# Parallel Branch-and-Cut Integer Program Solver

Weixi Zhuo (wz2603)
(ParBnC)
*Columbia University*

(Dated: 21 December 2022)

## I. OVERVIEW

This project aims to implement a parallel Haskell program that solves general integer linear programs (ILP) using the branch-and-cut algorithm[1]. We shall implement both sequential and parallel versions and compare their run-time performances. The source project can be found on my github[2].

## II. INTRODUCTION

### A. Background

It is known that general integer linear programming problems (ILP) are NP-hard. A maximization ILP without mixing constraints can be formulated in the following mathematical notations.

$$
\begin{aligned}
\max\ & c^\top x \\
\text{s.t.} & Ax \leq b \\
& x \geq 0, x_i \in \mathbb{Z}, \forall i \\
& b \geq 0
\end{aligned}
$$

where $A$ and $b$ describe the linear constraints on variable vector $x$ and $c$ represents the cost/reward on each variable. To obtain heuristic-based integral solutions of an ILP, branch-and-bound search algorithm[3] was developed. This can already benefit from parallelism as different branches can be processed separately[4]. To make branch-and-bound more efficient, branch-and-cut algorithm[1] introduces a better method to fathom subproblems by including Gomory's cut[5] constraints. This is also the standard way to solve mixed-integer programs (MIP) in most solvers, which is how GLPK solves ILP in particular.

### B. Linear-Relaxed Problems

One of the most repeated tasks in this project is the process of solving linear subprogram by relaxing the integral constraints on the original ILP. This is not well supported in Haskell as there is no available module that can solve linear programs without calling an external solver. Moreover, external solver usually only provides solution and optimal value without a crucial intermediate simplex tableau, which is the source of cutting plane computations. We implement a two-phase simplex LP solver in native Haskell for this project to enable the branch-and-cut algorithm.

1 (*Initialization*): Set $L = \{\mathrm{IP}^0\}$, $\bar{z}_0 = +\infty$, and $\underline{z}_{ip} = \infty$.

2 (*Termination*): If $L = \emptyset$, then the solution $x^*$ which yielded the incumbent objective value $\underline{z}_{ip}$ is optimal. If no such $x^*$ exists (i.e., $\underline{z}_{ip} = -\infty$), then (IP) is infeasible.

3 (*Problem selection and relaxation*): Select and delete a problem $\mathrm{IP}^i$ from $L$. Solve a relaxation of $\mathrm{IP}^i$. Let $z_i^R$ denote the optimal objective value of the relaxation, and let $x^{iR}$ be an optimal solution if one exists. (Thus, $z_i^R = c^\top x^{iR}$, or $z_i^R = -\infty$.)

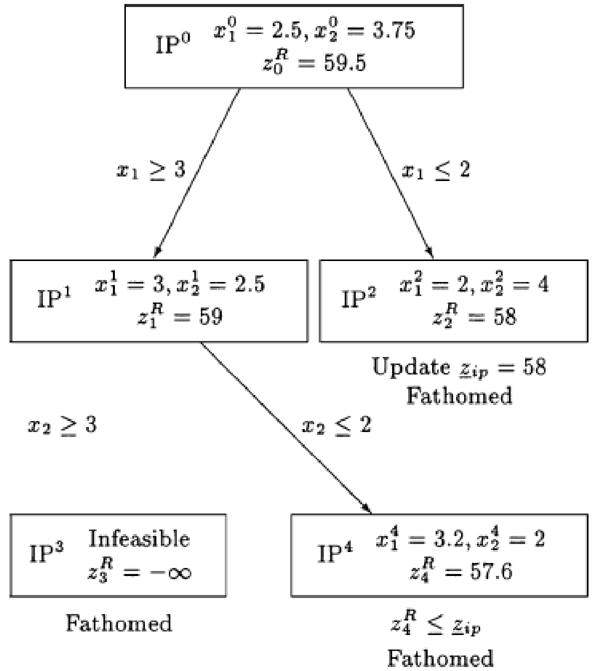
4 (*Fathoming and Pruning*):

i) If $z_i^R \leq \underline{z}_{ip}$ go to Step 2.

ii) If $z_i^R > \underline{z}_{ip}$ and $x^{iR}$ is integral feasible, update $\underline{z}_{ip} = z_i^R$. Delete from $L$ all problems with $\bar{z}_i \leq \underline{z}_{ip}$. Go to Step 2.

5 (*Partitioning*): Let $\{S^{ij}\}_{j=1}^{j=k}$ be a partition of the constraint set $S^i$ of the problem $\mathrm{IP}^i$. Add problems $\{\mathrm{IP}^{ij}\}_{j=1}^{j=k}$ to $L$, where $\mathrm{IP}^{ij}$ is $\mathrm{IP}^i$ with feasible region restricted to $S^{ij}$ and $\bar{z}_{ij} = z_i^R$ for $j = 1, \ldots, k$.
Go to Step 2.

(a) Branch and bound algorithm[3]




(b) Branch and bound example on two integer variables[3]

## C. Branch-and-Cut Algorithm

The branch-and-cut algorithm is a hybrid method between branch-and-bound and cutting-plane methods. In a typical branch-and-bound algorithm, binary branching constraints are added to the relaxed linear program to force the simplex solver to consider integral solutions. Such a branching process terminates when the solution is entirely integral, thus, resulting in a leaf node. Overall, a branch-and-bound algorithm is a binary tree model. The general branch-and-bound algorithm is shown in figure 1a along with a search tree example in figure 1b. On the other hand, the cut that we are interested in is Gomory's cut[5], which is adding a cutting plane constraint on the non-integral solution so that the linear program does not lose feasible integral points while reducing the feasible region to make the process converge. Individually, both approaches deteriorate quickly as the dimension of the program grows, number of variables and number of constraints. Branch-and-bound potentially can yield $\mathcal{O}(2^N)$ nodes where $N$ is the number of variables and Gomory's cut can lead to numerical errors when applied repetitively. For our project, the branch-and-cut algorithm performs Gomory's cut to reduce the feasible region and branch on an non-integral variable to create a branch-and-bound tree. The algorithm is described in Algorithm:1

