Implementation of Backtracking to Find Quadratic and Biquadratic Integer Ring's Factors

Zeki Amani – 13524082
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: zekiamanioke@gmail.com, 13524082@std.stei.itb.ac.id

***Abstract*—This paper discusses the application of backtracking to search for nontrivial divisors in quadratic and biquadratic integer rings. The algorithm constructs candidate divisors from integer coefficients and uses norm calculation as a search boundary. A candidate is accepted when it divides the target element exactly, is not a unit, and is not associate to the target element. The method is implemented in Go and tested on several quadratic and biquadratic integer ring examples.**

***Index Terms*—Backtracking, factorization, quadratic integer ring, biquadratic integer ring.**

I. Introduction

Factorization is an important concept in mathematic. In ordinary integer arithmetic, a nonzero integer can be expressed as a product of prime numbers. However, factorization in other rings can have a more complicated structure. An element may contain radical components, have multiple nontrivial divisors, or have more than one distinct factorization into irreducible elements [1].

Quadratic integer rings and biquadratic integer rings are examples of rings in which factorization can be studied beyond the ordinary integers. Searching for divisors in these rings requires more than checking ordinary integer factors because a candidate divisor may involve one or more radical components.

In this paper, we will discuss the implementation of backtracking to find the factors of both quadratic and biquadratic integer ring. and in the end we will show a demonstration of factorization using this method in golang.

II. Theoretical Foundations

This section explains the algebraic structures and algorithmic concepts used in this paper. The concepts include groups, rings, factorization in integral domains, quadratic integer rings, biquadratic integer rings, and backtracking.

A. Group and Ring

A group is a mathematical structure consisting of a set of elements together with a binary operation. A group is usually denoted by $(G, \circ)$, where G is a set and $\circ$ is an operation defined on the elements of G.

For $(G, \circ)$ to be called a group, it must satisfy the following properties [1].

1) *Closure*. For every $a, b \in G$, the result $a \circ b$ must also belong to G.
2) *Associativity*. For every $a, b, c \in G$,
$$(a \circ b) \circ c = a \circ (b \circ c). \tag{1}$$
3) *Identity element*. There exists an element $e \in G$ such that
$$a \circ e = e \circ a = a \tag{2}$$
for every $a \in G$.
4) *Inverse element*. For every $a \in G$, there exists an element $a^{-1} \in G$ such that
$$a \circ a^{-1} = a^{-1} \circ a = e. \tag{3}$$

One example of a group is $(\mathbb{Z}, +)$, which is the set of integers under addition. The identity element is 0, and the inverse of an integer a is $-a$.

A ring is a set equipped with two binary operations, namely addition and multiplication. A ring is denoted by $(R, +, \cdot)$. Under addition, R forms an abelian group. Multiplication is associative and satisfies the distributive properties [1]:

$$a(b + c) = ab + ac \tag{4}$$

and

$$(a + b)c = ac + bc \tag{5}$$

for every $a, b, c \in R$.

The rings used in this paper are commutative rings with an identity element. Therefore, for every $a, b \in R$,

$$ab = ba \tag{6}$$

and there exists an element $1 \in R$ such that

$$a \cdot 1 = 1 \cdot a = a. \tag{7}$$

B. Integral Domain and Factorization

An integral domain is a nonzero commutative ring with identity that has no zero divisors. In other words, for every $a, b \in R$,

$$ab = 0 \tag{8}$$

implies that

$$a = 0 \qquad \text{or} \qquad b = 0. \tag{9}$$

The absence of zero divisors allows factorization and divisibility to behave similarly to ordinary integer arithmetic.

An element $u \in R$ is called a *unit* if there exists another element $v \in R$ such that

$$uv = vu = 1. \tag{10}$$

Two elements $a, b \in R$ are called associates if there exists a unit u such that

$$a = ub. \tag{11}$$

For example, in $\mathbb{Z}$, the units are 1 and -1. Therefore, 3 and -3 are associates.

A nonzero nonunit element $p \in R$ is called irreducible if

$$p = ab \tag{12}$$

implies that either a or b is a unit. Thus, an irreducible element cannot be factored into two nonunit elements.

A nonzero nonunit element $p \in R$ is called prime if

$$p \mid ab \tag{13}$$

implies that

$$p \mid a \qquad \text{or} \qquad p \mid b. \tag{14}$$

Every prime element is irreducible. However, the converse is not always true. An irreducible element can fail to be prime in an integral domain that does not have unique factorization [1].

A Unique Factorization Domain, commonly abbreviated as UFD, is an integral domain in which every nonzero nonunit element can be factored into irreducible elements, and the factorization is unique up to the order of factors and multiplication by units [1].

For example, $\mathbb{Z}$ is a UFD because every nonzero integer has a unique prime factorization, except for the order of factors and multiplication by -1.

C. Quadratic Integer Ring

Let d be a square-free integer. A quadratic integer ring considered in this paper has the form

$$\mathbb{Z}[\sqrt{d}] = \{a + b\sqrt{d} \mid a, b \in \mathbb{Z}\}. \tag{15}$$

For two elements

$$\alpha = a + b\sqrt{d} \tag{16}$$

and

$$\beta = c + e\sqrt{d}, \tag{17}$$

their product is

$$\alpha\beta = (ac + bde) + (ae + bc)\sqrt{d}. \tag{18}$$

Since both coefficients in (18) are integers, the product remains in $\mathbb{Z}[\sqrt{d}]$.

