HOCHSCHULE HAMM-LIPPSTADT

Implementierung eines Simplex-Algorithmus

Präsentiert von Jean Sokolov

Inhaltsverzeichnis

- Problemstellung
- Implementation in Rust
- Praktische Demonstration

PROBLEMSTELLUNG

Problemstellung

- Lineare Optimierung
- Standardminimierungsprobleme

$$\begin{array}{ll} \textbf{Minimize} & Z = 12x_1 + 16x_2 \\ \textbf{Subject to:} & x_1 + 2x_2 \geq 40 \\ & x_1 + x_2 \geq 30 \\ & x_1 \geq 0; x_2 \geq 0 \end{array}$$

- Simplex-Algorithmus in Rust

IMPLEMENTATION IN RUST

Datei einlesen

```rust
fn main() {
    // Read file
    let mut file_content: Vec<String> = read_file().unwrap();

    // separate objective function from rest of file content (constraints)
    let objective_function: &String = &file_content.clone()[0];
    file_content.remove(index: 0);
```

```rust
const DEFAULT_PATH: &'static str = "./target/release/KI_30.txt";

// Reading file and removing comments such as "// Objective function"
fn read_file() -> std::io::Result<Vec<String>> {
    let mut args: Args = std::env::args();
    let file_path: String = args.nth(1).unwrap_or_else(|| DEFAULT_PATH.to_string());
    let file: File = File::open(file_path)?;
    let reader: BufReader<File> = BufReader::new(inner: file);
    let mut tmp: Vec<String> = Vec::new();
    for lines: Result<String, Error> in reader.lines() {
        let tmpstr: String = lines.unwrap();
        if !tmpstr.contains("//") {
            tmp.push(tmpstr);
        }
    }
    return Ok(tmp);
}
```

Kostenfunktion parsen

```rust
    // parse file contents
    let mut objective_fn: Vec<f64> = generate_matrix_objective_fn(objective_function);
    let constraints: Vec<Vec<f64>> = generate_matrix_constraints(&mut file_content);

```

```rust
/// Iterates through the substrings, which are split of the objective function at '+'
/// Then splits substrings at '*' to get coefficient of variables
/// Parses and transforms the variables into a 64-bit float vector
/// e.g. min: + 3*x0 + 2*x1; to [3.0,2.0]
/// slack variables are added in function add_slack_variables_objective_fn
fn generate_matrix_objective_fn(objective_function: &String) -> Vec<f64> {
    let mut objective_fn: Vec<f64> = Vec::new();
    let variables: Vec<&str> = objective_function.split('+').collect();
    for s: &str in variables.into_iter().skip(1) {
        objective_fn.push(
            s.trim().split('*').collect::<Vec<&str>>()[0] &str
                .parse::<f64>() Result<f64, ParseFloatError>
                .unwrap(),
        );
    }
    return objective_fn;
}
```

Bedingungen parsen

```rust
/// Iterates through the file content, which contains the constraints
/// Parses and transforms the individual lines/strings into 64-bit float vectors
/// e.g. + 3*x0 + 2*x1 >= 12 to [3.0,2.0,12.0]
/// slack variables are added in function add_slack_variables_constraints
fn generate_matrix_constraints(file_content: &mut Vec<String>) -> Vec<Vec<f64>> {
    let mut constraints: Vec<Vec<f64>> = Vec::new();
    for s: &mut String in file_content {
        let tmp_vec_str: Vec<&str> = s.split('+').collect::<Vec<&str>>();
        let mut tmp_vec_f64: Vec<f64> = Vec::new();

        // grabbing lhs
        for t: &str in tmp_vec_str.clone().into_iter().skip(1) {
            tmp_vec_f64.push(
                t.trim().split('*').collect::<Vec<&str>>()[0] &str
                    .parse::<f64>() Result<f64, ParseFloatError>
                    .unwrap(),
            );
        }

        // grabbing rhs
        tmp_vec_f64.push(
            tmp_vec_str Vec<&str>
                .clone() Vec<&str>
                .last() Option<&&str>
                .unwrap() &&str
                .trim() &str
                .split(">=") Split<'_, &str>
                .collect::<Vec<&str>>()[1] &str
                .replace(from: ";", to: "") String
                .replace(from: " ", to: "") String
                .parse::<f64>() Result<f64, ParseFloatError>
                .unwrap(),
        );
        constraints.push(tmp_vec_f64);
    }

    return constraints;
} fn generate matrix constraints
```

