ANA - Seminary work 2

Empirical evaluation of the subset-sum problem

Luka Šveigl (63200301)

1 Introduction

This report outlines our empirical evaluations of the four algorithms for the subset-sum problem, namely the dynamic programming solution, the exhaustive solution, a greedy approach and the FPTAS algorithm. We implemented the algorithms and additionally implemented various generators which we used to generate problems that might prove difficult for the algorithms to solve. In this report we present our implementations, before focusing on the empirical results and concluding the report with a discussion.

2 Methodology

2.1 Algorithms

Dynamic programming

The first algorithm we implemented was a dynamic programming approach to solving the subset-sum problem. This approach is defined using the following reccurrence relation:

$$S(i,j) = max(S(i-1,j), S(i-1,j-a_i)+a_i) \quad (1)$$

$$S(0,j) = 0 \quad (2)$$

$$S(i,0) = 0 \quad (3)$$

The implemented algorithm did not include any early stop conditions in case a solution matching the value of k was found before execution stopped, which means the dynamic programming table was filled in full.

Exhaustive solution

The idea of the exhaustive solution is to build lists L_i, which contain the sums that can be achieved with elements $\{a_1, ..., a_i\}$. The resulting lists are merged and sorted, with elements that are larger than k removed. The solution can be described using the following pseudocode:

```
L0 = [0]
for i = 1 ... n:
   Li = merge_and_sort(Li-1, Li-1 +
      ↪ ai)
   Li = filter(Li, =< k)
return Ln[last]
```

Greedy solution

The idea of the greedy solution is simple: we iterate through a (descending) sorted list and check if the current sum + current element is smaller than the solution. If it is, we add the element to the sum and continue, otherwise we skip the element. This algorithm can be described using the following pseudocode:

```
k' = 0
for i = 1 ... n:
   if k - k' => ai:
      k' = k' + ai
return k'
```

This algorithm is not optimal, with it's worst case being $A = \{k+1, k, k\}$ for $2k$, which will return $k+1$.

FPTAS

The idea of the FPTAS algorithm is similar to the exhaustive search. We try to keep the lists small, for which we introduce the trim operation. The trim operation removes one of two elements if they are sufficiently close to each other. The algorithm can be defined using the following pseudocode:

```
L0 = [0]
for i = 0 ... n-1:
   Li = merge_and_sort(Li-1, Li-1 +
      ↪ ai)
   Li = trim(Li, epsilon/2n)
   Li = filter(Li, =< k)
return Ln[last]
```

The trim operation can be defined using the following pseudocode:

```
L' = [L[0]]
last = L[0]
for i = 1...n-1:
   if L[i] > last * (1 - delta):
      L' = L' + [L[i]]
      last = L[i]
return L'
```

2.2 Problem generators

For each algorithm, we were required to examine ways of generating ”difficult” problem instances, i.e. instances that increase runtime in case of the optimal solutions (dynamic programming, exhaustive search) and instances that lead the algorithm away from the optimal solution in case of approximational algorithms (greedy, FPTAS).

Dynamic programming

For our dynamic programming solution, we devised generators that generate two types of tests. The first type are tests where the larger solution is comprised from many small elements, and the other type are tests where the solution is impossible to be obtained, forcing the dynamic table to be filled in full. Due to the fact that our algorithm implementation included no early stop condition, this test was redundant. The results of our tests are presented in Section 3.1.

Exhaustive solution

For our exhaustive solution, we implemented generators that generate arrays that, that contain small numbers close to each other. The results of our tests are presented in Section 3.2.

Greedy solution

For our greedy solution, we leveraged the worst case of the algorithm described in Section 2.1. The tests generate a large array, where 10% of elements have the value of $\frac{k}{2} + 1$ and 90% of elements have values that are $\frac{k}{2}$. The results of our tests are present in Section 3.3.