The conjugate of an element

$$\alpha = a + b\sqrt{d} \tag{19}$$

is defined as

$$\overline{\alpha} = a - b\sqrt{d}. \tag{20}$$

The norm of α is

$$N(\alpha) = \alpha\overline{\alpha} = a^2 - db^2. \tag{21}$$

The norm is multiplicative:

$$N(\alpha\beta) = N(\alpha)N(\beta). \tag{22}$$

This property is useful for factorization because if β divides α, then $N(\beta)$ must divide $N(\alpha)$ [2].

Not every quadratic integer ring is a UFD.

$$R_q = \mathbb{Z}[\sqrt{-5}], \tag{23}$$

is an example of a quadratic integer ring that is not a UFD [3].

For an element

$$\alpha = a + b\sqrt{-5}, \tag{24}$$

the norm becomes

$$N(\alpha) = a^2 + 5b^2. \tag{25}$$

The only units in R_q are 1 and -1. This follows because the equation

$$a^2 + 5b^2 = 1 \tag{26}$$

only has the solutions $(a, b) = (1, 0)$ and $(a, b) = (-1, 0)$.

The element 6 has two factorizations in R_q:

$$6 = 2 \cdot 3 \tag{27}$$

and

$$6 = (1 + \sqrt{-5})(1 - \sqrt{-5}). \tag{28}$$

The norms of the factors are

$$\begin{aligned} N(2) &= 4, \\ N(3) &= 9, \\ N(1 + \sqrt{-5}) &= 6, \\ N(1 - \sqrt{-5}) &= 6. \end{aligned} \tag{29}$$

There is no element in R_q with norm 2 or norm 3. Therefore, the elements 2, 3, $1 + \sqrt{-5}$, and $1 - \sqrt{-5}$ are irreducible.

Since the only units are 1 and -1, none of the factors from the first factorization is associate to a factor from the second factorization. Therefore, the two factorizations are distinct up to associates. Hence,

$$\mathbb{Z}[\sqrt{-5}] \tag{30}$$

is not a UFD.

D. Biquadratic Integer Ring

Let r and s be distinct square-free integers such that rs is not a perfect square. A biquadratic field is a field of the form

$$K = \mathbb{Q}(\sqrt{r}, \sqrt{s}). \tag{31}$$

The field K has degree four over $\mathbb{Q}$. An element of K can be written as

$$\alpha = a + b\sqrt{r} + c\sqrt{s} + d\sqrt{rs}, \tag{32}$$

where $a, b, c, d \in \mathbb{Q}$.

The ring considered in this paper is

$$R = \mathbb{Z}[\sqrt{r}, \sqrt{s}] = \left\{ a + b\sqrt{r} + c\sqrt{s} + d\sqrt{rs} \mid a, b, c, d \in \mathbb{Z} \right\}. \tag{33}$$

More precisely, R is an order contained in the ring of integers of K. For some values of r and s, the full ring of integers of K may contain additional elements with fractional coefficients. However, the ring in (54) is closed under addition, subtraction, and multiplication, so it is sufficient for the factorization method discussed in this paper [2].

Let

$$\alpha = a + b\sqrt{r} + c\sqrt{s} + d\sqrt{rs} \tag{34}$$

and

$$\beta = e + f\sqrt{r} + g\sqrt{s} + h\sqrt{rs}. \tag{35}$$

Their product is

$$\begin{aligned} \alpha\beta = {} & (ae + rbf + scg + rsdh) \\ & + (af + be + sch + sdg)\sqrt{r} \\ & + (ag + ce + rbh + rdf)\sqrt{s} \\ & + (ah + de + bg + cf)\sqrt{rs}. \end{aligned} \tag{36}$$

All coefficients in (36) are integers. Therefore, $\alpha\beta \in R$ whenever $\alpha, \beta \in R$.

There are four embeddings of K into the complex numbers. They are obtained by independently changing the signs of $\sqrt{r}$ and $\sqrt{s}$. Since

$$\sqrt{rs} = \sqrt{r}\sqrt{s}, \tag{37}$$

changing the sign of only one radical also changes the sign of $\sqrt{rs}$. Thus, the four conjugates of α are

$$\alpha_0 = a + b\sqrt{r} + c\sqrt{s} + d\sqrt{rs}, \tag{38}$$

$$\alpha_1 = a - b\sqrt{r} + c\sqrt{s} - d\sqrt{rs}, \tag{39}$$

$$\alpha_2 = a + b\sqrt{r} - c\sqrt{s} - d\sqrt{rs}, \tag{40}$$

and

$$\alpha_3 = a - b\sqrt{r} - c\sqrt{s} + d\sqrt{rs}. \tag{41}$$

The field norm of α from K to $\mathbb{Q}$ is defined as the product of all of its conjugates:

$$N(\alpha) = \alpha_0\alpha_1\alpha_2\alpha_3. \tag{42}$$

For an element of R, the norm is an integer. By first multiplying α_0 and α_2, define

$$A = a^2 + rb^2 - sc^2 - rsd^2 \tag{43}$$

and

$$B = 2ab - 2scd. \tag{44}$$

The norm can then be written as

$$N(\alpha) = A^2 - rB^2. \tag{45}$$

The norm is multiplicative:

$$N(\alpha\beta) = N(\alpha)N(\beta). \tag{46}$$

For the search method in this paper, the radicals are restricted to

$$r < 0 \qquad \text{and} \qquad s < 0. \tag{47}$$

E. Backtracking

Backtracking is a systematic search method that constructs a solution step by step. It can be viewed as an improvement of exhaustive search. In exhaustive search, every possible candidate solution is generated and evaluated. In backtracking, a partial candidate is discarded as soon as it is known that the candidate cannot lead to a valid solution. This process is called *pruning* [4].