Matrix anlegen und transponieren

```rust
    // transpose matrix with constraints and objective function
    let mut transposed_matrix: Vec<Vec<f64>> = constraints.clone();
    transposed_matrix.push(objective_fn);
    transposed_matrix = transpose_matrix(transposed_matrix);

    // grab transposed objective_fn
    objective_fn = transposed_matrix.pop().unwrap();

```

```rust
/// transposes the matrix consisting of constraints and objective function
/// this process turns a standard minimization problem into a standard maximization problem
fn transpose_matrix(m: Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    let mut t: Vec<Vec<f64>> = vec![Vec::with_capacity(m.len()); m[0].len()];
    for r: Vec<f64> in m {
        for i: usize in 0..r.len() {
            t[i].push(r[i]);
        }
    }
    t
}
```

Slackvariablen: Kostenfunktion

```rust
    // add slack variables to transposed matrix
    objective_fn = add_slack_variables_objective_fn(m: objective_fn, n: transposed_matrix.len());
    transposed_matrix = add_slack_variables_constraints(transposed_matrix);

    //println!("\n\nObjective f: {:?}", objective_fn);
    //println!("Constraints: {:?}", transposed_matrix);
```

```rust
/// adds slack variables to the objective function
fn add_slack_variables_objective_fn(m: Vec<f64>, n: usize) -> Vec<f64> {
    let mut t: Vec<f64> = m;
    // add n slack-variables to objective fn, n being the number of constraints
    for _i: usize in 0..n {
        t.push(0.0);
    }
    t
}

```

Slackvariablen: Bedingungen

```rust
/// adds slack variables to the constraints
fn add_slack_variables_constraints(m: Vec<Vec<f64>>) -> Vec<Vec<f64>> {
    let mut t: Vec<Vec<f64>> = m.clone();
    for index: usize in 0..m.len() {
        for _i: usize in 0..m.len() {
            // add n slack-variables to each constraint, n being the number of constraints
            t[index].push(0.0);
        }
        // swap initial rhs with last element of constraint, as rhs value is not the right-most value after appending the slacks
        t[index].swap(a: m[0].len() + m.len() - 1, b: m[0].len() - 1);
        // setting this constraint's slack variable to 1
        t[index][m[0].len() + index - 1] = 1.0;
    }
    t
}
```

Ausführen des Algorithmus

```
    // Run simplex algorithm
    solve(objective_fn, constraints: transposed_matrix);

```

```
fn solve(mut objective_fn: Vec<f64>, mut constraints: Vec<Vec<f64>>) {
    let mut i: usize = 0;
    let mut cost: f64 = 0.0;
    let mut init_cost: f64 = INFINITY;
    let mut init_obj_fn: Vec<f64> = objective_fn.clone();
    loop {
        // find most promising variable, ignoring variables with value of 0
        let mut tmp_obj_fn: Vec<f64> = objective_fn.clone();
        for n: usize in 0..tmp_obj_fn.len() {
            if tmp_obj_fn[n].eq(&0.0) {
                tmp_obj_fn[n] = -INFINITY;
            }
        }

        let index_max_factor: usize = (tmp_obj_fn Vec<f64>
            .iter() Iter<'_, f64>
            .enumerate() impl Iterator<Item = (usize, …)>
            .max_by(compare: |(_, a: &&f64), (_, b: &&f64)| a.total_cmp(b)) Option<(usize, &f64)>
            .map(|(index: usize, _)| index)) Option<usize>
        .unwrap();
```

Finden der limitierensten Bedingung

```rust
// find most limiting constraint
let mut lhs: Vec<f64> = Vec::new();
let mut rhs: Vec<f64> = Vec::new();
for constraint: Vec<f64> in constraints.clone() {
    lhs.push(constraint[index_max_factor]);
    rhs.push(constraint.last().unwrap().to_owned());
}
let mut n: usize = 0;

// iterate through the this iteration's constraints,
// dividing rhs value by coefficient of this iteration's variable e.g. 3*x0=12 to x0=4
// while doing so, check that the variables respect the non-negative constraint
while n < lhs.len() {
    //println!("Pre Calc {:?}={:?}", lhs[n], rhs[n]);
    if lhs[n].ge(&0.0) {
        if rhs[n].ge(&0.0) {
            rhs[n] = rhs[n] / lhs[n];
            lhs[n] = 1.0;
        } else {
            rhs[n] = INFINITY;
            lhs[n] = INFINITY;
        }
    } else {
        if rhs[n].ge(&0.0) {
            rhs[n] = INFINITY;
            lhs[n] = INFINITY;
        } else {
            rhs[n] = rhs[n] / lhs[n];
            lhs[n] = 1.0;
        }
    }
    //println!("Post Calc {:?}={:?}", lhs[n], rhs[n]);
    n += 1;
}
// determine index/row of most restrictive constraint for most significant variable in objective function of this iteration
let most_significant_constraint_index: usize = (rhs Vec<f64>
    .iter() Iter<'_, f64>
    .enumerate() impl Iterator<Item = (usize, …)>
    .min_by(compare: |(_, a: &&f64), (_, b: &&f64)| a.total_cmp(b)) Option<(usize, &f64)>
    .map(|(index: usize, _)| index)) Option<usize>
.unwrap();
// grab value/vector behind determined index
let mut most_significant_constraint: &mut Vec<f64> = &mut constraints[most_significant_constraint_index];
```

