Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Süleyman YOLCU

Student Number:
b2210765016

1 Problem Definition

- The problem addressed in this coding assignment revolves around analyzing the performance of various sorting and searching algorithms. The goal is to conduct experiments to compare the efficiency of these algorithms under different scenarios.

- In the context of sorting algorithms, the assignment aims to investigate how well algorithms such as Insertion Sort, Merge Sort, and Counting Sort perform when sorting random data, already sorted data, and reverse-sorted data. This involves measuring the execution times of these algorithms for different input sizes and types of data arrangements.

- Similarly, for searching algorithms, the assignment aims to evaluate the performance of Linear Search and Binary Search algorithms. The focus is on comparing their execution times when searching for elements within random data and sorted data.

2 Solution Implementation

To address the problem outlined in the previous section, a Java implementation is provided utilizing various sorting and searching algorithms. Below is an overview of the solution implementation:

2.1 Insertion Sort

This algorithm iterates through the array, repeatedly moving elements that are out of order into their correct position.

```java
public static void insertionSort(int[] array) {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j - 1;
            while (i >= 0 && array[i] > key) {
                array[i + 1] = array[i];
                i = i - 1;
            }
            array[i + 1] = key;
        }
    }
```

2.2 Merge Sort

It is a divide-and-conquer algorithm that recursively divides the array into halves, sorts the halves, and then merges them back together.

```
public static void mergeSort(int[] array) {
        int n = array.length;
        if (n <= 1) {
            return;
        }
        int mid = n / 2;
        int[] left = Arrays.copyOfRange(array, 0, mid);
        int[] right = Arrays.copyOfRange(array, mid, n);
        mergeSort(left);
        mergeSort(right);
        merge(array, left, right);
    }



private static void merge(int[] array, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                array[k] = left[i];
                i++;
            } else {
                array[k] = right[j];
                j++;
            }
            k++;
        }
        while (i < left.length) {
            array[k] = left[i];
            i++;
            k++;
        }
        while (j < right.length) {
            array[k] = right[j];
            j++;
            k++;
        }
    }
```

2.3 Counting Sort

This algorithm counts the number of occurrences of each distinct element in the array and uses this information to place the elements in sorted order.

```java
public static int[] countingSort(int[] array, int k) {
        int[] count = new int[k + 1];
        int[] output = new int[array.length];
        for (int j : array) {
            count[j]++;
        }
        for (int i = 1; i <= k; i++) {
            count[i] += count[i - 1];
        }
        for (int i = array.length - 1; i >= 0; i--) {
            int j = array[i];
            count[j]--;
            output[count[j]] = array[i];
        }
        return output;
    }
```

2.4 Linear Search

This algorithm sequentially checks each element of the array until it finds the target value or reaches the end of the array.

```java
public static int linearSearch(int[] array, int x) {
        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] == x) {
                return i;
            }
        }
        return -1;
    }
```

2.5 Binary Search

It requires the array to be sorted. It repeatedly divides the search interval in half until the target value is found (or not).

```java
public static int binarySearch(int[] array, int x) {
        int low = 0 ,  high = array.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (array[mid] == x) {
                return mid;
            } else if (array[mid] < x) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
        return -1;
    }
```

3 Results, Analysis, Discussion

3.1 Results

Insertion Sort:

- Best Case: $\Omega(n)$ The input data is already sorted.Average Case: $\Theta(n^2)$ comparisons and swaps.Worst Case: $O(n^2)$ comparisons and swaps when the input is in reverse sorted order.

Merge Sort:

- Best, Average, and Worst Case: Always $O(n \log n)$, regardless of input data arrangement. Due to its divide-and-conquer nature, it consistently performs efficiently.

Counting Sort:

- Best, Average, and Worst Case: $O(n + k)$, where **k** is the range of input values. This makes it highly efficient for inputs with a limited range.

Linear Search:

- Best Case: $\Omega(1)$ when the target element is found at the beginning of the array.
- Average and Worst Case: $O(n)$, as it must traverse the entire array in the worst case.