**Algorithm 1** Construct Branch-and-Cut Tree

**function** BnC$(A, b, c)$ ▷ some integral threshold $\epsilon$
    $(x, T, v) \leftarrow \text{simplex}(A, b, c)$ ▷ generate solution, a tableau, and objective value
    **if** inIntegral$(x)$ **then**
        **return** Node$((x, v), \text{Nil}, \text{Nil})$
    **else**
        $T' \leftarrow \text{addGomoryCut}(T)$
        $(A_l, b_l, c_l), (A_r, b_r, c_r) \leftarrow \text{getBranches}(T')$
        **return** Node$((x, v), \text{BnC}(A_l, b_l, c_l), \text{BnC}(A_r, b_r, c_r))$
    **end if**
**end function**

## III. SEQUENTIAL IMPLEMENTATION

### A. Two-Phase Simplex Algorithm

The most naive simplex method is fact quite easy to implement with the assumption that $b \geq 0$. However, a single-phase simplex method will fail if we have mixed constraints with some $b_j < 0$. As we were implementing the solver, it is inevitable to have such scenario since for each branching process, we need to include $x_i \leq \lfloor x_i^* \rfloor$ and $x_i \leq \lceil x_i^* \rceil$ as branching constraints to the left and right branches respectively, where $x^*$ is the current non-integral solution. Two-phase simplex algorithm is quite robust in this context. The first phase manipulate the simplex tableau such that it only has non-negative bounds $b$ and check for potential program infeasibility to fathom the branch early, this is primarily handled by `updateMixedTab`.

```
%-- update tableau till ready for naive simplex
updateMixedTab :: LA.Matrix R -> Int -> LA.Matrix R
updateMixedTab tab counter
    | not $ isImprovableMixed tab = tab
    | counter == 0 = LA.fromLists [[]]
    | otherwise = updateMixedTab newTab (counter - 1) where
        pivotPos = getPivotPositionMixed tab
        newTab = pivotStep tab pivotPos
```

If feasible, we proceed to phase two by applying a naive simplex algorithm handled by `updateTab`. For the purpose of this project, we assume `ObjectiveType` to be `Maximization` by default but it can be easily changed by maximizing the negative cost in case of a minimization problem.

```
%-- update tableau till a solution found
updateTab :: ObjectiveType -> LA.Matrix R -> Int -> LA.Matrix R
updateTab obj tab counter
    | not $ isImprovable obj tab = tab
    | counter == 0 = LA.fromLists [[]]
    | otherwise = updateTab obj newTab (counter - 1) where
        pivotPos = getPivotPosition obj tab
```

```
        newTab = pivotStep tab pivotPos
```

Furthermore, we store simplex tableau in hmatrix's matrices and solutions in vectors.

### B. Gomory's Cut

For a given simplex tableau, we select a non-integral variable's corresponding pivot row. Then, construct an additional constraint following Gomory's cut approach. Our implementation follows the procedure shown by Dr. Shokoufeh Mirzaei[6]. It is implemented by the following function.

```
%
getGomoryCut :: Vector R -> Vector R
getGomoryCut rowVec = gomoryCons where
    [varPart, constPart] = takesV [LA.size rowVec - 1, 1] rowVec
    posDec num = -posFrac where
        (intPart, fracPart) = properFraction num
        posFrac = if fracPart >= 0.0 then fracPart else 1.0 + fracPart
    gomoryCons = vjoin [cmap posDec varPart, vector [1::R], cmap posDec constPart]
```

### C. Branch-and-Cut Algorithm

We construct the branch-and-cut tree and fathom infeasible branches following the same logic in Algorithm:1. In particular, `maxDepth` controls the depths of the tree to construct. For low-dimension problem with less than 10 variables, we can set it to the same number of variables. But for high-dimensions, the performance still deteriorates and it might be preferable to set a reasonable depth and obtain a candidate solution instead of the global optimal solution.

```
%
constructBranchAndCut :: ObjectiveType -> Matrix R -> Vector Bool -> Vector R -> Int -> Tree BranchProblem
constructBranchAndCut obj tab intMask costVec maxDepth
    | maxDepth == 0 = Nil
    | infeasible = Nil
    | and $ integerSolved intList solMask = currProb Nil Nil
    | otherwise = currProb leftTree rightTree where
        (y, currSol, newTab) = simplexWithMixedTab obj tab
        infeasible = y == (-infinity)
        candVal = costVec <.> LA.subVector 0 (LA.size costVec) currSol
        currProb = Node BranchProblem {
           solution = currSol, value = candVal
        }
        intList = LA.toList intMask
        solMask =  map (isInt . roundSolution) $ LA.toList currSol
```

```
        nextIdx = findNonIntIndex intList solMask
        cutTab = addGomoryCut tab $ getGomoryCut $ newTab ! nextIdx
        (leftTab, rightTab) = getBranches cutTab currSol nextIdx
        leftTree = constructBranchAndCut obj leftTab intMask costVec (maxDepth - 1)
        rightTree = constructBranchAndCut obj rightTab intMask costVec (maxDepth - 1)
```

### D. Tree Solution Search

After the tree is constructed, we traverse the tree to prune and search for final optimal solution.

```
%
searchBBTreeMax :: Tree BranchProblem -> BranchProblem
searchBBTreeMax Nil = BranchProblem{solution = LA.fromList [], value = -infinity}
searchBBTreeMax (Node bp Nil Nil) = bp
searchBBTreeMax (Node bp leftBpNode Nil) = searchBBTreeMax leftBpNode
searchBBTreeMax (Node bp Nil rightBpNode) = searchBBTreeMax rightBpNode
searchBBTreeMax (Node bp leftBpNode rightBpNode)
    | leftBpVal > rightBpVal = leftBp
    | otherwise = rightBp where
        leftBp = searchBBTreeMax leftBpNode
        rightBp = searchBBTreeMax rightBpNode
        midBpVal = value bp
        leftBpVal = value leftBp
        rightBpVal = value rightBp
```

We perform testing on various ILPs with different dimensions. It is apparent that the performance varies potentially among different problems even if they are of comparable sizes. We obtain two hard ILP questions from the MIPLIB 2017 benchmark[7], one naive demonstration problem, and one medium-size problem. As we can see from Figure:2

