Одеський національний університет імені І. І. Мечникова
Інститут математики, економіки і механіки
Кафедра оптимального керування та економічної кібернетики

Дипломна робота

бакалавра

на тему: **«Задача упаковки в R2»**

«Two-dimensional packing problem»

Виконав: студент денної форми навчання
напряму підготовки
6.040301 Прикладна математика
Добровольський Георгій Костянтинович

Керівник: к. ф.-м. н., доц. Кічмаренко О.Д.
Рецензент: док. т. н., проф. Положаєнко С. А.

Рекомендовано до захисту:
Протокол засідання кафедри
№ ___ від «_____» _____ р.
Завідувач кафедри
_____ _____

Захищено на засіданні ЕК № _____
Протокол № ____ від «____» ____ р.
Оцінка _____ / _____ / _____
Голова ЕК
_____ _____

Одеса — 2017 р.

ЗМІСТ

ВСТУП

Задачу упаковки застосовують в різноманітних сферах виробництва: починаючи з виготовлення листового скла, проектування деталей та викройок для виготовлення меблів, закінчуючи комп'ютерною графікою та системами розподілення ресурсів в великих комп'ютерних кластерах. Метою задачі упаковки є розміщення елементів в контейнері з максимальною щільністю або використання мінімальної кількості таких контейнерів. Людина здатна виконувати цю задачу, але це потребує багато часу і не завжди результат є оптимальним.

Масштаби можливої економії за допомогою використання алгоритмів для автоматизації вирішення ростуть кожного року: тільки в Китаї обсяг випуску текстильної промисловості сягає більше 40 мільярдів квадратних метрів тканин, тому навіть порівняно невелика економія за рахунок мінімізації надлишків виробництва може призвести до значного скорочення витрат матеріалу.

У більшості випадків така задача є NP-повною, і для задач з великою кількістю об'єктів є неможливим використання традиційних алгоритмів через обмеженість обчислювальних ресурсів. В таких випадках, альтернативою може бути підхід, який ґрунтується на використанні еволюційних алгоритмів, зокрема генетичного.

Вперше еволюційний алгоритм був представлений Нільсом Барічеллі [1] в 1954-му році в Інституті перспективних досліджень в Прінстоні, а вирішити інженерні проблеми за допомогою стратегії еволюції вдалося вперше групі Інго Рехенберга [2].

Метою дипломної роботи є дослідження задачі упаковки, можливість та ефективність вирішення її підкласу – задачі розкрою – за допомогою генетичного алгоритму.

ВИСНОВКИ

У роботі розглянуто генетичний алгоритм як один з видів еволюційних алгоритмів. Розроблено програму на мові програмування Go, яка моделює задачу гільйотинного розкрою та реалізує генетичний алгоритм для її вирішення. Для порівняння ефективності вирішення задачі оптимізації з з різними способами кросоверу було проведено обчислювальний експеримент на різних наборах вхідних даних.

Продемонстровано, що еволюційні алгоритми, зокрема генетичний, можна використовувати для задач двовимірного розкрою. Генетичний алгоритм показав задовільні результати в обчислювальному експерименті. Розроблено та представлено візуалізацію отриманих рішень.

За допомогою генетичного алгоритму можливо вирішувати більш широкий клас задач оптимізації розміщення, зокрема з фігурами довільної форми. Серед недоліків алгоритму потрібно відзначити недетермінованість та неможливість асинхронної чи паралельної реалізації алгоритму.

СПИСОК ЛІТЕРАТУРИ

1. Barricelli, Nils Aall (1954). «Esempi numerici di processi di evoluzione». Methodos: 45–68.
2. Rechenberg Ingo. Evolutionsstrategie. — Holzmann-Froboog, 1973. — ISBN 3-7728-0373-3.
3. Lodi, Martello, Monaci, 2002, c. 241–252.
4. Gerhard Wäscher, Heike Haußner, Holger Schumann, An improved typology of cutting and packing problems, European Journal of Operational Research 183 (2007) 1109—1130
5. Канторович Л. В., Залгаллер В. А. Рациональный раскрой промышленных материалов. — Новосибирск: Наука, 1971.
6. P. C. Gilmore, R. E. Gomory A linear programming approach to the cutting-stock problem // Operations Research. — 1961. — № 9. — С. 849—859.
7. P. C. Gilmore, R. E. Gomory A linear programming approach to the cutting-stock problem - Part II // Operations Research. — 1963. — № 11. — С. 863—888.
8. V. de Carvalho Exact solution of cutting stock problems using column generation and branch-and-bound // International Transactions in Operational Research. — 1998. — № 5. — С. 35—44.
9. Barricelli, Nils Aall (1957). «Symbiogenetic evolution processes realized by artificial methods». Methodos: 143–182.