Binary Search

- Best Case: $\Omega(1)$ when the target element is found at the first iteration.
- Average and Worst Case: $O(\log n)$ when the target element is found at half of the iterations or last iteration.

Computational complexity comparison of the given algorithms are given in Table 3

3.2 Analysis

The obtained results generally match the theoretical asymptotic complexities of the algorithms. For sorting algorithms, Merge Sort consistently demonstrates $O(n \log n)$ performance across all data arrangements, aligning with its theoretical complexity.Figure3 Counting Sort exhibits $O(n + k)$ performance, showing stability across different data arrangements and input sizes, as expected. (Since I pass the data sorted beforehand for sorted experiments and for the input size 250k the k value is 119999780 which is significantly higher than others we observe a peak at the last point Figure2) Insertion Sort's performance matches its $O(n^2)$ complexity, with noticeable differences in execution times between best, average, and worst-case scenarios.Figure1 Linear Search's $O(n)$complexity is evident in its increasing execution times with larger input sizes, while Binary Search's $O(n \log n)$ complexity results in faster performance for sorted data.Figure4

Table 1: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion sort	0.4	0.2	0.2	0.8	3.0	15.0	48.3	175.8	667.3	2895.1
Merge sort	0.1	0.1	0.2	0.2	0.6	1.9	2.2	5.1	11.3	24.1
Counting sort	147.2	103.7	101.2	98.8	93.1	92.6	97.1	115.0	104.0	103.3
	Sorted Input Data Timing Results in ms									
Insertion sort	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.1	0.2	0.3
Merge sort	0.0	0.1	0.1	0.2	0.4	0.8	1.8	3.0	4.0	10.1
Counting sort	0.0	0.0	0.0	0.1	0.0	0.1	0.1	0.1	0.7	106.2
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0.1	0.1	0.7	1.4	5.1	19.3	78.4	314.8	2407.4	10218.7
Merge sort	0.0	0.1	0.1	0.2	0.5	0.9	1.8	4.1	8.1	17.7
Counting sort	112.0	101.6	99.8	103.5	105.5	106.4	101.7	104.1	104.7	106.5

Table 2: Results of the running time tests of search algorithms of varying sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	812.9	877.3	2272.8	2146.8	938.7	1613.3	3048.8	4302.7	7108.2	10061.9
Linear search (sorted data)	85.1	110.2	183.7	323.7	571.9	1042.3	2132.5	4452.5	9057.2	16866.4
Binary search (sorted data)	226.5	108.7	131.5	81.7	82.9	112.8	117.8	140.5	125.7	130.6

Table 3: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Counting Sort	$\Omega(n + k)$	$\Theta(n + k)$	$O(n + k)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$O(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$O(\log n)$

Table 4: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$O(1)$
Merge sort	$O(n)$
Counting sort	$O(k)$
Linear Search	$O(1)$
Binary Search	$O(1)$

3.2.1 Insertion Sort:

- Additional memory space: O(1) - It sorts the array in place without using any additional memory.

- Line(s) used: No additional memory allocation is done in this algorithm.

3.2.2 Merge Sort:

- Additional memory space: O(n) - It requires auxiliary space for merging the two halves in the merge step.

- Line(s) used: Line 12 where array C is initialized for merging.

3.2.3 Counting Sort:

- Additional memory space: O(k) - It requires additional memory to store the count array of size k+1.

- Line(s) used: Line 2 where the count array is initialized.

3.2.4 Linear Search:

- Additional memory space: O(1) - It does not require any additional memory space, it performs the search in place.

- Line(s) used: No additional memory allocation is done in this algorithm.

3.2.5 Binary Search:

- Additional memory space: O(1) - It does not require any additional memory space, it performs the search in place.

- Line(s) used: No additional memory allocation is done in this algorithm.

Overall, the algorithms that require additional memory space are Merge Sort and Counting Sort, with Merge Sort requiring O(n) space and Counting Sort requiring O(k) space, where k is the range of elements in the input array. Linear Search and Binary Search perform in-place search operations and Insertion Sort is an in-place sorting algorithm.