## IV. PARALLEL IMPLEMENTATION

Unfortunately, the two-phase simplex method cannot be easily parallelized as the tableau is updated sequentially and dependent on the previous iteration. Thus, we want to parallelize the algorithm at each node level. However, as we have seen in class, creating a spark at every single node might not be desirable. We include an additional `parDepth` parameter to control the number of sparks to create while handling these smaller branches sequentially.

```
%
(leftTree, rightTree) = runEval $ do
    if parDepth == 0 then do
```

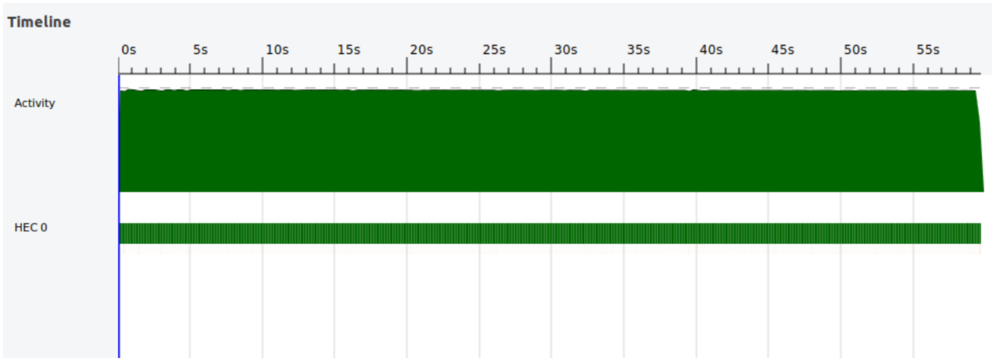



FIG. 2: `gen-ip054.mps` from MIPLIB 2017 benchmark[7] with 30 variables and 24 constraints, `maxDepth` = 20

| # Core | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Runtime (s) | 60.39 | 39.76 | 31.01 | 27,68 | 26.95 | 26.63 |
| Speed-up | 1.00 | 1.52 | 1.95 | 2.18 | 2.24 | 2.27 |

TABLE I: `gen-ip054.mps` from MIPLIB 2017 benchmark[7] with 30 variables and 24 constraints, `maxDepth` = 20 and `parDepth` = 4

```
        leftRawTree <- rparWith rdeepseq $ constructBranchAndCut obj leftTab intMask costVec (maxDepth - 1)
        rightRawTree <- rparWith rdeepseq $  constructBranchAndCut  obj rightTab intMask costVec (maxDepth - 1)
        return (leftRawTree, rightRawTree)
    else do
        leftTree <- rparWith rdeepseq  $  constructParBranchAndCut  obj leftTab intMask costVec (maxDepth - 1) (parDepth - 1)
        rightTree <- rparWith rdeepseq  $  constructParBranchAndCut  obj rightTab intMask costVec (maxDepth - 1) (parDepth - 1)
        return (leftTree, rightTree)
```

The most important part is to write `NFData` for the local data structures and apply `rparWith` strategies to each fully evaluated sub-tree to speed up until we reach the `parDepth` level to finish the program sequentially on those sub-trees. As we can observe in the Figure:3, each processor has relatively balanced workload for the majority of the run, which shows signs of successful speed-ups. Even though threadscope shows promising signs of speed-up, the speed-ups are material with 2 or 3 cores. One potential source of problem was the increasing size of memory needed to store large simplex tableau since the dimension of the tableau grows as the algorithm branches, which leads more work spent on garbage collection. We have also tried tuning the `parDepth` to obtain optimal run time for a given number of cores. For this particular problem, depth of 8 is optimal in Figure:4. Additionally, to tackle the memory allocation problem, we have also tried adding `-A256M` when compiling, which led to less garbage collections but the performance improvements seemed non-material.

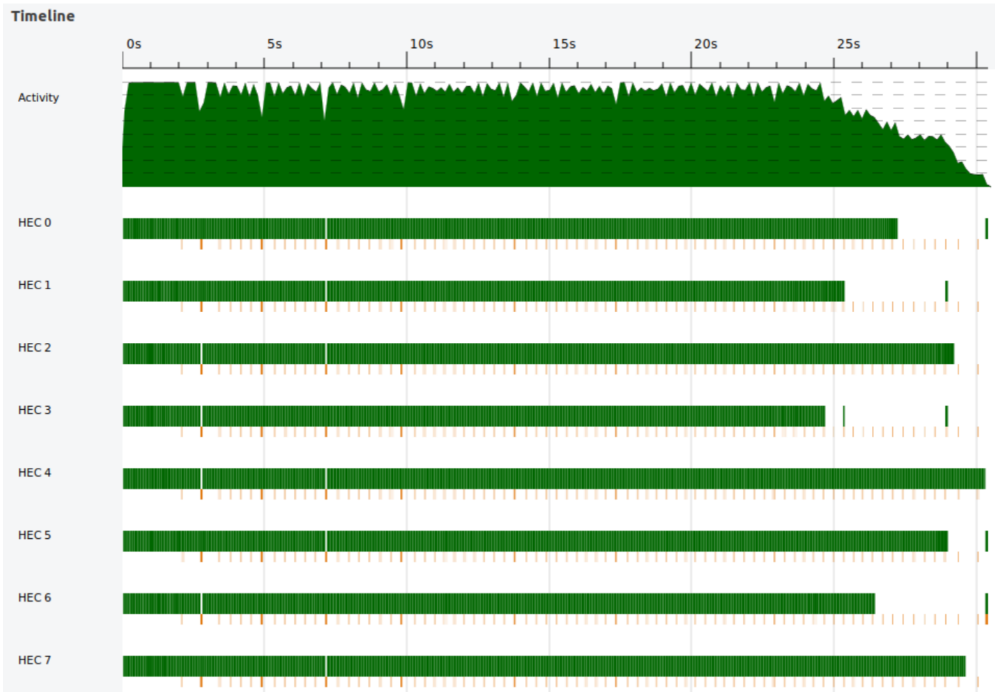



FIG. 3: `gen-ip054.mps` from MIPLIB 2017 benchmark[7] with 30 variables and 24 constraints, `maxDepth` = 20 and `parDepth` = 8