In general, a solution is represented as an n-tuple

$$X = (x_1, x_2, \ldots, x_n), \tag{48}$$

where each component x_k is selected from a set of possible values S_k. A complete solution is obtained only after every component has been assigned a value.

Backtracking has three general properties [4].

1) The solution is represented as a vector

$$X = (x_1, x_2, \ldots, x_n). \tag{49}$$

2) A candidate generation function, denoted by T, generates possible values for the next component. If the first $k-1$ components have been assigned, then

$$T(x_1, x_2, \ldots, x_{k-1}) \tag{50}$$

generates candidate values for x_k.

3) A bounding function, denoted by B, determines whether a partial solution may still lead to a valid complete solution. The function is written as

$$B(x_1, x_2, \ldots, x_k). \tag{51}$$

If the value is `false`, the partial solution is discarded and is not expanded further.

All possible candidate solutions form a *solution space*. This solution space can be organized as a rooted tree called a *state-space tree*. The root represents an empty solution. A node at level k represents a partial solution

$$(x_1, x_2, \ldots, x_k). \tag{52}$$

Each edge from a node at level $k-1$ to a node at level k represents the selection of one value for x_k. Therefore, every path from the root to a leaf represents one complete candidate solution.

The search follows a depth-first order. A node that has been generated but has not yet been expanded is called a *live node*. The live node currently being explored is called an *expand node*. If the bounding function determines that a node cannot lead to a solution, the node becomes a *dead node*. Its descendants are not generated, which means that the corresponding branch is pruned.

The general search process is as follows.

1) Start from the root, which represents an empty solution.
2) Generate a candidate value for the next component using the function T.
3) Evaluate the bounding function B for the resulting partial solution.
4) If B is false, discard the current node and return to the previous decision level.
5) If B is true and the partial solution is complete, test whether it is a valid solution.
6) If B is true but the solution is not yet complete, expand the current node by recursively generating the next component.
7) Continue until every relevant branch has been explored or until the desired solution has been found.

In the worst case, backtracking can still explore every node in the state-space tree. Therefore, its time complexity can remain exponential. However, an effective bounding function can substantially reduce the number of candidate solutions that need to be evaluated.

III. Proposed Solution

This paper proposes a backtracking-based method for searching nontrivial divisors in quadratic and biquadratic integer rings. The method does not directly construct a complete factorization into irreducible elements. Instead, it searches for elements that divide a given target element exactly and returns every valid nontrivial divisor found within the search boundary.

The method is designed for two ring representations. The first is a quadratic integer ring

$$\mathbb{Z}[\sqrt{d}], \tag{53}$$

where $d < 0$. The second is a biquadratic integer ring

$$\mathbb{Z}[\sqrt{d}, \sqrt{e}] = \left\{ a + b\sqrt{d} + c\sqrt{e} + f\sqrt{de} \mid a, b, c, f \in \mathbb{Z} \right\}. \tag{54}$$

where $d < 0$, $e < 0$, and $de > 0$.

A. Problem Formulation

Let α be a nonzero nonunit target element in one of the supported rings. The objective is to find a candidate element β such that

$$\alpha = \beta\gamma, \tag{55}$$

where α, β, and γ belong to the same ring.

A candidate β is accepted as a solution only if it satisfies the following conditions:

1) $\beta | \alpha$
2) β not a unit;
3) β is not an associate of α.

The output of the algorithm is a list of valid divisors a.k.a factors of α.

B. Candidate Representation

The candidate factor is represented by an integer coefficient vector. The number of coefficients depends on the ring type.

For a quadratic integer ring, a candidate has the form

$$\beta = x_1 + x_2\sqrt{d}, \tag{56}$$

where $x_1, x_2 \in \mathbb{Z}$. Thus, the solution vector is

$$X = (x_1, x_2). \tag{57}$$

For a biquadratic integer ring, a candidate has the form

$$\beta = x_1 + x_2\sqrt{r} + x_3\sqrt{s} + x_4\sqrt{rs}, \tag{58}$$

where $x_1, x_2, x_3, x_4 \in \mathbb{Z}$. The solution vector is

$$X = (x_1, x_2, x_3, x_4). \tag{59}$$

Therefore, the state-space tree has depth two for quadratic integer rings and depth four for biquadratic integer rings. A node at level k represents a partial coefficient vector

$$(x_1, x_2, \ldots, x_k). \tag{60}$$

All coefficients that have not yet been assigned are temporarily treated as zero when the norm of the partial candidate is evaluated.