Додаток А

Код програми

```
package guillotine

import (
        "fmt"
        "math/rand"
        "sort"
        "time"
)

var _ = fmt.Println

func GetPhenotype(spec *CutSpec, genotype Genotype) *
   LayoutTree {
        genotype = genotype.copy()
        sort.Sort(genotype)
        lt := NewLayoutTree(spec)
        remaining := len(spec.Boards) - 1
        for i := 0; remaining > 0; i++ {
                wj := &genotype[i]
                if lt.take(wj.i, wj.j, wj.config) {
                        remaining -= 1
                }
        }
        return lt
}

type Population []Genotype

func NewRandomPopulation(nboards uint16, size uint, r *
   rand.Rand) Population {
        pop := make([]Genotype, size)
```

```
        for i := range pop {
                pop[i] = NewRandomGenotype(nboards, r)
        }
        return pop
}

type RankedPopulation struct {
        Pop        Population
        Fitnesses  []uint
}

func (rp *RankedPopulation) Less(i, j int) bool {
        return rp.Fitnesses[i] < rp.Fitnesses[j]
}
func (rp *RankedPopulation) Swap(i, j int) {
        rp.Pop[i], rp.Pop[j] = rp.Pop[j], rp.Pop[i]
        rp.Fitnesses[i], rp.Fitnesses[j] = rp.Fitnesses[j
          ], rp.Fitnesses[i]
}
func (rp *RankedPopulation) Len() int { return len(rp.Pop
  ) }

type Selector interface {
        next() Genotype
}
type SelectorBuilder func(rp *RankedPopulation) Selector

type TournamentSelector struct {
        size int
        buf  FitnessPositions
        p    float32
        rp   *RankedPopulation
        r    *rand.Rand
        min  bool
```

```
}

func NewTournamentSelectorBuilder(size int, p float32, r
  *rand.Rand, min bool) SelectorBuilder {
        return func(rp *RankedPopulation) Selector {
                return &TournamentSelector{
                        size: size,
                        buf:  make([]FitnessPosition,
                           size),
                        p:    p,
                        r:    r,
                        rp:   rp,
                        min:  min,
                }
        }
}

type FitnessPosition struct {
        i       int
        fitness uint
}

type FitnessPositions []FitnessPosition

func (fps FitnessPositions) Len() int           { return
  len(fps) }
func (fps FitnessPositions) Less(i, j int) bool { return
  fps[i].fitness < fps[j].fitness }
func (fps FitnessPositions) Swap(i, j int)      { fps[i],
   fps[j] = fps[j], fps[i] }

func (ts *TournamentSelector) winnerRank() int {
        //This could be faster if modelled with a
        //negative binomial distribution generator
```

```
        for {
                for i := 0; i < ts.size; i++ {
                        if ts.r.Float32() > ts.p {
                                return i
                        }
                }
        }
}

func (ts *TournamentSelector) next() Genotype {
        fps := ts.buf
        for i := 0; i < ts.size; i++ {
                ri := ts.r.Intn(len(ts.rp.Fitnesses))
                fps[i].i = ri
                fps[i].fitness = ts.rp.Fitnesses[ri]
        }
        var winnerIndex int
        winnerRank := ts.winnerRank()
        if ts.min {
                winnerIndex = fps.getKminIndex(winnerRank
                   )
        } else {
                winnerIndex = fps.getKmaxIndex(winnerRank
                   )
        }
        return ts.rp.Pop[winnerIndex]
}

func (fps FitnessPositions) getKminIndex(k int) int {
        sort.Sort(fps)
        return fps[k].i
}
func (fps FitnessPositions) getKmaxIndex(k int) int {
        sort.Reverse(fps)
```

```
        return fps[k].i
}

func (pop Population) checkEvenSize() {
        if len(pop)%2 != 0 {
                panic("Population size must be even")
        }
}

type GeneticAlgorithm struct {
        Spec             *CutSpec
        Evaluator        Fitness
        Mutator          Mutator
        Breeder          Crossover
        SelectorBuilder  SelectorBuilder
        R                *rand.Rand
        EliteSize        uint
