Implementation Experiments of Subset Problem

He Tianyang, Xiong Rongyu

June 10, 2024

Abstract

The subset sum problem is a classic optimization problem that belongs to the class of NP-hard problems. The reproduced paper has optimized the complexity of this problem to $O(\sqrt{n} \times t)$. Additionally, we compared our experimental results with two other algorithms, namely the DFS Search method and the Dynamic Programming method. We evaluated their runtime and tested them on seven given samples(Florida State University), and analyze the expermient result. The code can be found in this repository

1 Purpose

1.1 Problem Definition

The Subset Sum problem is a classic optimization problem. Given a set X and an upper bound u, the goal is to find a subset $Y \subseteq X$ such that $\sum_{y \in Y} y = u$. If it is not possible to find such a subset, the objective is to minimize the difference $\left| u - \sum_{y \in Y} y \right|$, with the sum being as close to u as possible but not exceeding u.

1.2 Possible Solutions

The classic solution to the Subset Sum problem is to use dynamic programming, which has a complexity of $O(nt)$. However, by applying a divide-and-conquer approach, we can optimize this complexity to $O(\sqrt{n}t)$ through a straightforward analysis. Additionally, this algorithm use the Fast Fourier Transform (FFT) to efficiently handle the results of each division.

2 Experiment Process

By reproducing the given paper [1] and verifying its efficiency, we designed the following experimental process:

1. Implement the DFS algorithm.
2. Implement the Dynamic Programming algorithm.
3. Implement the Given Algorithm.
4. Apply the three algorithms to the given test cases and calculate the computational cost.
5. Analyze the results and provide conclusions.

3 Algorithms

3.1 DFS Algorithm

DFS algorithm is the most simple and straightforward algorithm. given a set X, try out all possible combination of the subset Y and calculate the result. Fig. 3.1 shows the pseudo code.
the complexity can be easily calculated as $O(N!)$, and we have the code implement it in Code. 1

Algorithm 1 DFS Algorithm

Function dfs(u, sum)
if $u > n$ **then**
  if $sum \neq m$ **then**
    return
  end if
  print current choice
  return
end if
$choose[u] \leftarrow$ **true**
dfs($u + 1$, $sum + seq[u]$)
$choose[u] \leftarrow$ **false**
dfs($u + 1$, sum)

Code Frame 1: DFS Algorithm in C++

```
void dfs(int u, ll sum) {
  if (u > n) {
    if (sum != m)
      return;
    cout << "sum: " << sum << endl;
    for (int i = 1; i <= n; i++)
      cout << choose[i];
    cout << endl;
    return;
  }
  choose[u] = true;
  dfs(u + 1, sum + seq[u]);
  choose[u] = false;
  dfs(u + 1, sum);
}
```

3.2 Dynamic Planning Algorithm

Dynamic Planning Algorithm is one of the most classic and widely used algorithm for solving Knapsack Problem. Let $f(i, j)$ be the maximum sum when the sum of the first i elements is j. The initial status and the state transition equation are as follow. x_i is the i-th given number.

$$f(1, j) = \begin{cases} 0 & x_1 > j \\ x_1 & x_1 \leq j \end{cases}$$

$$f(i, j) = max\{f(i-1, j), f(i-1, j-x_i)\}$$

To provide all subsets with total sum of m, we can perform a backtracking algorithm. Fig. 3.2 and Fig. 3.2 shows the pseudo code.

Algorithm 2 DP

for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to u **do**
    $f[i][j] = f[i-1][j]$
    if $j \geq x_i$ **then**
      $f[i][j] = max\{f[i][j], f[i-1][j-x_i] + x_i\}$
    end if
  end for
end for

Algorithm 3 DP

Function `Backtracking`(i, sum)
if i = 1 **then**
  if $sum = x_1$ **then**
    $choose[1] \leftarrow$ **true**, $sum \leftarrow 0$
  end if
  if $sum = 0$ **then**
    print current choice
  end if
  return
end if
if $f[i-1][sum] = sum$ **then**
  `Backtracking`($i-1$, sum)
end if
if $f[i-1][sum - x_i] = sum - x_i$ **then**
  $choice[i] \leftarrow 1$, `Backtracking`($i-1$, $sum - x_i$), $choice[i] \leftarrow 0$
end if

The complexity can be written as $O(nu)$
and we have the code implement it in Code. 2 and Code. 3

Code Frame 2: Dynamic Programing in C++

```
void Dynamic_Programing() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            f[i][j] = f[i - 1][j];
            if (j >= seq[i])
                f[i][j] = std::max(f[i][j], f[i - 1][j - seq[i]] + seq[i]);
        }
}
```

Code Frame 3: Backtracking in C++

```
bool choice[MAXN];
void TraceBack(int x, int rest) {
    if (x == 1) {
        if (rest == seq[1])  choice[1] = 1, rest = 0;
        if (!rest) {
            for (int i = 1; i <= n; i++)
                if (choice[i])
                    cout << seq[i] << " ";
            cout << endl;
            for (int i = 1; i <= n; i++)
                cout << choice[i];
            cout << endl;
        }
        choice[1] = 0; return;
    }
    if (f[x][rest - seq[x]] == rest - seq[x])
        choice[x] = 1, TraceBack(x - 1, rest - seq[x]), choice[x] = 0;
    if (f[x][rest] == rest) TraceBack(x - 1, rest);
}
```