C. Candidate Generation Function

For each coefficient, the search begins from zero and explores positive values first:

$$0, 1, 2, 3, \ldots \tag{61}$$

After the positive values have been explored, the search continues with negative values:

$$-1, -2, -3, \ldots \tag{62}$$

For example, in a quadratic integer ring, the partial path

$$() \to (1) \to (1, 1) \tag{63}$$

represents the candidate

$$1 + \sqrt{d}. \tag{64}$$

In a biquadratic integer ring, the path

$$() \to (1) \to (1, 0) \to (1, 0, 1) \to (1, 0, 1, 0) \tag{65}$$

represents the candidate

$$1 + \sqrt{s}. \tag{66}$$

The search procedure recursively extends the coefficient vector until the number of assigned coefficients is equal to the number of coefficients required by the ring.

D. Boundary Function

The norm function is used for the boundary function as follows,

$$B(X) = \begin{cases} \texttt{true}, & N(\beta) \le N(\alpha), \\ \texttt{false}, & N(\beta) > N(\alpha). \end{cases} \tag{67}$$

When the norm of a generated partial candidate exceeds the norm of the target, the current coefficient search direction is stopped. The algorithm then returns to the previous level and tries another coefficient value.

For the quadratic case, this boundary is mathematically valid. If β divides α, then norm multiplicativity gives

$$N(\alpha) = N(\beta)N(\gamma), \tag{68}$$

for some γ in the same ring. Since the norm is positive for nonzero elements when $d < 0$,

$$N(\beta) \le N(\alpha). \tag{69}$$

Furthermore, because the quadratic norm is positive definite, adding a coefficient with a larger absolute value cannot reduce the norm.

E. Divisibility Test for Quadratic Integer Rings

Let the target element be

$$\alpha = a + b\sqrt{d} \tag{70}$$

and let the complete candidate be

$$\beta = p + q\sqrt{d}. \tag{71}$$

The conjugate of β is

$$\overline{\beta} = p - q\sqrt{d}. \tag{72}$$

The quotient is obtained using

$$\frac{\alpha}{\beta} = \frac{\alpha\overline{\beta}}{\beta\overline{\beta}}. \tag{73}$$

After expansion,

$$\frac{\alpha}{\beta} = \frac{ap - dbq}{p^2 - dq^2} + \frac{bp - aq}{p^2 - dq^2}\sqrt{d}. \tag{74}$$

The candidate β divides α if

$$N(\beta) \neq 0 \tag{75}$$

and both coefficients in (74) are integers.

F. Divisibility Test for Biquadratic Integer Rings

Let

$$\beta = a + b\sqrt{r} + c\sqrt{s} + e\sqrt{rs}. \tag{76}$$

The three nontrivial conjugates of β are

$$\beta_1 = a - b\sqrt{r} + c\sqrt{s} - e\sqrt{rs}, \tag{77}$$

$$\beta_2 = a + b\sqrt{r} - c\sqrt{s} - e\sqrt{rs}, \tag{78}$$

and

$$\beta_3 = a - b\sqrt{r} - c\sqrt{s} + e\sqrt{rs}. \tag{79}$$

The quotient of a target element α by β is calculated as

$$\frac{\alpha}{\beta} = \frac{\alpha\beta_1\beta_2\beta_3}{\beta\beta_1\beta_2\beta_3} = \frac{\alpha\beta_1\beta_2\beta_3}{N(\beta)}. \tag{80}$$

The candidate β divides α if

$$N(\beta) \neq 0 \tag{81}$$

and every coefficient of the numerator in (80) is divisible by $N(\beta)$. This condition ensures that the quotient remains in the same biquadratic integer ring.

G. Goal Function and Output

A complete candidate is accepted only if it satisfies

$$\beta \mid \alpha, \tag{82}$$

$$\beta \text{ is not a unit}, \tag{83}$$

and β is not associate to the target element. Associates are excluded because they produce only trivial factorizations.

H. Search Procedure

The overall procedure can be summarized as follows.

1) Receive a target element α and identify its ring type.
2) Determine the number of coefficients required by the ring.
3) Start from an empty coefficient vector.
4) Generate one coefficient value at a time in depth-first order.
5) Construct a partial candidate by assigning the generated coefficients and setting all remaining coefficients to zero.
6) Calculate the norm of the partial candidate.
7) Stop the current positive or negative coefficient sequence when the norm exceeds the norm of the target element.
8) When all coefficients have been assigned, test whether the candidate divides the target element exactly.
9) Reject the candidate if it is a unit or an associate of the target element.
10) Store every remaining candidate as a valid nontrivial divisor.

IV. Implementation

The implementation of this algorithm is made in golang and is separated into 3 main part which is main, factorization, and ring.