        PopulationSize   uint
        Generations      uint
}

func (ga GeneticAlgorithm) breed(p1, p2 Genotype) (c1, c2
    Genotype) {
        c1, c2 = ga.Breeder(p1, p2, ga.R)
        ga.Mutator(c1, ga.R)
        ga.Mutator(c2, ga.R)
        return c1, c2
}

func (ga *GeneticAlgorithm) Evaluate(pop Population) (rp
  *RankedPopulation) {
        fitness := make([]uint, len(pop))
        for i, genotype := range pop {
                phenotype := GetPhenotype(ga.Spec,
```

```
                genotype)
            fitness[i] = ga.Evaluator(phenotype)
        }
        rp = &RankedPopulation{pop, fitness}
        sort.Sort(rp)
        return rp
}

func (ga *GeneticAlgorithm) Next(rp *RankedPopulation)
   Population {
        selector := ga.SelectorBuilder(rp)
        psize := uint(len(rp.Pop))
        pepsi := make([]Genotype, psize)
        copy(pepsi[:ga.EliteSize], rp.Pop[:ga.EliteSize])
        for i := ga.EliteSize; i < psize; i++ {
            p1, p2 := selector.next(), selector.next
               ()
            c1, c2 := ga.breed(p1, p2)
            pepsi[i] = c1
            if i < psize-1 {
                i++
                pepsi[i] = c2
            }
        }
        return pepsi
}

func (ga *GeneticAlgorithm) Run() *LayoutTree {
        pop := NewRandomPopulation(uint16(len(ga.Spec.
           Boards)), ga.PopulationSize, ga.R)
        rankedPop := ga.Evaluate(pop)
        for i := uint(1); i < ga.Generations; i++ {
            pop = ga.Next(rankedPop)
            rankedPop = ga.Evaluate(pop)
```

```
        }
        return GetPhenotype(ga.Spec, rankedPop.Pop[0])
}

func (ga *GeneticAlgorithm) TimeBoundedRun(limit time.
    Duration) (gn uint, lt *LayoutTree) {
        start := time.Now()
        pop := NewRandomPopulation(uint16(len(ga.Spec.
            Boards)), ga.PopulationSize, ga.R)
        rankedPop := ga.Evaluate(pop)
        for i := uint(1); i < ga.Generations; i++ {
                ng := int64(i)
                if (time.Since(start).Nanoseconds()*(ng
                    +1))/ng > limit.Nanoseconds() {
                        return i, GetPhenotype(ga.Spec,
                            rankedPop.Pop[0])
                } else {
                        pop = ga.Next(rankedPop)
                        rankedPop = ga.Evaluate(pop)
                }
        }
        return ga.Generations, GetPhenotype(ga.Spec,
            rankedPop.Pop[0])
}
```

```
package guillotine

import "math/rand"

type WeightedJoin struct {
        weight float32
        i, j   uint16
        config Join
}
```

```
type Genotype []WeightedJoin

func (g Genotype) copy() Genotype {
        c := make([]WeightedJoin, len(g))
        copy(c, g)
        return c
}

func NewGenotype(n uint16) Genotype {
        length := n * (n - 1) / 2
        return make([]WeightedJoin, length)
}

func NewRandomGenotype(nboards uint16, r *rand.Rand)
   Genotype {
        c := NewGenotype(nboards)
        k := 0
        for i := uint16(0); i < nboards; i++ {
                for j := i + 1; j < nboards; j++ {
                        config := r.Intn(8)
                        wj := &c[k]
                        wj.i = i
                        wj.j = j
                        wj.config = Join(config)
                        wj.weight = r.Float32()
                        k++
                }
        }
        return c
}

func (c Genotype) Len() int           { return len(c) }
func (c Genotype) Swap(i, j int)      { c[i], c[j] = c[j
   ], c[i] }
```

```
func (c Genotype) Less(i, j int) bool { return c[i].
    weight < c[j].weight }

//Create a fresh pair of Genotypes, utility function for
    Crossovers
func freshPair(p1, p2 Genotype) (n int, c1, c2 Genotype)
    {
        if n := len(p1); n != len(p2) {
                panic("weighted joins must have the same
                    length")
        } else {
                return n, make([]WeightedJoin, n), make
                    ([]WeightedJoin, n)
        }
}

type Crossover func(p1, p2 Genotype, r *rand.Rand) (c1,