FPTAS

When designing the test generator for our FPTAS solution, we utilized the fact that the Trim function removes elements that are close to each other. Our test generator therefore first selects some random base value, and then based on this value generates arrays where each element deviates from it by 10. The value of k is then set to the sum of the array, minus some random value between 1 and 50. The results of these tests are presented in Section 3.4.

3 Results

3.1 Dynamic programming

The dynamic programming approach behaved as expected - the run-time of the algorithm increased steadily with the increase in the size of the array (value of n), for both types of tests, as can be observed in Figures 1 and 2.

3.2 Exhaustive solution

The exhaustive solution behaved much more erratically than the dynamic programming solution in terms of run-time. The increase in time was still present with an increasing value of n, however, at some values the algorithm took less time than with smaller values of n. This can be observed in Figure 3.

3.3 Greedy solution

Due to our utilization of the worst case of the greedy algorithm, the algorithm performed as expected in terms of the difference between the result obtained and the value k, with all of the resulting values being exactly $\frac{k}{2} + 1$. This can be observed in Figure 4.

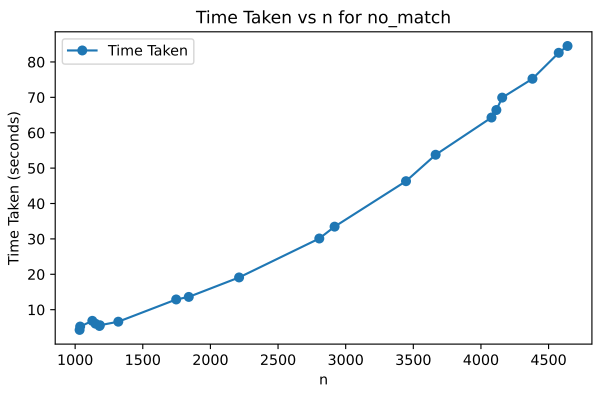

Figure 1: Dynamic programming - no match: Runtime vs n

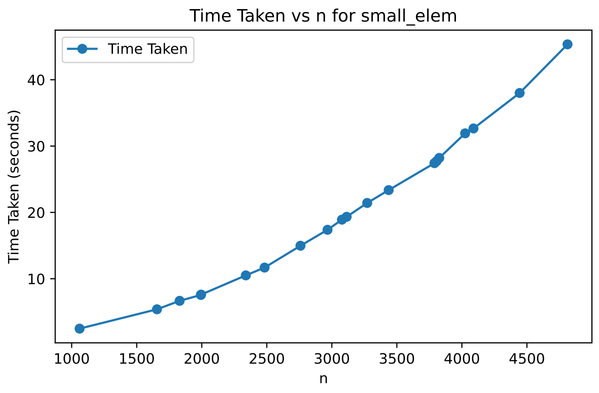

Figure 2: Dynamic programming - small elements: Runtime vs n

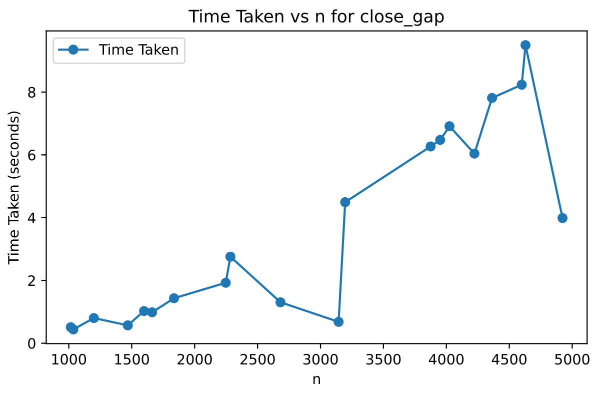

Figure 3: Exhaustive search - close gaps: Runtime vs n

3.4 FPTAS

For the FPTAS algorithm, we were required to analyze both the accuracy of the algorithm on our test cases and the importance of its epsilon parameter on the run time.