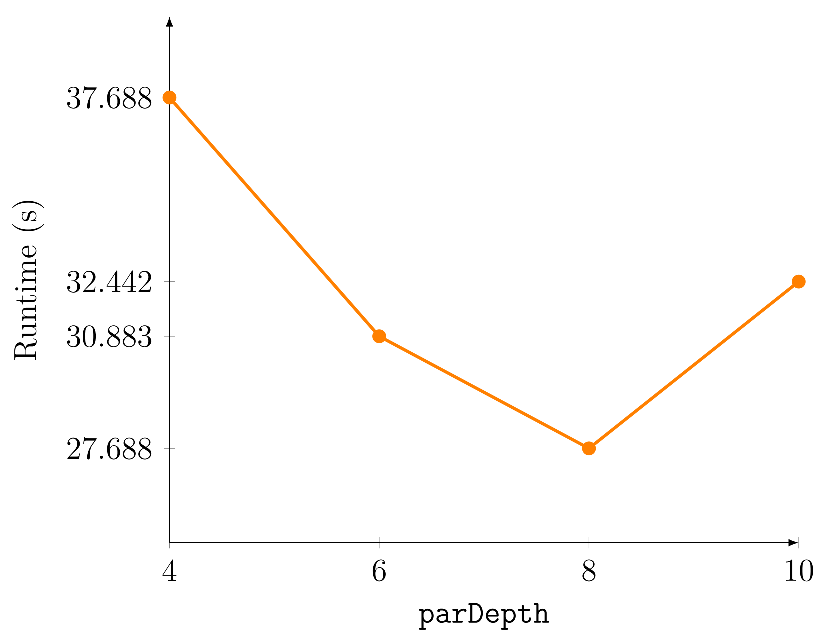



FIG. 4: gen-ip054.mps from MIPLIB 2017 benchmark[7] with 30 variables and 24 constraints, `maxDepth` = 20 and 4 cores

## V. CONCLUSION

By simply applying one layer of parallelism we can speed up a traditionally computationally intensive branch-and-cut algorithm. Moreover, we have built a library to solve linear sub-problems in native Haskell, which can be extended to develop other heuristics on speeding up the integer solving program algorithmically.

## REFERENCES

[1]J. E. Mitchell, "Integer programming: branch and cut algorithmsinteger programming: Branch and cut algorithms," in *Encyclopedia of Optimization*, edited by C. A. Floudas and P. M. Pardalos (Springer US, Boston, MA, 2009) pp. 1643–1650.

[2]Weixi Zhuo's github: https://github.com/WeixiSilhouetteZed/ParBnC.

[3]E. K. Lee and J. E. Mitchell, "Integer programming: branch and bound methodsinteger programming: Branch and bound methods," in *Encyclopedia of Optimization*, edited by C. A. Floudas and P. M. Pardalos (Springer US, Boston, MA, 2009) pp. 1634–1643.

[4]L. Barreto and M. Bauer, "Parallel branch and bound algorithm - a comparison between serial, openmp and mpi implementations," Journal of Physics: Conference Series **256**, 012018 (2010).

[5]J. E. Mitchell, "Integer programming: cutting plane algorithmsinteger programming: Cutting plane algorithms," in *Encyclopedia of Optimization*, edited by C. A. Floudas and P. M. Pardalos (Springer US, Boston, MA, 2009) pp. 1650–1657.

[6]S. Mirzaei, "How to solve an integer programming problem using cutting-plane method," .

[7]A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano, "MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library," Mathematical Programming Computation (2021), 10.1007/s12532-020-00194-3.

## APPENDIX A: USAGE

In fact, our implementation does provide both branch-and-bound and branch-and-cut solvers. There are 4 test cases stored in JSON format. To execute these test cases, please use the following commands.

*a. Sequential Branch-and-Bound with tree output*

```
stack exec ParBnC-exe <json_filename> <maxDepth> seq b tree
```

*b. Sequential Branch-and-Bound with only solution*

```
stack exec ParBnC-exe <json_filename> <maxDepth> seq b solution
```

*c. Sequential Branch-and-Cut with tree output*

```
stack exec ParBnC-exe <json_filename> <maxDepth> seq c tree
```

*d. Sequential Branch-and-Cut with only solution*

```
stack exec ParBnC-exe <json_filename> <maxDepth> seq c solution
```

*e. Parallel Branch-and-Bound with tree output*

```
stack exec ParBnC-exe <json_filename> <maxDepth> par b tree
```

*f. Parallel Branch-and-Bound with only solution*

```
stack exec ParBnC-exe <json_filename> <maxDepth> par b solution
```

*g. Parallel Branch-and-Cut with tree output*

```
stack exec ParBnC-exe <json_filename> <maxDepth> par c tree
```

*h. Parallel Branch-and-Cut with only solution*

```
stack exec ParBnC-exe <json_filename> <maxDepth> par c solution
```