Limitierenste Bedingung umformen

```rust
        let x: Vec<f64> = transform_equation(&mut most_significant_constraint, current_index: index_max_factor);
```

```rust
/// Transformes an equation/singular constraint to have the lhs be a singular variable, which then can be inserted into the other constraints and objective function
/// e.g. + 3*x0 + 2*x1 + s1 = 12 to x0 = 12 - 2/3*x1 - 1/3*s1
/// or rather as matrix:
/// [3.0,2.0,1.0,12.0] to [1.0,-0.66667,-0,33333,-12.0]
/// the rhs value can be inverted aswell, as upon insertion (if rhs>=0) you end up with a positive value on the lhs, which has to be subtracted from the rhs value anyways
fn transform_equation(
    most_significant_constraint: &mut Vec<f64>,
    current_index: usize,
) -> Vec<f64> {
    let mut transformed_equation: Vec<f64> = Vec::new();
    for mut values: f64 in most_significant_constraint.clone() {
        values = values / most_significant_constraint[current_index] * -1.0;
        transformed_equation.push(values);
    }

    return transformed_equation;
}
```

Neuaufstellen des Gleichungssystem

```
// insert transformed equation/value of currently selected variable into all constraints other than currently selected constraint/row
// for currently selected constraint: normalize values, so that coefficient of currently selected variable is 1
// e.g. + 3*x0 + 2*x1 + 1*s1 = 12 to + x0 + 2/3*x1 + 1/3*s1 = 4
for j: usize in 0..lhs.len() {
    let mult: f64 = constraints[j][index_max_factor];
    if j.ne(&most_significant_constraint_index) {
        for k: usize in 0..constraints[j].len() {
            constraints[j][k] += x[k] * mult;
        }
        constraints[j][index_max_factor] = 0.0;
    } else {
        for k: usize in 0..constraints[j].len() {
            constraints[j][k] /= mult;
        }
    }
}

let mult: f64 = objective_fn[index_max_factor];

// insert transformed equation/value of currently selected variable into objective function
for k: usize in 0..x.len() {
    if k.ne(&x.len().checked_sub(1).unwrap()) {
        objective_fn[k] += x[k] * mult;
    } else {
        cost += x.last().unwrap() * mult;
    }
}
objective_fn[index_max_factor] = 0.0;
```

Stopbedingung

```rust
        println!("Initial rhs of objective fn {:?}, changed to {:?} with this iteration. Value changed by: {:?}", init_cost,cost, cost-init_cost);
        // if stop condition (minimization -> cost of objective function increasing) is triggered, print the values of the objective function and variables that were calculated
        if cost.gt(&init_cost) {
            println!(
                "\n________________________________\n\n\nOptimal solution found.\nVariables are:\n"
            );
            // grab all non-slack-variables
            let mut variables: Vec<f64> = vec![0.0; objective_fn.len()];
            for n: usize in 0..init_obj_fn.len() {
                if n.ge(&(init_obj_fn.len() - constraints.len())) {
                    variables[n] = init_obj_fn[n];
                }
            }
            // print all variables with a value, writing their name and value to the output/console
            for n: (usize, &f64) in variables.iter().enumerate() {
                if n.1.ne(&0.0) {
                    println!(
                        "x{} = {}",
                        n.0 - (init_obj_fn.len() - constraints.len()),
                        n.1.abs()
                    );
                }
            }
            // print final cost of objective function
            println!("p = {}", init_cost.abs());
            break;
        }
        //println!("\nNEW OBJECTIVE FUNCTION: {:?}= {cost}", objective_fn);
        init_cost = cost;
        init_obj_fn = objective_fn.clone();
        i += 1;
    }
    println!("\nDone after {} iterations", i + 1);
} fn solve
```

Quellcode

- https://github.com/JeanSokolov/sat_solver_v2/blob/master/src/main.rs#L275

PRAKTISCHE DEMONSTRATION

Vielen Dank für Ihre Aufmerksamkeit