Accuracy

The results of our tests for the FPTAS algorithm are present in Figure 5. The algorithm performs much better than the greedy

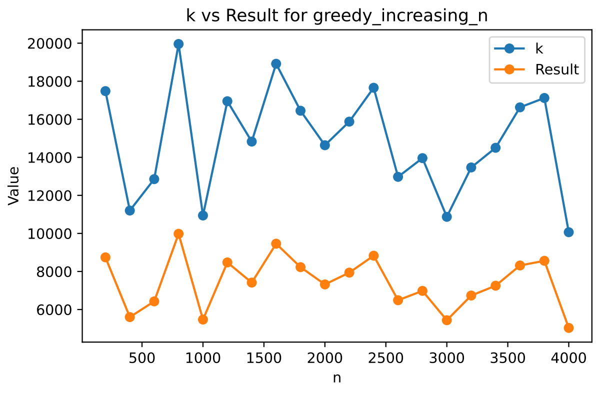

Figure 4: Greedy algorithm: result vs k

solution on our tests, indicating that we might have failed to encounter the worst case of this algorithm. The difference between the result and the target value is sufficiently small as to not be perceptible on the graph, but when examining the numerical differences, the values deviate by a small amount.

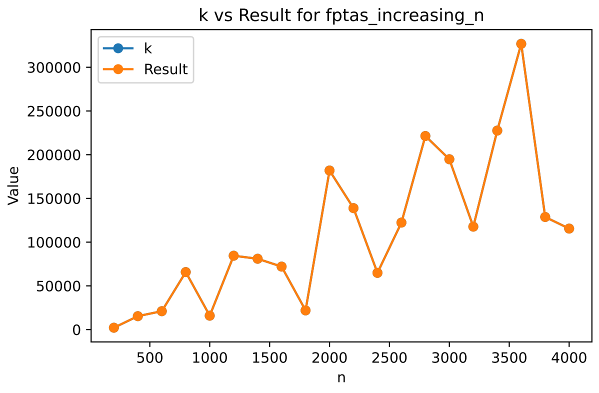

Figure 5: FPTAS algorithm: result vs k

Epsilon impact

The impact of the epsilon parameter on the run time of the algorithm is presented in Figure 6. All tests were carried out on a randomly generated array of 1800 elements (n), with the target value of 21936 (k). As expected, change in the epsilon parameter proves to be quite impactful to the run time of the algorithm, with larger values providing faster speeds. The time falls almost linearly with the increase in the epsilon parameter.

4 Discussion

The experimental results align well with theoretical expectations. The dynamic programming and exhaustive algorithms consistently produced exact results but showed poor scalability as input size increased, particularly due to the lack of early stopping in our implementation. The greedy algorithm, while extremely fast, performed poorly on adversarial inputs, highlighting its unreliability for accurate solutions. In contrast,

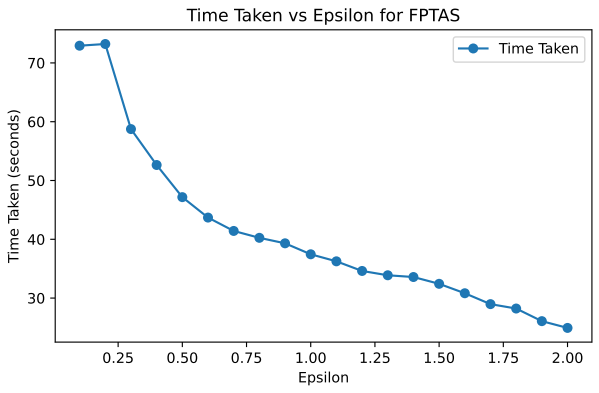

Figure 6: Impact of the epsilon parameter on the runtime of the FPTAS algorithm

FPTAS consistently produced results close to optimal, which might point to either its efficiency, or our inability to design the best adversarial inputs possible. The runtime of FPTAS decreased noticeably with larger epsilon values, reflecting the expected trade-off between speed and precision.