## APPENDIX B: SOURCE CODE

### A. app/Main.hs

```haskell
%
{-# LANGUAGE DeriveGeneric #-}
module Main (main) where

import ParBnC (
    ObjectiveType (Maximization, Minimization),
    BranchProblem,
    toTableau,
    addSlackMatrix,
    constructBranchAndBound,
    constructParBranchAndBound,
    constructBranchAndCut,
    constructParBranchAndCut,
    searchBBTreeMax)
import Data.Aeson
    ( (.:),
      object,
      FromJSON(parseJSON),
      Value(Object),
      KeyValue((.=)),
      ToJSON(toJSON),
      eitherDecode
```

```
    )
import Data.Text (Text)
import qualified Data.ByteString.Lazy as B
import GHC.Generics
import System.Exit (die)
import System.Environment (getArgs, getProgName)
import Numeric.LinearAlgebra as LA
    ( fromLists, fromList, Matrix, R, Vector, size)

data IP = IP {
    name :: !Text,
    costC :: [Double],
    matA :: [[Double]],
    boundB :: [Double]
} deriving (Show, Generic)

instance FromJSON IP
instance ToJSON IP

jsonFile :: FilePath
jsonFile = "test/medium_ip.json"

getJSON :: IO B.ByteString
getJSON = B.readFile jsonFile

obj :: ObjectiveType
obj = Maximization

main :: IO ()
main = do
    args <- getArgs
    case args of
        -- Sequential
        [fileName, maxDepth, "seq", bc, to] -> do
            let jsonFile = fileName
            let getJSON = B.readFile jsonFile
            let md = read maxDepth :: Int
            d <- (eitherDecode <$> getJSON) :: IO (Either String IP)
            case d of
                Left err -> putStrLn $ "Failed to load test case" ++ err
                Right ps -> do
                    let testC = LA.fromList $ costC ps
                    let testMat = LA.fromLists $ matA ps
                    let testB = LA.fromList $ boundB ps
```

```haskell
                let testTab = addSlackMatrix $ toTableau testC testMat testB
                let testIntMask = LA.fromList $ replicate (LA.size testC) True
                case bc of
                    -- Branch and Bound
                    "b" -> do
                        case to of
                            -- full tree output
                            "tree" -> print $ constructBranchAndBound obj testTab testIntMask testC md
                            -- just optimal solution
                            "solution" -> print $ searchBBTreeMax $ constructBranchAndBound obj testTab testIntMask testC md
                            _ -> error "Wrong input format for output type"
                    -- Branch and Cut
                    "c" -> do
                        case to of
                            -- full tree output
                            "tree" -> print $ constructBranchAndCut obj testTab testIntMask testC md
                            -- just optimal solution
                            "solution" -> print $ searchBBTreeMax $ constructBranchAndCut obj testTab testIntMask testC md
                    _ -> error "Wrong input format for Bound/Cut"
    [fileName, maxDepth, "par", bc, to, parDepth] -> do
        let jsonFile = fileName
        let getJSON = B.readFile jsonFile
        let md = read maxDepth :: Int
        let pd = read parDepth :: Int
        d <- (eitherDecode <$> getJSON) :: IO (Either String IP)
        case d of
            Left err -> putStrLn $ "Failed to load test case" ++ err
            Right ps -> do
                let testC = LA.fromList $ costC ps
                let testMat = LA.fromLists $ matA ps
                let testB = LA.fromList $ boundB ps
                let testTab = addSlackMatrix $ toTableau testC testMat testB
                let testIntMask = LA.fromList $ replicate (LA.size testC) True
                case bc of
                    -- Branch and Bound
                    "b" -> do
```

```
                            case to of
                                -- full tree output
                                "tree" -> print $ constructParBranchAndBound obj testTab testIntMask testC md pd
                                -- just optimal solution
                                "solution" -> print $ searchBBTreeMax $ constructParBranchAndBound obj testTab testIntMask testC md pd
                                _ -> error "Wrong input format for output type"
                        -- Branch and Cut
                        "c" -> do
                            case to of
                                -- full tree output
                                "tree" -> print $ constructParBranchAndCut obj testTab testIntMask testC md pd
                                -- just optimal solution
                                "solution" -> print $ searchBBTreeMax $ constructParBranchAndCut obj testTab testIntMask testC md pd
                        _ -> error "Wrong input format for Bound/Cut"
        _ -> do
            pn <- getProgName
            die $ "Usage: stack exec " ++ pn
                ++ "<json_filename> <maxDepth> <seq> <bc> <to> <parDepth>"
```

### B. src/ParBnC.hs

```
%
module ParBnC
    (
        ObjectiveType,
        toTableau,
        addSlackMatrix,
        constructBranchAndBound,
        constructParBranchAndBound,
        constructBranchAndCut,
        constructParBranchAndCut,
        searchBBTreeMax
    ) where

import Numeric.IEEE ( IEEE(infinity) )
import Data.List ( elemIndex )
import Numeric.LinearAlgebra as LA
    ( (<.>),
      dropColumns,
```

```haskell
      fromLists,
      takeColumns,
      cols,
      flatten,
      fromColumns,
      fromRows,
      rows,
      toColumns,
      toRows,
      cmap,
      find,
      maxElement,
      maxIndex,
      scalar,
      sumElements,
      diagl,
      size,
      vector,
      subVector,
      takesV,
      toList,
      vjoin,
      fromList,
      Matrix,
      Konst(konst),
      Indexable((!)),
      R,
      Vector )
import Control.Parallel(par, pseq)
import Control.Parallel.Strategies
    ( rdeepseq, rparWith, runEval, NFData )
import Control.Monad ( when )
import Control.DeepSeq ( NFData(..) )

data MatConstraints = MatVec [[Double]] [Double] deriving Show

data VariableType = INTEGER | CONTINUOUS deriving (Show, Eq)

data ProblemType = LP | MIP deriving Show

data ObjectiveType = Maximization | Minimization deriving (Show, Eq)

epsilonTol :: R
epsilonTol = 1e-10

```

```haskell
failNodeThreshold :: Int
failNodeThreshold = 100

isInt :: (RealFrac a) => a -> Bool
isInt x = x == fromInteger (round x)

toTableau :: LA.Vector R -> LA.Matrix R -> LA.Vector R -> LA.Matrix R
toTableau costC matA constB = tab where
    xb = LA.fromColumns $ LA.toColumns matA ++ [constB]
    z = mappend costC $ vector [0]
    tab = LA.fromRows $ LA.toRows xb ++ [z]

costCheck :: ObjectiveType -> (R -> Bool)
costCheck Maximization = (> 0)
costCheck Minimization = (< 0)

boundCheck :: ObjectiveType -> (R -> Bool)
boundCheck Maximization = (< 0)
boundCheck Minimization = (> 0)

isImprovable :: ObjectiveType -> LA.Matrix R -> Bool
isImprovable obj tab = any (costCheck obj . roundSolution) $ LA.toList cost
    where
    cost = subVector 0 (cols tab - 1) $ tab ! (rows tab - 1)

isImprovableDual :: ObjectiveType -> LA.Matrix R -> Bool
isImprovableDual obj tab = any (boundCheck obj) $ LA.toList bounds where
    lastCol = last $ LA.toColumns tab
    bounds = subVector 0 (rows tab - 1) lastCol

getPivotPosition :: ObjectiveType -> LA.Matrix R -> (Int, Int)
getPivotPosition obj tab = (row, column) where
    z = tab ! (rows tab - 1)
    cost = subVector 0 (cols tab - 1) z
    column = LA.maxIndex cost
    getElem rowEq
        | elem == 0 = infinity::R
        | val < 0 = infinity::R
        | otherwise = val
        where
            elem = rowEq ! column
            lastColVal = rowEq ! (LA.size rowEq - 1)
            val =  lastColVal / (rowEq ! column)
    restrictions = map getElem $ init (LA.toRows tab)

