Ex1(implementacao2)

May 1, 2024

1 Trabalho Prático 3

André Freitas PG54707

Bruna Macieira PG54467

1.1 Exercício 1

No capítulo 5 dos apontamentos é descrito o chamado Hidden Number Problem. No capítulo 8 dos apontamentos é discutida um artigo de Nguyen & Shparlinsk , onde se propõem reduções do HNP a problemas difíceis em reticulados. Neste trabalho pretende-se construir, com a ajuda do Sagemath, uma implementação da solução discutida nos apontamentos para resolver o HNP com soluções aproximadas dos problemas em reticulados.

[ ]:
```
%pip install sagemath-standard

from sage.all import *
from fpylll import *
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: sagemath-standard in /usr/lib/python3/dist-
packages (9.5)
Requirement already satisfied: cysignals>=1.10.2 in
/home/fura/.sage/local/lib/python3.10/site-packages (from sagemath-standard)
(1.11.4)
Note: you may need to restart the kernel to use updated packages.
```

[ ]:
```
def msb(y, k):
    """
    Devolve os k bits mais significativos de n.

    Args:
    - y (int): O número do qual se quer obter os bits.
    - k (int): O número de bits a obter.

    Returns:
    - int: Os k bits mais significativos de n.
    """
```

```python
    q = y.order()  # Get the order of the finite field that y belongs to
    B = q // (2 ** k)  # Calculate the bound
    return y // B  # Calculate and return the most significant k bits of y
```

[ ]:
```python
def generate_hnp_instance(p, s, k, N):
    """
    Gera uma instância do Hidden Number Problem (HNP).

    Args:
    - q (int): Um primo, o tamanho do campo finito.
    - s (int): O segredo a ser descoberto.
    - k (int): O número de bits significativos a serem considerados.
    - N (int): O número de pares (x, u) a serem gerados.

    Returns:
    - list: Uma lista de pares (x, u).
    """
    Fp = FiniteField(p)
    secret = Fp(s)
    pairs = []
    for _ in range(N):
        x = Fp.random_element()
        u = msb(secret * x, k)  # Use the msb function here
        pairs.append((x, u))

    print("Pairs:", pairs)
    return pairs
```

[ ]:
```python
def hnp_to_bdd_reduction(hnp_instance, p, k, s, M):
    """
    Reduces a Hidden Number Problem (HNP) instance to the Bounded Distance␣
 ↪Decoding (BDD) problem.

    Args:
    - hnp_instance (list): A list of pairs (x, u) generated by the HNP.
    - q (int): A prime, the size of the finite field.
    - k (int): The number of significant bits considered in the HNP.
    - s (int): The secret to be discovered.
    - A (int): A parameter for the lattice.
    - M (int): A parameter for the lattice.
    - lambda_ (float): The smoothing parameter for the BKZ reduction.

    Returns:
    - list: A list of vectors representing the lattice.
    - vector: The target vector of the lattice.
    """
    n = len(hnp_instance)
```

```
    m = n + 2
    lambda_ = 2 ** k
    A = 1 // lambda_

    B = p // lambda_  # Bound of the lattice

    lattice_matrix = Matrix(ZZ, m, m)
    target_vector = vector(ZZ, m)

    Fp = FiniteField(p)
    z = Fp.random_element()
    secret = Fp(s)
    u = msb(secret * z, k)

    # Fill the lattice matrix with the pairs (x, y)
    for i in range(n):
        x, y = hnp_instance[i]
        lattice_matrix[i, i] = p
        lattice_matrix[i, n] = -B * y
        lattice_matrix[i, n + 1] = x
        target_vector[i] = B * u

    # Fill the last row of the matrix with the values A and M
    lattice_matrix[n, :n] = vector([hnp_instance[i][0] for i in range(n)])
    lattice_matrix[n, n] = A
    lattice_matrix[n + 1, n + 1] = M

    # Apply the BKZ reduction
    L = lattice_matrix.BKZ()

    print("Lattice matrix")
    print(lattice_matrix)
    print("Target vector")
    print(target_vector)
    print("BKZ-reduced matrix")
    print(L)

    # The last row after BKZ reduction should be [e_1, e_2, ..., e_n+1, M]
    e_n_plus_1 = L[n, n]
    s = ceil(lambda_ * e_n_plus_1)  # s is ceil(lambda * e_n+1)
    print("s:", s)

    return L, s, target_vector
```

[ ]:
```
def solve_bdd_lattice(lattice_matrix,  target_vector, bound):
    """
```

```
    Resolve o Problema de Decodificação de Base em Reticulados (BDD) usando o␣
 ↪algoritmo BKZ.

    Args:
    - lattice_matrix (Matrix): A matriz representando o reticulado.
    - target_vector (Vector): O vetor alvo do reticulado.
    - bound (int): O bound do reticulado.

    Returns:
    - list: Uma lista de vetores que aproximam o vetor alvo.
    """
    L = IntegerMatrix.from_matrix(lattice_matrix.T)
    L = LLL.reduction(L)
    bkz = BKZ.Param(block_size=100, strategies=BKZ.DEFAULT_STRATEGY)
    BKZ.reduction(L, bkz)

    closest_vectors = []
    for i in range(L.nrows):
        v = L[i]
        norm = v.norm()
        norm_t = target_vector.norm()
        print("Target vector norm:", norm_t)
        print(f"Norm of vector {i}: {norm}")
        if bound > norm_t:
            continue
        closest_vectors.append(list(v))

    print("Closest vectors:", closest_vectors)
    print(f"Bound: {bound}")
    return closest_vectors
```