    c2 Genotype)

func UniformCrossover(p1, p2 Genotype, r *rand.Rand) (c1,
    c2 Genotype) {
        n, c1, c2 := freshPair(p1, p2)
        for i := 0; i < n; {
                rs := r.Int63()
                for j := uint(0); i < n && j < 63; j++ {
                        if (rs & (1 << j)) == 0 {
                                c1[i] = p1[i]
                                c2[i] = p2[i]
                        } else {
                                c1[i] = p2[i]
                                c2[i] = p1[i]
                        }
                        i++
                }
```

```
        }
        return
}

var _ Crossover = UniformCrossover

func OnePointCrossover(p1, p2 Genotype, r *rand.Rand) (c1
    , c2 Genotype) {
        n, c1, c2 := freshPair(p1, p2)
        cpoint := rand.Intn(n)
        copy(c1[:cpoint], p1[:cpoint])
        copy(c1[cpoint:], p2[cpoint:])

        copy(c2[:cpoint], p2[:cpoint])
        copy(c2[cpoint:], p1[cpoint:])
        return
}

var _ Crossover = OnePointCrossover

func TwoPointCrossover(p1, p2 Genotype, r *rand.Rand) (c1
    , c2 Genotype) {
        n, c1, c2 := freshPair(p1, p2)
        point1 := rand.Intn(n)
        point2 := rand.Intn(n)
        if point1 > point2 {
                point1, point2 = point2, point1
        }

        copy(c1[:point1], p1[:point1])
        copy(c1[point1:point2], p2[point1:point2])
        copy(c1[point2:], p1[point2:])

        copy(c2[:point1], p2[:point1])
```

```
        copy(c2[point1:point2], p1[point1:point2])
        copy(c2[point2:], p2[point2:])
        return
}

var _ Crossover = TwoPointCrossover

type Mutator func(Genotype, *rand.Rand)

//In-place chromosome mutation, by replacing some of the
//gene weights by a new random weight.
//Given a chromosome, RandomNorm(p, sigma)
//genes will mutate
//Mutations are done with replacement, meaning a weight
//can be mutated multiple times. The effect is that
//for a mu value close to the chromosome length,
//the actual number of mutated genes will be less.
//Hopefully that's not an intended usecase.
type NormalWeightMutator struct {
        Mean, StdDev float64
}

func (p NormalWeightMutator) Mutate(c Genotype, r *rand.
   Rand) {
        take := uint16(r.NormFloat64()*p.StdDev + p.Mean)
        for ; take > 0; take-- {
                i := rand.Intn(len(c))
                c[i].weight = rand.Float32()
        }
}

type NormalConfigMutator struct {
        Mean, StdDev float64
}
```

```
func (p NormalConfigMutator) Mutate(c Genotype, r *rand.
    Rand) {
        take := uint16(r.NormFloat64()*p.StdDev + p.Mean)
        for ; take > 0; take-- {
                i := rand.Intn(len(c))
                c[i].config = Join(rand.Intn(8))
        }
}

type CompoundWeightConfigMutator struct {
        Weight NormalWeightMutator
        Config NormalConfigMutator
}

func (c CompoundWeightConfigMutator) Mutate(g Genotype, r
    *rand.Rand) {
        c.Weight.Mutate(g, r)
        c.Config.Mutate(g, r)
}

var _ Mutator = NormalWeightMutator{}.Mutate
var _ Mutator = NormalConfigMutator{}.Mutate
var _ Mutator = CompoundWeightConfigMutator{}.Mutate
```

```
package guillotine

func max(a, b uint) uint {
        if a >= b {
                return a
        } else {
                return b
        }
}
```

```
type Board struct {
        Width, Height uint
}

func (b Board) rotated() Board {
        return Board{b.Height, b.Width}
}

func (left Board) Hstack(right Board) Board {
        return Board{left.Width + right.Width, max(left.
           Height, right.Height)}
}

func (top Board) Vstack(bottom Board) Board {
        return Board{max(top.Width, bottom.Width), top.
           Height + bottom.Height}
}

func (b Board) Hsplit(y uint) (b1, b2 Board) {
        if y > b.Height {
                panic("invalid split position")
        }
        return Board{b.Width, y}, Board{b.Width, b.Height
           - y}
}

func (b Board) Vsplit(x uint) (b1, b2 Board) {
        if x > b.Width {