3.3 Proposed Algorithm

3.3.1 Proof of algorithm

To prove this algorithm, we need to define some notations following the original paper. $[u]$ donates the set $0, 1, \ldots, u$ set of integers in the interval $[0, \lceil u \rceil]$, and ΣX donates that $\Sigma_{x \in X} x$. now we can define these

notations.

$$S_u(X) = \{\Sigma Y \leq u | Y \subseteq X\} \tag{1}$$

and

$$S_u(X) = \left\{ \left(\sum Y, |Y| \right) \mid Y \subseteq X, \sum Y \leq u \right\} \tag{2}$$

Then, we define a operation $\oplus_u$ where

$$X \oplus_u Y = \{(x_1 + y_1, x_2 + y_2) \mid (x_1, x_2) \in X, (y_1, y_2) \in Y\} \cap ([u] \times \mathbb{N})$$

The author gives two types of subset sum problem definition as below.
AllSubsetSums
Input: Given a set S of n positive integers and an upper bound integer u.
Output: The set of all realizable subset sums of S up to u. **AllSubsetSums#**

Input: Given a set S of n positive integers and an upper bound integer u.
Output: The set of all realizable subset sums along with the size of the subset that realizes each sum of S up to u.

And we have:
Lemma 1 (Computing pairwise sums $\oplus_u$)
The following statements hold true:

(A) Given two sets $S, T \subseteq [u]$, one can compute $S \oplus_u T$ in $O(u \log u)$ time.

(B) Given k sets $S_1, \ldots, S_k \subseteq [u]$, one can compute $S_1 \oplus_u \cdots \oplus_u S_k$ in $O(ku \log u)$ time.

(C) Given two sets of points $S, T \subseteq [u] \times [v]$, one can compute $S \oplus_u T$ in $O(uv \log(uv))$ time.

We can prove Lemma 1 by applying the Fast Fourier Transform [2] (FFT) to optimize the $\oplus_u$ process. Consider a set X as a polynomial, where $f_S(x) = \sum_{i \in S} x^i$ and $f_S(x, y) = \sum_{(i,j) \in S^\#} x^i y^j$. The result can be represented by $f_S * f_T$, where $*$ denotes the convolution of polynomials.
the algorithm is shown as the psuedo-code below:

Algorithm 4 AllSubsetSums#(S, u)

Input: A set S of n positive integers and an upper bound integer u
Output: The set of all subset sums with cardinality information of S up to u
if $S = \{x\}$ **then**
  return $\{(0, 0), (x, 1)\}$
end if
$T \leftarrow$ an arbitrary subset of S of size $\lfloor \frac{n}{2} \rfloor$
return AllSubsetSums#(T, u) $\oplus_u$ AllSubsetSums#($S \setminus T$, u)

We can written the recursive formula $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(un \log u)$, therefore the complexity is given by $O(un \log u \log n)$

Algorithm 5 AllSubsetSums(S, u)

Input: A set S of n positive integers and an upper bound integer u
Output: The set of all realizable subset sums of S up to u
$b \leftarrow \lfloor \sqrt{n \log n} \rfloor$
for each $\ell \in [b - 1]$ **do**
  $S_\ell \leftarrow S \cap \{x \in \mathbb{N} \mid x \equiv \ell \pmod{b}\}$
  $Q_\ell \leftarrow \{(x - \ell)/b \mid x \in S_\ell\}$
  $S^\#_{u/b}(Q_\ell) \leftarrow$ AllSubsetSums#($Q_\ell, \lfloor u/b \rfloor$)
  $R_\ell \leftarrow \{zb + \ell j \mid (z, j) \in S^\#_{u/b}(Q_\ell)\}$
end for
return $R_0 \oplus_u \cdots \oplus_u R_{b-1}$

The complexity is give by

$$O\left(\left(\frac{u}{b}\sqrt{n\log n}\right)n\log n\log u + b\sqrt{n\log nu}\log u\right) = O(\sqrt{n\log n}\,u\log u) \tag{3}$$

3.3.2 Implementation Detail

Fast Fourier Transform (FFT) [2] : FFT is a classic algorithm and can be implementation by following code.