```

```
    Just row =  elemIndex (minimum restrictions) restrictions

getPivotPositionDual :: ObjectiveType -> LA.Matrix R -> (Int, Int)
getPivotPositionDual obj tab = (row, column) where
    lastCol = last $ LA.toColumns tab
    bounds = subVector 0 (rows tab - 1) lastCol
    row = head $ LA.find (boundCheck obj) bounds
    getElem rowEq
        | elem >= 0 = infinity::R
        | otherwise = elem / (rowEq ! (LA.size rowEq - 1))
        where
            elem = rowEq ! row
    restrictions = map getElem $ init (LA.toColumns tab)
    Just column =  elemIndex (minimum restrictions) restrictions

pivotStep :: Matrix R -> (Int, Int) -> Matrix R
pivotStep tab (row, column) = newTableau where
    pivotVal = (tab ! row) ! column
    newPivotRow = cmap (/ pivotVal) $ tab ! row
    updateRow rowIdx rowEq
        | rowIdx == row = newPivotRow
        | otherwise = rowEq - newPivotRow * LA.scalar (rowEq ! column)
    rowSize = LA.rows tab
    newTableau = LA.fromRows $ zipWith updateRow [0..(rowSize - 1)] $ LA.
   toRows tab

isBasic :: Vector R -> Bool
isBasic colVec = (LA.sumElements colVec == 1) && (length zeroVec == colVecLen)
     where
    zeroVec = filter (== 0) $ LA.toList colVec
    colVecLen = LA.size colVec - 1

getSolution :: Matrix R -> Vector R
getSolution tab = solution where
    colSize = LA.cols tab
    columns = LA.toColumns $ LA.takeColumns (colSize - 1) tab
    lastCol = LA.flatten $ LA.dropColumns (colSize - 1) tab
    findSol colVec
        | isBasic colVec = sol
        | otherwise = 0::R where
            oneIndex = head $ LA.find (==1) colVec
            sol = lastCol ! oneIndex
    solution = LA.fromList $ map findSol columns

updateTab :: ObjectiveType -> LA.Matrix R -> Int -> LA.Matrix R
```

```haskell
updateTab obj tab counter
    | not $ isImprovable obj tab = tab
    | counter == 0 = LA.fromLists [[]]
    | otherwise = updateTab obj newTab (counter - 1) where
        pivotPos = getPivotPosition obj tab
        newTab = pivotStep tab pivotPos


updateTabDebug :: ObjectiveType -> LA.Matrix R -> Int -> LA.Matrix R
updateTabDebug obj tab counter
    | counter == 0 = tab
    | not $ isImprovable obj tab = tab
    | otherwise = updateTabDebug obj newTab (counter - 1) where
        pivotPos = getPivotPosition obj tab
        newTab = pivotStep tab pivotPos

updateTabDual :: ObjectiveType -> LA.Matrix R -> LA.Matrix R
updateTabDual obj tab
    | not $ isImprovableDual obj tab = tab
    | otherwise = updateTabDual obj newTab where
        pivotPos = getPivotPositionDual obj tab
        newTab = pivotStep tab pivotPos

isImprovableMixed :: LA.Matrix R -> Bool
isImprovableMixed tab = any ((> 0) . roundSolution) (LA.toList costs) where
    lastRow = last $ LA.toRows tab
    costs = subVector 0 (LA.cols tab - 1) lastRow

getPhaseOneTab :: LA.Matrix R -> (LA.Matrix R, LA.Vector R, Bool)
getPhaseOneTab tab = (newTab, oldCost, needed) where
    (rowSize, colSize) = LA.size tab
    (constRows, [oldCost]) = splitAt (rowSize - 1) $ LA.toRows tab
    updateRow :: LA.Vector R -> LA.Vector R
    updateRow rowVec
        | boundVal < 0 = -rowVec
        | otherwise = rowVec where
            boundVal = last $ LA.toList rowVec
    zeroRow = LA.konst (0::R) colSize
    mixedRows = filter (\rowVec -> (rowVec ! (LA.size rowVec - 1)) < 0)
   constRows
    needed = not $ null mixedRows
    sumRow = sum $ map (\x -> -x) $ mixedRows ++ [zeroRow]
    newConstRows = map updateRow constRows
    newTab = LA.fromRows $ newConstRows ++ [sumRow]