3.3 Conclusion:

In conclusion, while there may be slight variations due to factors such as system resources and implementation details, the experimental results generally align with the theoretical complexities of the algorithms. This reaffirms the importance of understanding algorithmic complexities for predicting and optimizing algorithm performance in real-world scenarios.

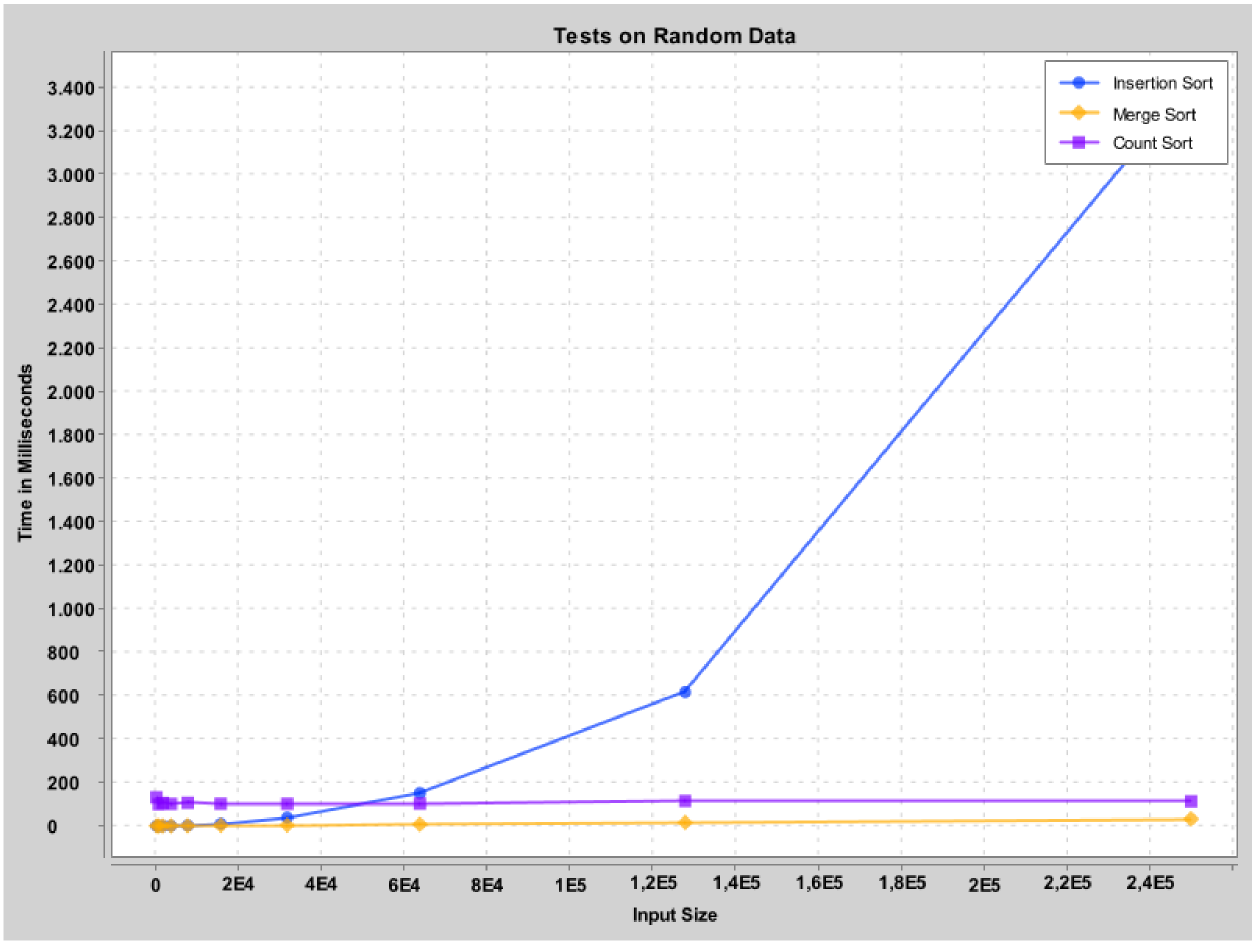

Figure 1: Sorting experiments on random data.