A. Ring.go

This file represent the integer ring as an interface which then will be implemented by QuadraticRing and BiquadraticIntegerRing. Operation like divisibility test, multiplication, norm, unit checking will be done here. All interaction with ring will be made trough this interface so it can be extended to for any other quadratic integer ring extension like triquadratic integer ring.

```
package main

import "fmt"

type IntegerRing interface {
    Norm() int
    CanDivide(divided IntegerRing) bool
    Divide(divided IntegerRing) (IntegerRing, bool)
    CoefficientCount() int
    FromCoefficients(coefficients []int) IntegerRing
    IsUnit() bool
    IsAssociate(other IntegerRing) bool
    print()
}
type BiquadraticIntegerRing struct {
    a        int
    b        int
    radical1 int //the first radical part
    c        int
    radical2 int //the second radical part
    d        int
    radical3 int //the last radical part
}
type QuadraticIntegerRing struct {
    a       int
    b       int
    radical int //the radical part
}

func (u QuadraticIntegerRing) Norm() int {
    return u.a*u.a - u.radical*(u.b*u.b)
}
func (this QuadraticIntegerRing) multiply(other QuadraticIntegerRing)
      QuadraticIntegerRing {
    return QuadraticIntegerRing{
        a:       this.a*other.a + this.b*other.b*this.radical,
        b:       this.a*other.b + this.b*other.a,
        radical: this.radical,
    }
}
func (this QuadraticIntegerRing) CanDivide(divided IntegerRing) bool {
    _, ok := this.Divide(divided)
    return ok
}

func (this QuadraticIntegerRing) Divide(divided IntegerRing) (IntegerRing, bool) {
    var dividedRing QuadraticIntegerRing
    switch value := divided.(type) {
    case QuadraticIntegerRing:
        dividedRing = value
    case *QuadraticIntegerRing:
        if value == nil {
            return nil, false
        }
        dividedRing = *value
    default:
        return nil, false
    }

    if this.radical != dividedRing.radical {
        return nil, false
    }
    norm := this.Norm()
    if norm == 0 {
        return nil, false
    }
    conjugate := QuadraticIntegerRing{
        a:       this.a,
        b:       -this.b,
        radical: this.radical,
    }
    multiplicated := dividedRing.multiply(conjugate)
    if multiplicated.a%norm == 0 && multiplicated.b%norm == 0 {
        return QuadraticIntegerRing{
            a:       multiplicated.a / norm,
            b:       multiplicated.b / norm,
            radical: this.radical,
        }, true
    }
    return nil, false
}

func (this QuadraticIntegerRing) CoefficientCount() int {
    return 2
}

func (this QuadraticIntegerRing) FromCoefficients(coefficients []int) IntegerRing {
    candidate := QuadraticIntegerRing{radical: this.radical}
    if len(coefficients) > 0 {
        candidate.a = coefficients[0]
    }
    if len(coefficients) > 1 {
        candidate.b = coefficients[1]
    }
    return candidate
}

func (this QuadraticIntegerRing) IsUnit() bool {
    norm := this.Norm()
    return norm == 1 || norm == -1
}

func (this QuadraticIntegerRing) IsAssociate(other IntegerRing) bool {
    if other == nil {
        return false
    }
    switch value := other.(type) {
    case QuadraticIntegerRing:
        if this.a == value.a && this.b == value.b && this.radical == value.radical {
            return true
        }
    case *QuadraticIntegerRing:
        if value != nil && this.a == value.a && this.b == value.b && this.radical ==
        value.radical {
            return true
        }
    }
    return this.CanDivide(other) && other.CanDivide(this)
}

func (this QuadraticIntegerRing) print() {
    printed := false
    if this.a != 0 {
        fmt.Print(this.a)
        printed = true
    }
    if this.b != 0 {
        if printed && this.b > 0 {
            fmt.Print(" + ")
        } else if printed {
            fmt.Print(" - ")
        } else if this.b < 0 {
            fmt.Print("-")
        }

        coefficient := this.b
```

```
        if coefficient < 0 {
            coefficient = -coefficient
        }
        fmt.Printf("%d(%d)", coefficient, this.radical)
        printed = true
    }
    if !printed {
        fmt.Print(0)
    }
    fmt.Println()
}

func (u BiquadraticIntegerRing) Norm() int {
    normPartA := u.a*u.a + u.radical1*(u.b*u.b) - u.radical2*(u.c*u.c) - u.radical3
       *(u.d*u.d)
    normPartB := 2*u.a*u.b - 2*u.radical2*u.c*u.d
    return normPartA*normPartA - u.radical1*(normPartB*normPartB)
}

func (this BiquadraticIntegerRing) multiply(other BiquadraticIntegerRing)
       BiquadraticIntegerRing {
    return BiquadraticIntegerRing{
        a:        this.a*other.a + this.b*other.b*this.radical1 + this.c*other.c*
       this.radical2 + this.d*other.d*this.radical3,
        b:        this.a*other.b + this.b*other.a + this.c*other.d*this.radical2 +
       this.d*other.c*this.radical2,
        radical1: this.radical1,
        c:        this.a*other.c + this.c*other.a + this.b*other.d*this.radical1 +
       this.d*other.b*this.radical1,
        radical2: this.radical2,
        d:        this.a*other.d + this.d*other.a + this.b*other.c + this.c*other.b,
        radical3: this.radical3,
    }
}

func (this BiquadraticIntegerRing) CanDivide(divided IntegerRing) bool {
    _, ok := this.Divide(divided)
    return ok
}

func (this BiquadraticIntegerRing) Divide(divided IntegerRing) (IntegerRing, bool) {
    var dividedRing BiquadraticIntegerRing
    switch value := divided.(type) {
    case BiquadraticIntegerRing:
        dividedRing = value
    case *BiquadraticIntegerRing:
        if value == nil {
            return nil, false
        }
        dividedRing = *value
    default:
        return nil, false
    }

    if this.radical1 != dividedRing.radical1 || this.radical2 != dividedRing.
       radical2 || this.radical3 != dividedRing.radical3 {
        return nil, false
    }

    norm := this.Norm()
    if norm == 0 {
        return nil, false
    }

    conjugate1 := BiquadraticIntegerRing{
        a:        this.a,
        b:        -this.b,
        radical1: this.radical1,
        c:        this.c,
        radical2: this.radical2,
        d:        -this.d,
        radical3: this.radical3,
    }
    conjugate2 := BiquadraticIntegerRing{
        a:        this.a,
        b:        this.b,
        radical1: this.radical1,
        c:        -this.c,
        radical2: this.radical2,
        d:        -this.d,
        radical3: this.radical3,
    }
    conjugate3 := BiquadraticIntegerRing{
        a:        this.a,
        b:        -this.b,
        radical1: this.radical1,
        c:        -this.c,
        radical2: this.radical2,
        d:        this.d,
        radical3: this.radical3,
    }

    multiplicated := dividedRing.multiply(conjugate1).multiply(conjugate2).multiply(
       conjugate3)
    if multiplicated.a%norm == 0 && multiplicated.b%norm == 0 && multiplicated.c%
       norm == 0 && multiplicated.d%norm == 0 {
        return BiquadraticIntegerRing{
            a:        multiplicated.a / norm,
            b:        multiplicated.b / norm,
            radical1: this.radical1,
            c:        multiplicated.c / norm,
            radical2: this.radical2,
            d:        multiplicated.d / norm,
            radical3: this.radical3,
        }, true
    }
    return nil, false
}

func (this BiquadraticIntegerRing) CoefficientCount() int {
    return 4
}

func (this BiquadraticIntegerRing) FromCoefficients(coefficients []int) IntegerRing
       {
    candidate := BiquadraticIntegerRing{
        radical1: this.radical1,
        radical2: this.radical2,
        radical3: this.radical3,
    }
    if len(coefficients) > 0 {
        candidate.a = coefficients[0]
    }
    if len(coefficients) > 1 {
        candidate.b = coefficients[1]
    }
    if len(coefficients) > 2 {
        candidate.c = coefficients[2]
    }
    if len(coefficients) > 3 {
        candidate.d = coefficients[3]
    }
    return candidate
}

func (this BiquadraticIntegerRing) IsUnit() bool {
    norm := this.Norm()
    return norm == 1 || norm == -1
}

func (this BiquadraticIntegerRing) IsAssociate(other IntegerRing) bool {
    if other == nil {
        return false
    }
    switch value := other.(type) {
    case BiquadraticIntegerRing:
        if this.a == value.a && this.b == value.b && this.radical1 == value.radical1
        && this.c == value.c && this.radical2 == value.radical2 && this.d == value.d
        && this.radical3 == value.radical3 {
            return true
        }
    case *BiquadraticIntegerRing:
        if value != nil && this.a == value.a && this.b == value.b && this.radical1
       == value.radical1 && this.c == value.c && this.radical2 == value.radical2 &&
       this.d == value.d && this.radical3 == value.radical3 {
            return true
        }
    }
    return this.CanDivide(other) && other.CanDivide(this)
}

func (this BiquadraticIntegerRing) print() {
    printed := false
    if this.a != 0 {
        fmt.Print(this.a)
        printed = true
    }
    if this.b != 0 {
        if printed && this.b > 0 {
            fmt.Print(" + ")
        } else if printed {
            fmt.Print(" - ")
        } else if this.b < 0 {
            fmt.Print("-")
        }

        coefficient := this.b
        if coefficient < 0 {
            coefficient = -coefficient
        }
        fmt.Printf("%d(%d)", coefficient, this.radical1)
        printed = true
    }
    if this.c != 0 {
        if printed && this.c > 0 {
            fmt.Print(" + ")
        } else if printed {
            fmt.Print(" - ")
        } else if this.c < 0 {
            fmt.Print("-")
        }

        coefficient := this.c
        if coefficient < 0 {
            coefficient = -coefficient
        }
        fmt.Printf("%d(%d)", coefficient, this.radical2)
        printed = true
    }
    if this.d != 0 {
        if printed && this.d > 0 {
            fmt.Print(" + ")
        } else if printed {
            fmt.Print(" - ")
        } else if this.d < 0 {
            fmt.Print("-")
        }

        coefficient := this.d
        if coefficient < 0 {
            coefficient = -coefficient
        }
        fmt.Printf("%d(%d)", coefficient, this.radical3)
        printed = true
    }
    if !printed {
        fmt.Print(0)
    }
    fmt.Println()
}
```