[ ]:
```
def hnp_bdd_solver(p, s, k, N, M):
    """
    Resolve o Hidden Number Problem (HNP) e o reduz para o Problema de␣
 ↪Decodificação de Base em Reticulados (BDD).

    Args:
    - q (int): Um primo, o tamanho do campo finito.
    - s (int): O segredo a ser descoberto.
    - k (int): O número de bits significativos a serem considerados.
    - N (int): O número de pares (x, u) a serem gerados.

    Returns:
    - int: O segredo recuperado.
    """
    # Gera uma instância do HNP
    hnp_instance = generate_hnp_instance(p, s, k, N)
```

```
    # Reduz o HNP para o BDD
    lattice_matrix,s,target_vector = hnp_to_bdd_reduction(hnp_instance, p, k,␣
↪s, M)

    # Calculate lambda as the absolute value of the determinant of the lattice␣
↪matrix
    lambda_ = 2 ** k
    print("Lambda:", lambda_)

    # Bound do reticulado
    bound = p // lambda_

    # Resolve o BDD usando BKZ
    closest_vectors = solve_bdd_lattice(lattice_matrix, target_vector, bound)

    # Recupera o segredo s
    recovered_secret = None
    if closest_vectors:
        # Select the closest vector based on its distance to the target vector
        closest_vector = min(closest_vectors, key=lambda v: (vector(v) -␣
↪target_vector).norm())
        recovered_secret = lambda_ * closest_vector[-2]  # Multiply the second␣
↪last component by lambda and round to the nearest integer

    print("Target vector:", target_vector)
    print("Closest vector:", closest_vector)
    print("last Closest vector:", closest_vector[-2])
    print("Recovered secret:", recovered_secret)
    print("Correct secret:", s)

    return recovered_secret
```

[ ]:
```
# Teste da implementação
q = next_prime(1351)  # Tamanho do campo finito
s = 4   # Segredo
k = 1    # Número de bits significativos
N = 6   # Número de pares (x, u) a serem gerados
M = 1000 # Parâmetro do reticulado
```

[ ]:
```
closest_vectors = hnp_bdd_solver(q, s, k, N, M)
```

```
Pairs: [(621, 476), (46, 993), (581, 796), (1305, 448), (183, 1258), (155, 121)]
Lattice matrix
[1361    0    0    0    0    0  238  621]
[   0 1361    0    0    0    0 1177   46]
[   0    0 1361    0    0    0  398  581]
```

```
[   0    0    0 1361    0    0  224 1305]
[   0    0    0    0 1361    0  629  183]
[   0    0    0    0    0 1361  741  155]
[ 621   46  581 1305  183  155    0    0]
[   0    0    0    0    0    0    0 1000]
Target vector
(434, 434, 434, 434, 434, 434, 0, 0)
BKZ-reduced matrix
[ 621   46  581  -56  183  155 -224 -305]
[   0    0    0    0    0    0    0 1000]
[ 740  -46 -581   56 -183 -155  462  -74]
[-621  -46  780   56 -183 -155  622 -114]
[   0    0    0 1361    0    0  224  305]
[   0    0    0    0 1361    0  629  183]
[-119 1453 -199 -112  366  310   93  234]
[   0    0    0    0    0 1361  741  155]
s: 186
Lambda: 2
Target vector norm: 434*sqrt(6)
Norm of vector 0: 1154.5748135136155
Target vector norm: 434*sqrt(6)
Norm of vector 1: 1150.270837672589
Target vector norm: 434*sqrt(6)
Norm of vector 2: 1147.515577236318
Target vector norm: 434*sqrt(6)
Norm of vector 3: 1286.0991408130246
Target vector norm: 434*sqrt(6)
Norm of vector 4: 1416.1091059660623
Target vector norm: 434*sqrt(6)
Norm of vector 5: 1369.040905159521
Target vector norm: 434*sqrt(6)
Norm of vector 6: 1444.5566794002928
Target vector norm: 434*sqrt(6)
Norm of vector 7: 1455.182806385507
Closest vectors: [[621, 0, 740, -621, 0, 0, -119, 0], [581, 0, -581, 780, 0, 0,
-199, 0], [-305, 1000, -74, -114, 305, 183, 234, 155], [-224, 0, 462, 622, 224,
629, 93, 741], [379, 0, -617, -777, -224, -629, 217, 620], [-56, 0, 56, 56,
1361, 0, -112, 0], [183, 0, -183, -183, 0, 1361, 366, 0], [46, 0, -46, -46, 0,
0, 1453, 0]]
Bound: 680
Target vector: (434, 434, 434, 434, 434, 434, 0, 0)
Closest vector: [-224, 0, 462, 622, 224, 629, 93, 741]
last Closest vector: 93
Recovered secret: 186
Correct secret: 186
```

[ ]:
```
if closest_vectors:
    print("Segredo recuperado:", closest_vectors)
else:
    print("Não foi possível encontrar uma solução.")
```

```
Segredo recuperado: 186
```