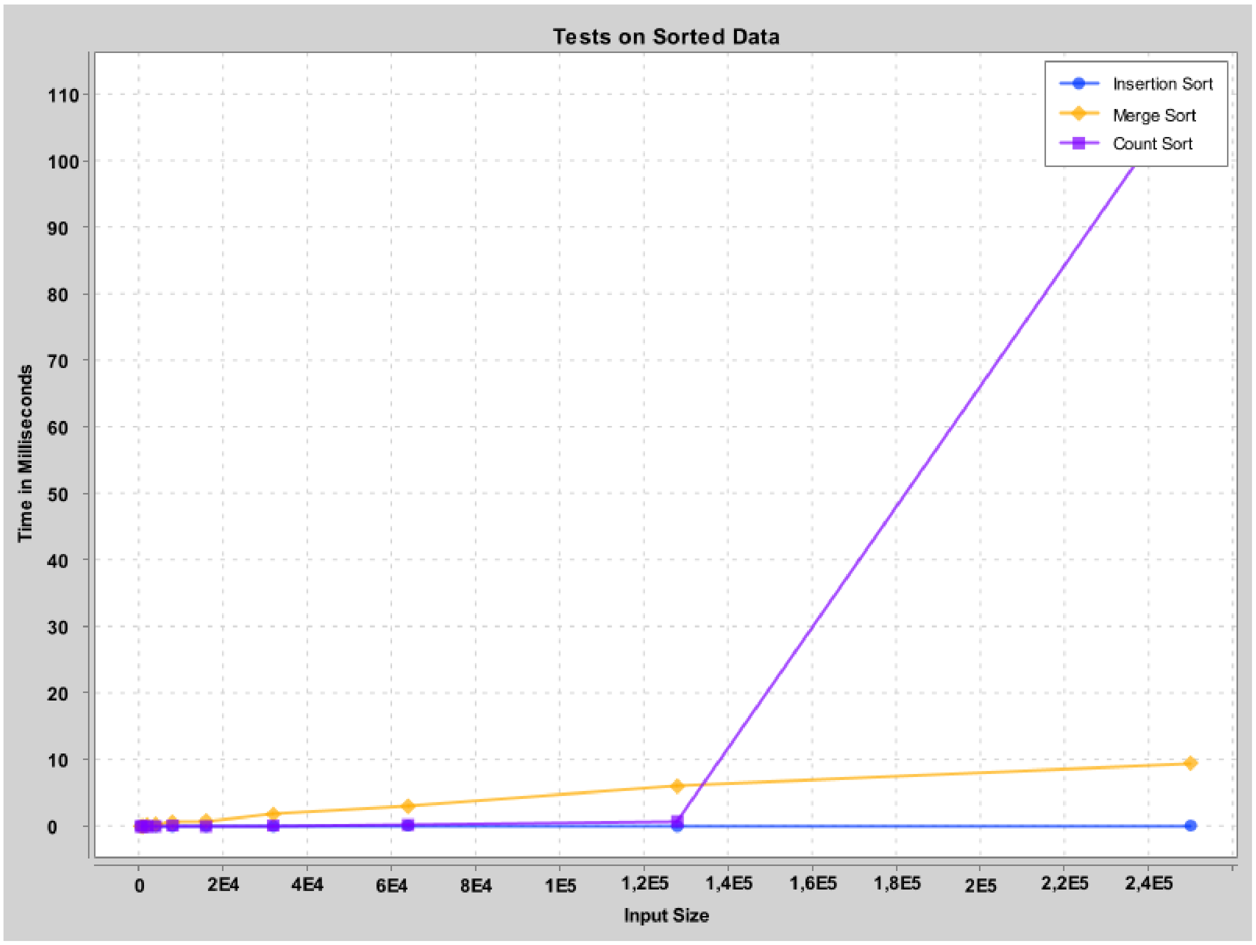

Figure 2: Sorting experiments on sorted data.