Listing 1. ring.go

B. *factorization.go*

This file contains the factorization process as discussed before. including the boundary function, and goal function

```
package main

func B(target, candidate IntegerRing) bool { //boundary function
    return candidate.Norm() > target.Norm()
}
func goal(target, candidate IntegerRing) bool { //goal function
    return candidate.CanDivide(target) && !candidate.IsUnit() && !candidate.
       IsAssociate(target)
}
func factorize(target IntegerRing, CandidateCoeff []int) (solutions []IntegerRing,
       found bool) {
    if len(CandidateCoeff) >= target.CoefficientCount() {
        return nil, false
    }
    i := 0
    for {
        newcandidateCoeff := append(CandidateCoeff, i)
        candidate := target.FromCoefficients(newcandidateCoeff)
        if B(target, candidate) {
            break
        }
        if len(newcandidateCoeff) == target.CoefficientCount() && goal(target,
       candidate) {
            solutions = append(solutions, candidate)
            found = true
        }
        value, founded := factorize(target, newcandidateCoeff)
        if founded {
            found = true
            solutions = append(solutions, value...)
        }
        i++
    }
    i = -1
    for {
        newcandidateCoeff := append(CandidateCoeff, i)
        candidate := target.FromCoefficients(newcandidateCoeff)
        if B(target, candidate) {
            break
        }
        if len(newcandidateCoeff) == target.CoefficientCount() && goal(target,
       candidate) {
            solutions = append(solutions, candidate)
            found = true
        }
        value, founded := factorize(target, newcandidateCoeff)
        if founded {
            found = true
            solutions = append(solutions, value...)
        }
        i--
    }
    return
}
```