```

```haskell
getPivotPositionMixed :: LA.Matrix R -> (Int, Int)
getPivotPositionMixed tab = (row, column) where
    z = tab ! (rows tab - 1)
    colSize = cols tab - 1
    cost = subVector 0 colSize z
    maxCost = LA.maxElement cost
    column = head $ LA.find (== maxCost) cost
    getElem rowEq
        | elem == 0 = infinity::R
        | val < 0 = infinity::R
        | otherwise = val
        where
            elem = rowEq ! column
            lastColVal = rowEq ! (LA.size rowEq - 1)
            val =  lastColVal / (rowEq ! column)
    restrictions = map getElem $ init (LA.toRows tab)
    minRatio = minimum restrictions
    row =  last [idx | (idx, ratio) <- zip [0..colSize] restrictions, ratio ==
    minRatio]

updateMixedTab :: LA.Matrix R -> Int -> LA.Matrix R
updateMixedTab tab counter
    | not $ isImprovableMixed tab = tab
    | counter == 0 = LA.fromLists [[]]
    | otherwise = updateMixedTab newTab (counter - 1) where
        pivotPos = getPivotPositionMixed tab
        newTab = pivotStep tab pivotPos

simplexWithTab :: ObjectiveType -> LA.Matrix R -> (R, LA.Vector R, LA.Matrix R
   )
simplexWithTab obj tab = (optVal, solution, lastTab) where
    lastTab = updateTab obj tab failNodeThreshold
    solution = getSolution lastTab
    (rowSize, colSize) = LA.size lastTab
    lastVal = lastTab ! (rowSize - 1) ! (colSize - 1)
    optVal = if obj == Maximization then lastVal else (-lastVal)

simplexWithMixedTab :: ObjectiveType -> LA.Matrix R -> (R, LA.Vector R, LA.
   Matrix R)
simplexWithMixedTab obj tab
    | failedInter = (-infinity, getSolution phaseOneTab, phaseOneTab)
    | infeasible = (-infinity, getSolution interTab, interTab)
    | failedNode = (-infinity, getSolution interTab, interTab)
    | otherwise  = (lastVal, solution, lastTab) where
        (phaseOneTab, oldCost, needed) = getPhaseOneTab tab
```

```
        interTab
            | needed = updateMixedTab phaseOneTab failNodeThreshold
            | otherwise = phaseOneTab
        failedInter = LA.fromLists [[]] == interTab
        (rowSize, colSize) = LA.size interTab
        lastPhaseOneVal = interTab ! (rowSize - 1) ! (colSize - 1)
        infeasible = not $ isClose lastPhaseOneVal (0::R)

        (constRows, _) = splitAt (rowSize - 1) $ LA.toRows interTab

        phaseTwoTab = LA.fromRows $ constRows ++ [oldCost]
        lastTab = updateTab obj phaseTwoTab failNodeThreshold
        failedNode = lastTab == LA.fromLists [[]]
        solution = getSolution lastTab
        lastVal = lastTab ! (rowSize - 1) ! (colSize - 1)
        optVal = lastVal

getGomoryCut :: Vector R -> Vector R
getGomoryCut rowVec = gomoryCons where
    [varPart, constPart] = takesV [LA.size rowVec - 1, 1] rowVec
    posDec num = -posFrac where
        (intPart, fracPart) = properFraction num
        posFrac = if fracPart >= 0.0 then fracPart else 1.0 + fracPart
    gomoryCons = vjoin [cmap posDec varPart, vector [1::R], cmap posDec
   constPart]

addSlackColumn :: Matrix R -> Matrix R
addSlackColumn tab = newTab where
    columns = LA.toColumns tab
    (rowSize, colSize) = LA.size tab
    (varColumns, lastColumn) = splitAt (colSize - 1) columns
    zeroVec = LA.konst 0 rowSize :: Vector R
    extendedTab = varColumns ++ [zeroVec] ++ lastColumn
    newTab = LA.fromColumns extendedTab

addNewRow :: Matrix R -> Vector R -> Matrix R
addNewRow tab newRow = newTab where
    rows = LA.toRows tab
    (rowSize, colSize) = LA.size tab
    (constRows, lastRow) = splitAt (rowSize - 1) rows
    newTab = LA.fromRows $ constRows ++ [newRow] ++ lastRow

addGomoryCut :: Matrix R -> Vector R -> Matrix R
addGomoryCut tab gomoryRow = newTab where
    (rowSize, colSize) = LA.size tab
```

```
    interTab = addSlackColumn tab
    rows = LA.toRows interTab
    (consRows, costRow) = splitAt (rowSize - 1) rows
    newTab = LA.fromRows (consRows ++ [gomoryRow] ++ costRow)

performGomoryCut :: ObjectiveType -> LA.Matrix R -> Int -> (R, LA.Vector R, LA
    .Matrix R)
performGomoryCut obj tab varIdx = (newVal, newSol, newTab) where
    interTab = addGomoryCut tab $ getGomoryCut $ tab ! varIdx
    newTab = updateTabDual obj interTab
    newSol = getSolution newTab
    (rowSize, colSize) = LA.size newTab
    lastVal = newTab ! (rowSize - 1) ! (colSize - 1)
    newVal = if obj == Maximization then lastVal else (-lastVal)

addSlackMatrix :: LA.Matrix R -> LA.Matrix R
addSlackMatrix tab = newTab where
    (rowSize, colSize) = LA.size tab
    slackIdentity = LA.diagl $ replicate (rowSize - 1) (1::R)
    zeroVec = LA.konst (0::R)  (rowSize - 1)
    newSlackMat = LA.fromRows $ LA.toRows slackIdentity ++ [zeroVec]
    (prevCols, lastCol) = splitAt (colSize - 1) $ LA.toColumns tab
    newTab = LA.fromColumns (prevCols ++ LA.toColumns newSlackMat ++ lastCol)

isClose :: R -> R -> Bool
isClose x y = diff < epsilonTol where
    diff = abs $ x - y

roundSolution :: R -> R
roundSolution num
    | diff < epsilonTol = roundCand
    | otherwise = num where
        roundCand = fromIntegral $ round num
        diff = abs $ roundCand - num

integerSolved :: [Bool] -> [Bool] -> [Bool]
integerSolved = zipWith isIntSol where
    isIntSol False _ = True
    isIntSol True mask = mask

fromJust :: Maybe a -> a
fromJust (Just a) = a
fromJust Nothing = error "non-existing index"

findNonIntIndex :: [Bool] -> [Bool] -> Int
```

```
findNonIntIndex intMask solMask = solIdx where
    f False solBool = True
    f True True = True
    f True False = False
    maybeSolIdx = elemIndex False $ zipWith f intMask solMask
    solIdx = fromJust maybeSolIdx


getBranches :: Matrix R -> Vector R -> Int -> (Matrix R, Matrix R)
getBranches tab solVec branchIdx = (leftTab, rightTab) where
    currSolVal = solVec ! branchIdx
    colSize = LA.cols tab
    baseVec = LA.vjoin [LA.konst 0 branchIdx :: Vector R, vector [1::R], LA.konst (0::R) (colSize - branchIdx - 2)]

    leftBound = fromIntegral $ floor currSolVal
    leftRow = LA.vjoin [baseVec, vector [1::R, leftBound]]
    rightBound = fromIntegral $ ceiling currSolVal
    rightRow = LA.vjoin [-baseVec, vector [1::R, -rightBound]]

    slackTab = addSlackColumn tab
    leftTab = addNewRow slackTab leftRow