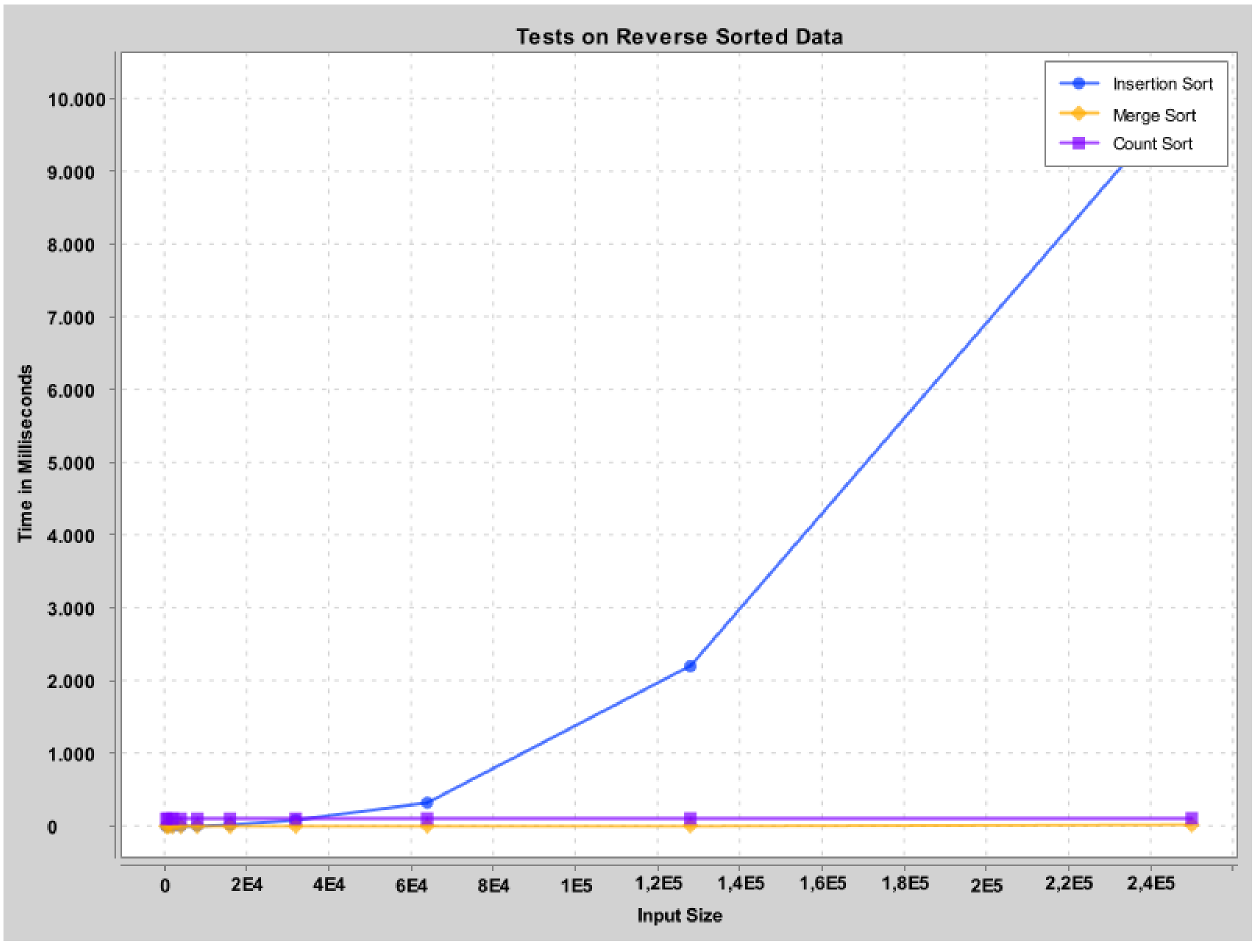

Figure 3: Sorting experiments on reverse sorted data.