Listing 2. factorization.go

C. *main.go*

This file contain the user interaction part where it take inputs from user and output the result back

```
package main

import "fmt"

func main() {
    fmt.Println("Integer ring factorizer")
    fmt.Println("1. Quadratic integer ring")
    fmt.Println("2. Biquadratic integer ring")
    fmt.Print("Choose ring: ")

    var choice int
    if _, err := fmt.Scan(&choice); err != nil {
        fmt.Println("Invalid input")
        return
    }

    var ring IntegerRing
    switch choice {
    case 1:
        var a int
        var b int
        var radical int

        fmt.Println("Form: a + b(radical)")
        fmt.Print("a: ")
        fmt.Scan(&a)
        fmt.Print("b: ")
        fmt.Scan(&b)
        fmt.Print("radical: ")
        fmt.Scan(&radical)

        ring = QuadraticIntegerRing{
            a:       a,
            b:       b,
            radical: radical,
        }
    case 2:
        var a int
        var b int
        var c int
        var d int
        var radical1 int
        var radical2 int

        fmt.Println("Form: a + b(radical1) + c(radical2) + d(radical1*radical2)")
        fmt.Print("a: ")
        fmt.Scan(&a)
        fmt.Print("b: ")
        fmt.Scan(&b)
        fmt.Print("radical1: ")
        fmt.Scan(&radical1)
        fmt.Print("c: ")
        fmt.Scan(&c)
        fmt.Print("radical2: ")
        fmt.Scan(&radical2)
        fmt.Print("d: ")
        fmt.Scan(&d)

        ring = BiquadraticIntegerRing{
            a:        a,
            b:        b,
            radical1: radical1,
            c:        c,
            radical2: radical2,
            d:        d,
            radical3: radical1 * radical2,
        }
    default:
        fmt.Println("Unknown ring choice")
        return
    }

    fmt.Print("Input: ")
    ring.print()

    if factors, found := factorize(ring, []int{}); found {
        fmt.Println("Factorization:")
        for _, factor := range factors {
            fmt.Print("Factor: ")
            factor.print()
        }
    } else {
        fmt.Println("Factor not found")
    }
}
```

Listing 3. main.go

V. Testing

A. *Test Case 1:* $\mathbb{Z}[\sqrt{-5}]$

The target element is

$$\alpha = 6 + 0\sqrt{-5}. \tag{84}$$

The expected factorizations are

$$6 = 2 \cdot 3 \tag{85}$$

and

$$6 = (1 + \sqrt{-5})(1 - \sqrt{-5}). \tag{86}$$

Therefore, the expected nontrivial factors are

$$\begin{aligned} &2, \quad -2, \quad 3, \quad -3, \\ &1 + \sqrt{-5}, \quad -1 - \sqrt{-5}, \\ &1 - \sqrt{-5}, \quad -1 + \sqrt{-5}. \end{aligned} \tag{87}$$

Result:

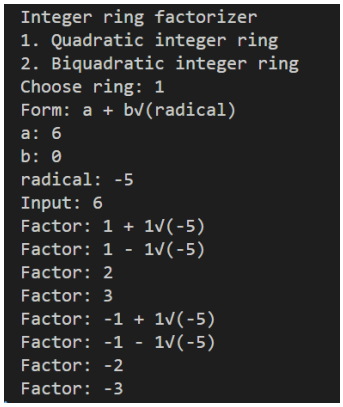

Fig. 1. Test case 1 result

B. Test Case 2: $\mathbb{Z}[\sqrt{-1}]$

The target element is

$$\alpha = 2 + 0\sqrt{-1}. \tag{88}$$

The expected factorization is

$$2 = (1 + \sqrt{-1})(1 - \sqrt{-1}). \tag{89}$$

Therefore, the expected nontrivial factors are

$$1 + \sqrt{-1}, \quad 1 - \sqrt{-1}, \quad -1 + \sqrt{-1}, \quad -1 - \sqrt{-1}, \\ 2\sqrt{-1}, \quad -2\sqrt{-1}, \quad 2, \quad -2. \tag{90}$$

Result:

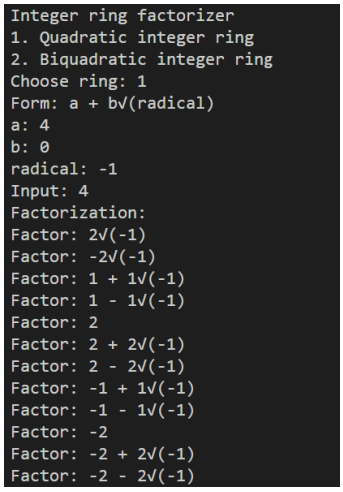

Fig. 2. Test case 2 result

C. Test Case 3: $\mathbb{Z}[\sqrt{-5}]$

The target element is

$$\alpha = 2 + 2\sqrt{-5}. \tag{91}$$

The expected factorization is

$$2 + 2\sqrt{-5} = 2(1 + \sqrt{-5}). \tag{92}$$

Therefore, the expected nontrivial factors are

$$2, \quad -2, \quad 1 + \sqrt{-5}, \quad -1 - \sqrt{-5}. \tag{93}$$

Result:

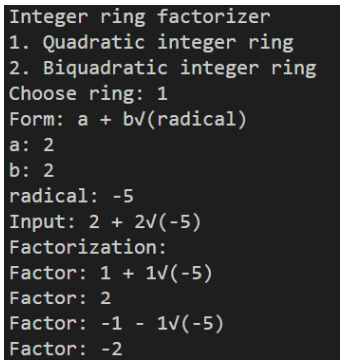

Fig. 3. Test case 3 result

D. Test Case 4: $\mathbb{Z}[\sqrt{-2}, \sqrt{-5}]$

The radical parameters are

$$r_1 = -2, \qquad r_2 = -5, \qquad r_3 = 10. \tag{94}$$

The target element is

$$\alpha = 2 + 0\sqrt{-2} + 0\sqrt{-5} + 0\sqrt{10}. \tag{95}$$

The expected factorization is

$$2 = \sqrt{-2}(-\sqrt{-2}). \tag{96}$$

Therefore, the expected nontrivial factors are

$$\sqrt{-2}, \qquad -\sqrt{-2}. \tag{97}$$

Result:

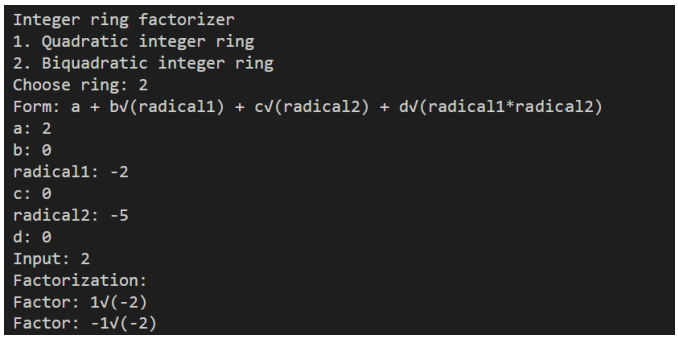

Fig. 4. Test case 4 result

E. Analysis

It's shown that the factorization is able to find the non trivial factors of given quadratic integer ring and biquadratic integer ring. also shown that some of the result is an associate of other result, this is intentional to faster the search because extra checking will be needed to remove it even though it technically is one of the factors so the result is still true.

VI. Conclusion

In this paper, we have demonstrated that the backtracking algorithm provides a systematic approach for searching nontrivial divisors in quadratic and biquadratic integer rings. By representing a candidate factor as a vector of integer coefficients, the method successfully constructs possible factors one coefficient at a time. The search process applies the general backtracking concept through a candidate generation function, a state-space tree, and a bounding function based on the norm of the candidate element.

The implementation through a Go program proves that this concept is applicable to quadratic integer rings of the form $\mathbb{Z}[\sqrt{d}]$ with negative radical d, as well as biquadratic integer rings of the form $\mathbb{Z}[\sqrt{r}, \sqrt{s}]$ with negative first and second radicals. The program shows that a candidate can be verified as a valid divisor by calculating its norm and testing whether the quotient remains in the same ring. The test cases also show that the algorithm can find expected nontrivial factors in several quadratic and biquadratic integer rings.

For quadratic integer rings with negative radicals, the norm provides a finite and complete search boundary because the norm increases as the absolute values of the coefficients increase. However, for biquadratic integer rings, the norm of a partial candidate is not always monotonic after additional coefficients are assigned. Therefore, the norm boundary in the current implementation is used as a practical restriction and does not guarantee that every possible divisor will be found.

This application demonstrates how the backtracking algorithm can be applied to an abstract algebra problem. Although the current program only searches for nontrivial divisors and does not recursively construct complete factorizations, it provides a foundation for further development. Future improvements may include recursive factorization of the quotient, removal of associate duplicates, and a more complete boundary strategy for biquadratic integer rings.

VII. Appendix

Github repository:Github

VIII. Acknowledgment

Praise be to God Almighty for His grace, the paper with the title "Implementation of Backtracking to Find Quadratic and Biquadratic Integer Ring's Factors" can be completed properly. The author would also like to thank the lecturer of K02 IF2211 Strategi Algoritma, Ir. Rila Mandala, M.Eng., Ph.D. for the knowledge that has been taught during lectures so that the author can complete this paper smoothly. In addition, the author also wants to thank everybody who has given their support to the author in completing this research paper.

References

[1] D. S. Dummit and R. M. Foote, *Abstract Algebra*, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, 2004.

[2] D. A. Marcus, *Number Fields*. New York, NY, USA: Springer-Verlag, 1977.

[3] MIT OpenCourseWare, "Factorization in Rings," RES.18-012 Algebra II Student Notes, Spring 2022. [Online]. Available: https://ocw.mit.edu/courses/res-18-012-algebra-ii-student-notes-spring-2022/mit18_702s22_lec12.tex. [Accessed: Jun. 19, 2026].

[4] R. Munir, "Algoritma Backtracking – Bagian 1," materi kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/15-Algoritma-backtracking-(2026)-Bagian1.pdf. [Accessed: Jun. 19, 2026].

IX. Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, june 19th 2026

Zeki Amani

NIM 13524082