    rightTab = addNewRow slackTab rightRow

data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show)
instance NFData a => NFData (Tree a) where
    rnf Nil = ()
    rnf (Node l a r) = rnf l `seq` rnf a `seq` rnf r

data BranchProblem = BranchProblem {
    solution:: Vector R, value :: R
} deriving (Show)
instance NFData BranchProblem where
    rnf BranchProblem {solution = s, value = v} = rnf s `seq` rnf v

constructBranchAndBound :: ObjectiveType -> Matrix R -> Vector Bool -> Vector R -> Int -> Tree BranchProblem
constructBranchAndBound obj tab intMask costVec maxDepth
    | infeasible = Nil
    | maxDepth == 0 = Nil
    | and $ integerSolved intList solMask = currProb Nil Nil
    | otherwise = currProb leftTree rightTree where
        (y, currSol, newTab) = simplexWithMixedTab obj tab
        infeasible = y == (-infinity)
        candVal = costVec <.> LA.subVector 0 (LA.size costVec) currSol
```

```
        currProb = Node BranchProblem {
           solution = currSol, value = candVal
        }
        intList = LA.toList intMask
        solMask =  map (isInt . roundSolution) $ LA.toList currSol
        nextIdx = findNonIntIndex intList solMask
        (leftTab, rightTab) = getBranches tab currSol nextIdx
        leftTree = constructBranchAndBound obj leftTab intMask costVec (maxDepth - 1)
        rightTree = constructBranchAndBound obj rightTab intMask costVec (maxDepth - 1)

constructParBranchAndBound :: ObjectiveType -> Matrix R -> Vector Bool -> Vector R -> Int -> Int -> Tree BranchProblem
constructParBranchAndBound obj tab intMask costVec maxDepth parDepth
    | infeasible = Nil
    | maxDepth == 0 = Nil
    | and $ integerSolved intList solMask = currProb Nil Nil
    | otherwise = currProb leftTree rightTree where
        (y, currSol, newTab) = simplexWithMixedTab obj tab
        infeasible = y == (-infinity)
        candVal = costVec <.> LA.subVector 0 (LA.size costVec) currSol
        currProb = Node BranchProblem {
            solution = currSol, value = candVal
        }
        intList = LA.toList intMask
        solMask =  map (isInt . roundSolution) $ LA.toList currSol
        nextIdx = findNonIntIndex intList solMask
        (leftTab, rightTab) = getBranches tab currSol nextIdx
        (leftTree, rightTree)
            | parDepth == 0 = (leftRawTree, rightRawTree)
            | otherwise = leftTree `par` rightTree `pseq` (leftTree, rightTree) where
                leftRawTree = constructBranchAndBound obj leftTab intMask costVec (maxDepth - 1)
                rightRawTree = constructBranchAndBound obj rightTab intMask costVec (maxDepth - 1)
                leftTree = constructParBranchAndBound obj leftTab intMask costVec (maxDepth - 1) (parDepth - 1)
                rightTree = constructParBranchAndBound obj rightTab intMask costVec (maxDepth - 1) (parDepth - 1)

constructBranchAndCut :: ObjectiveType -> Matrix R -> Vector Bool -> Vector R -> Int -> Tree BranchProblem
constructBranchAndCut obj tab intMask costVec maxDepth
```

```haskell
    | maxDepth == 0 = Nil
    | infeasible = Nil
    | and $ integerSolved intList solMask = currProb Nil Nil
    | otherwise = currProb leftTree rightTree where
        (y, currSol, newTab) = simplexWithMixedTab obj tab
        infeasible = y == (-infinity)
        candVal = costVec <.> LA.subVector 0 (LA.size costVec) currSol
        currProb = Node BranchProblem {
           solution = currSol, value = candVal
        }
        intList = LA.toList intMask
        solMask =  map (isInt . roundSolution) $ LA.toList currSol
        nextIdx = findNonIntIndex intList solMask
        cutTab = addGomoryCut tab $ getGomoryCut $ newTab ! nextIdx
        (leftTab, rightTab) = getBranches cutTab currSol nextIdx
        leftTree = constructBranchAndCut obj leftTab intMask costVec (maxDepth - 1)
        rightTree = constructBranchAndCut obj rightTab intMask costVec (maxDepth - 1)

constructParBranchAndCut :: ObjectiveType -> Matrix R -> Vector Bool -> Vector R -> Int -> Int -> Tree BranchProblem
constructParBranchAndCut obj tab intMask costVec maxDepth parDepth
    | infeasible = Nil
    | maxDepth == 0 = Nil
    | and $ integerSolved intList solMask = currProb Nil Nil
    | otherwise = currProb leftTree rightTree where
        (y, currSol, newTab) = simplexWithMixedTab obj tab
        infeasible = y == (-infinity)
        candVal = costVec <.> LA.subVector 0 (LA.size costVec) currSol
        currProb = Node BranchProblem {
            solution = currSol, value = candVal
        }
        intList = LA.toList intMask
        solMask =  map (isInt . roundSolution) $ LA.toList currSol
        nextIdx = findNonIntIndex intList solMask
        cutTab = addGomoryCut tab $ getGomoryCut $ newTab ! nextIdx
        (leftTab, rightTab) = getBranches cutTab currSol nextIdx
        (leftTree, rightTree) = runEval $ do
            if parDepth == 0 then do
                leftRawTree <- rparWith rdeepseq $ constructBranchAndCut obj leftTab intMask costVec (maxDepth - 1)
                rightRawTree <- rparWith rdeepseq $  constructBranchAndCut obj rightTab intMask costVec (maxDepth - 1)
                return (leftRawTree, rightRawTree)
```

```
            else do
                leftTree <- rparWith rdeepseq  $  constructParBranchAndCut obj leftTab intMask costVec (maxDepth - 1) (parDepth - 1)
                rightTree <- rparWith rdeepseq  $  constructParBranchAndCut obj rightTab intMask costVec (maxDepth - 1) (parDepth - 1)
                return (leftTree, rightTree)

searchBBTreeMax :: Tree BranchProblem -> BranchProblem
searchBBTreeMax Nil = BranchProblem{solution = LA.fromList [], value = -infinity}
searchBBTreeMax (Node bp Nil Nil) = bp
searchBBTreeMax (Node bp leftBpNode Nil) = searchBBTreeMax leftBpNode
searchBBTreeMax (Node bp Nil rightBpNode) = searchBBTreeMax rightBpNode
searchBBTreeMax (Node bp leftBpNode rightBpNode)
    | leftBpVal > rightBpVal = leftBp
    | otherwise = rightBp where
        leftBp = searchBBTreeMax leftBpNode
        rightBp = searchBBTreeMax rightBpNode
        midBpVal = value bp
        leftBpVal = value leftBp
        rightBpVal = value rightBp
```