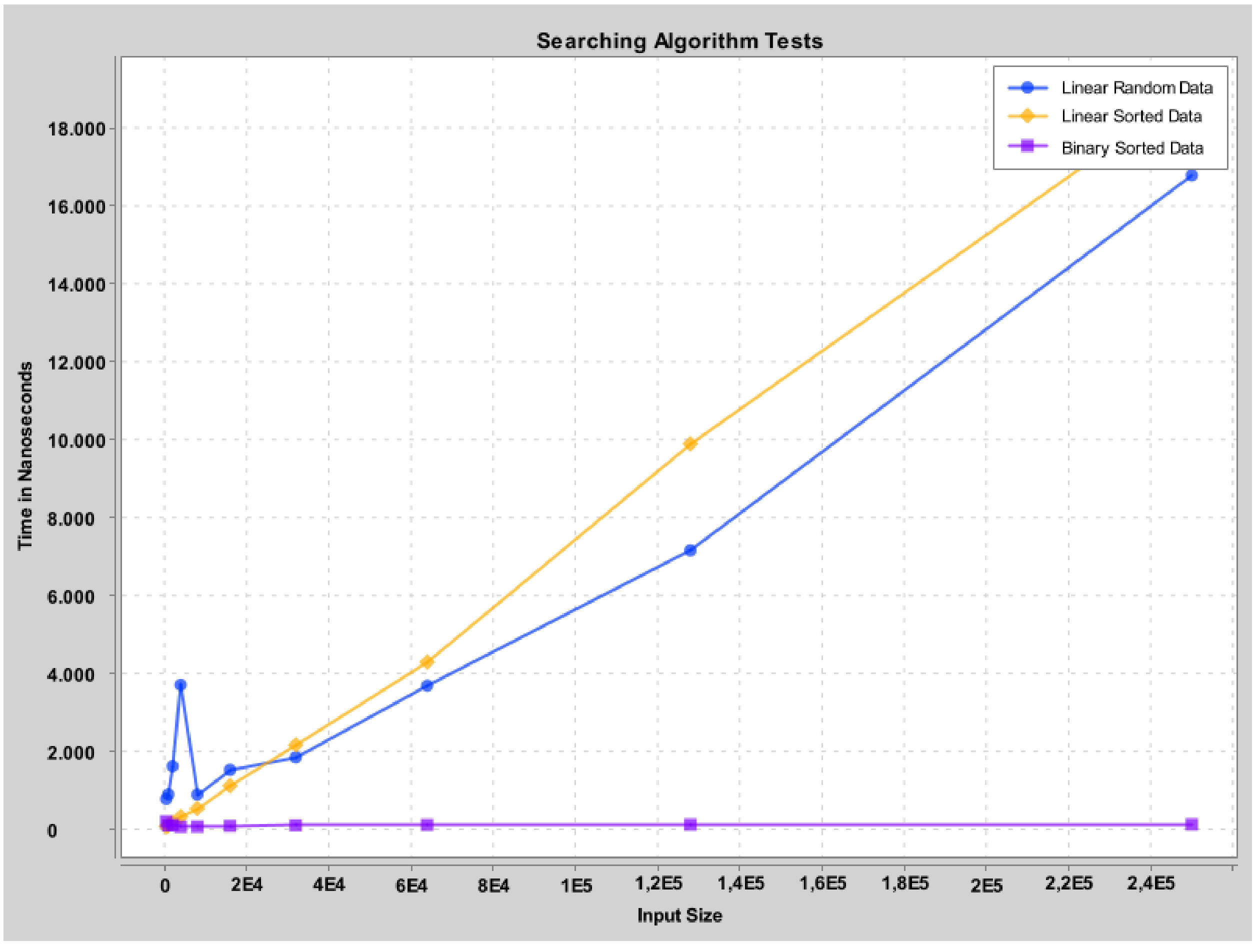

Figure 4: Search experiments.