Code Frame 4: FFT in C++

```
void fft(vector<cd> &a, bool invert) {
    int n = a.size();
    if (n == 1) return;

    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[i * 2];
        a1[i] = a[i * 2 + 1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n / 2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n / 2] /= 2;
        }
        w *= wn;
    }
}
```

Polynomial Convolution

Code Frame 5: Polynomial Convolution in C++

```
vector<cd> multiply(const vector<cd> &a, const vector<cd> &b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    return fa;
}
```

Implementation of $\oplus_u$ 1D and 2D

Code Frame 6: OPlus in C++

```

vector<int> computeXorU(const vector<int> &S, const vector<int> &T, int u) {
    vector<cd> fS(u + 1), fT(u + 1);

    // Build the characteristic polynomials
    for (int s: S) {
```

```
        if (s <= u) fS[s] = 1;
    }
    for (int t: T) {
        if (t <= u) fT[t] = 1;
    }

    // Multiply the polynomials using FFT
    vector<cd> g = multiply(fS, fT);

    // Extract the result
    vector<int> result;
    for (int i = 0; i <= u; ++i) {
        if (round(g[i].real()) > 0) {
            result.push_back(i);
        }
    }

    return result;
}


using S2d = vector<pair<int, int> >;
S2d computeXorU(const S2d& S, const S2d& T, int u, int v) {
    int max_u = max(u, v);
    int n = 1;
    while (n < 2 * max_u) n <<= 1;

    vector<cd> fS_x(n, 0), fT_x(n, 0);
    vector<cd> fS_y(n, 0), fT_y(n, 0);

    for (const auto& p : S) {
        fS_x[p.first] += 1;
        fS_y[p.second] += 1;
    }
    for (const auto& p : T) {
        fT_x[p.first] += 1;
        fT_y[p.second] += 1;
    }

    vector<cd> g_x = multiply(fS_x, fT_x);
    vector<cd> g_y = multiply(fS_y, fT_y);

    S2d result;
    for (int i = 0; i < 2 * u; ++i) {
        for (int j = 0; j < 2 * v; ++j) {
            if (round(g_x[i].real()) > 0 && round(g_y[j].real()) > 0) {
                result.emplace_back(i, j);
            }
        }
    }

    return result;
}
```

Core Algorithm: follow the pseudo-code, we implement the $AllSubSetSum^{\#}$ and $AllSubSetSum$

Code Frame 7: Core Algorithm

```
S2d AllSubsetSums_sharp(const std::vector<int> &S, int u) {
    if (S.size() == 1) {
        return convert_to_S2d(S[0]);
    }

    int n = S.size();
    int half = n / 2;
    std::vector<int> T(S.begin(), S.begin() + half);
    std::vector<int> S_minus_T(S.begin() + half, S.end());

    S2d subset_sums_T = AllSubsetSums_sharp(T, u);
    S2d subset_sums_S_minus_T = AllSubsetSums_sharp(S_minus_T, u);
```

```
    return computeXorU(subset_sums_T, subset_sums_S_minus_T, u, u);
}


vector<int> AllSubsetSums(const vector<int> &S, int u) {
    int n = S.size();
    int b = floor(sqrt(n * log(n)));
    vector<vector<int> > R(b);

    for (int ell = 0; ell < b; ++ell) {
        vector<int> S_ell;
        for (int x: S) {
            if (x % b == ell) {
                S_ell.push_back(x);
            }
        }

        vector<int> Q_ell;
        for (int x: S_ell) {
            Q_ell.push_back((x - ell) / b);
        }

        S2d S_u_div_b_sharp_Q_ell = AllSubsetSums_sharp(Q_ell, u / b);

        for (const auto &p: S_u_div_b_sharp_Q_ell) {
            int z = p.first;
            int j = p.second;
            R[ell].push_back(z * b + j * ell);
        }
    }

    // Combine all R_ell sets using the provided logic
    vector<int> result = R[0];
    for (int i = 1; i < b; ++i) {
        result = computeXorU(result, R[i], u);
    }

    return result;
}
```

4 Experiment

4.1 Experiment Settings

We use 7 test cases provided by Florida State University and measure the sum computation time and compare three different algorithms. The experiment is run on a Intel core i9-12900H CPU and with unlimited memory usage. Note that the given data have very small values for n where $n \leq 21$. Therefore, the most optimized algorithm may not be the proposed one due to the constant by the overhead of using the Standard Template Library (STL).

	1	2	3	4	5	6	7	sum	Gain
DFS	4061	2143	18300	2197	4728	5164	4852	41445	N/A
Dynamic Planning	7061	8029	162255	4958	4114	3267	4383	194067	-152622
Proposed [1]	7203	259823	172255	3561	4474	3378	3067	453761	-412316

Table 1: Comparison of mean computation time for different algorithms (μs)

5 Conclusion

The experiment results are shown in Table 1. We notice that the dynamic planning algorithm's complexity is $O(nu)$, and in this set of problems, u is usually very large (e.g., $u = 2,463,098$ in test case 3). Consequently, the computation time is much higher than that of the regular DFS algorithm. Although the new algorithm has a significant improvement in complexity, the given dataset features a large u but small n. Combined with the use of STL, this results in the worst performance. However, we cannot deny that the new algorithm shows considerable improvement in handling problems with large n.

In this report, we implement two common algorithms and a newly proposed algorithm from a recent paper. We conduct complexity experiments and analyze the results. The new algorithm uses a simple divide-and-conquer approach to tackle the problem and employs FFT (although it may seem unrelated) to merge two ranges. This demonstrates how simple ideas can be effectively utilized when addressing new problems.

References

[1] Konstantinos Koiliaris and Chao Xu. Subset sum made simple, 2018.

[2] C. Rader and N. Brenner. A new principle for fast fourier transformation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(3):264–266, 1976.