes whether the algorithm supports inserting or deleting data during the search process. Range query support refers to whether the algorithm can perform range queries. Implementation complexity evaluates the difficulty of implementing the algorithm from low to high.

Typical application scenarios provide the most suitable usage environment for each algorithm, such as sequential search is suitable for small-scale unordered data, binary search is suitable for static ordered data, hash search is suitable for precise search scenarios, AVL tree is suitable for high-frequency search and low-frequency update scenarios, and B-tree and B+ tree are often used in file systems and database indexes, etc. This comparison table provides us with a clear perspective to help us choose the most suitable search algorithm according to specific needs.

4 Conclusion

This study systematically analyzes and compares various search algorithms and reveals their performance characteristics in different scenarios. The results show that no algorithm is optimal in all cases, and the selection of an appropriate search algorithm requires comprehensive consideration of factors such as data characteristics, operation types, and system limitations.

For static data sets, binary search and hash search are efficient choices; for dynamic data sets, balanced tree structures such as red-black trees and AVL trees are more suitable; for large-scale data storage, B-trees and B+ trees can provide more efficient solutions. Future research can further explore new data structures, such as skip lists, cuckoo hash algorithms, etc., as well as their performance optimization in specific application scenarios.

About the Author

* Corresponding Author: Haomin Zhao, 20230218@gcc.edu.cn;
Dan Xu, 20220063@gcc.edu.cn

References

[1] A spatially explicit evolutionary algorithm for the spatial partitioning problem[J]. Yan Y. Liu;; Wendy K. Tam Cho. Applied Soft Computing Journal.2020.

[2] Overview of Big Data Related Analysis [J]. Liang Jiye; Feng Chenjiao; Song Peng Journal of Computer Science, 2016 (01).

[3] Zhen Chao, Di Haitao, Zhao Yimin, etc Research and Implementation of Multi Index Binary Search Method [J]. Electronic Testing, 2020, (07):83-84+97..2020.07.031.

[4] Zhang Hebing Research on the Application Strategy of Binary Search Algorithm [J]. Fujian Computer, 2018, 34 (08): 116-117. DOI: 10.16707/j.cnki. fjpc. 2018.08.056.

[5] SAL-Hashing: A Self-Adaptive Linear Hashing Index for SSDs[J]. Jin Peiquan; Yang Chengcheng; Wang Xiaoliang; Yue Lihua; Zhang Dezhi.IEEE Transactions on Knowledge and Data Engineering.2020.

[6] Research on Large Database Retrieval Based on Improved B+Tree Index [J]. Xiao Hanzhou; Liu Yingchun Journal of Guilin Aerospace Industry College. 2024 (03).