                panic("invalid split position")
        }
        return Board{x, b.Height}, Board{b.Width - x, b.
           Height}
}
```

```
func (board Board) Area() uint {
        return board.Width * board.Height
}

type CutSpec struct {
        Boards     []Board
        MaxWidth   uint
        TotalArea  uint
}

func (spec *CutSpec) Fits(width, height uint) bool {
        return width > 0 && height > 0 && (spec.MaxWidth
          == 0 ||
                width <= spec.MaxWidth || height <= spec.
                  MaxWidth)
}

func (spec *CutSpec) Add(width, height uint) *CutSpec {
        spec.Boards = append(spec.Boards, Board{width,
          height})
        spec.TotalArea += width * height
        return spec
}

func newCutSpec(nboards uint, maxWidth uint) *CutSpec {
        return &CutSpec{Boards: make([]Board, 0, nboards)
          , MaxWidth: maxWidth}
}
```

```
package guillotine

type Direction bool

const (
        HORIZONTAL = false
```

```
        VERTICAL    = true
)

type Join uint8

const JOIN Join = 0

func (j Join) direct(d Direction) Join {
        if d == VERTICAL {
                return j | DIRECTION_MASK
        } else {
                return j &^ DIRECTION_MASK
        }
}
func (j Join) irotated() Join {
        return j | IROT_MASK
}
func (j Join) istraight() Join {
        return j &^ IROT_MASK
}

func (j Join) jrotated() Join {
        return j | JROT_MASK
}

func (j Join) jstraight() Join {
        return j &^ JROT_MASK
}

const (
        DIRECTION_MASK = 1 << iota
        IROT_MASK
        JROT_MASK
)
```

```
func (c Join) direction() Direction {
        return (c & DIRECTION_MASK) != 0
}
func (c Join) irot() bool {
        return (c & IROT_MASK) != 0
}
func (c Join) jrot() bool {
        return (c & JROT_MASK) != 0
}

type PickLeaf struct {
        //1-based node index
        //0 means no parent
        Parent uint16
        Rot    bool
}

type StackNode struct {
        // 0-based mixed index.
        //0 to n-1 => leaf index
        //n to 2n-2 => node index
        Left, Right uint16
        //1-based node index
        //0 means no parent
        Parent    uint16
        Direction Direction `json:"Vertical"`
}

type LayoutTree struct {
        Picks    []PickLeaf  //size N for N boards
        Stacks   []StackNode //size N-1 for N boards
        Nboards  uint16
        Spec     *CutSpec
```

```
        Areas     []Board
        NextNode uint16
}

func NewLayoutTree(spec *CutSpec) *LayoutTree {
        n := len(spec.Boards)
        return &LayoutTree{
                Spec:     spec,
                Nboards: uint16(n),
                Picks:    make([]PickLeaf, n, n),
                Stacks:   make([]StackNode, n-1, n-1),
                Areas:    make([]Board, n-1),
        }
}

//Join two boards if they're not already connected
   together.
//Two boards can be already connected either directly or
   through
//other Joins. They're connected if they belong to the
   same tree
//component.
//Boards must be referred to by their index in the
   CutSpec.
//config is a Join configuration, which specifies whether
    the
//boards are rotated or not, and whether the join is
   vertical or
//horizontal. Rotation configuration is only considered
   if the board
//to be joined hasn't been picked before. The first pick
   determines
//rotation
func (lt *LayoutTree) take(i, j uint16, config Join) bool
```

```
    {
        iRoot := lt.getLeafRoot(i)
        jRoot := lt.getLeafRoot(j)
        config = lt.fixRotationConfig(i, j, config)
        k := lt.NextNode
        if iRoot == jRoot {
                return false
        } else {
                lt.setNode(k, iRoot, jRoot, config)
                lt.setChild(iRoot, k, config.irot())
                lt.setChild(jRoot, k, config.jrot())
                lt.areaStep(int(k), lt.Spec)
                if lt.Spec.MaxWidth > 0 && config.
                   direction() == HORIZONTAL &&
                        lt.Areas[k].Width > lt.Spec.
                           MaxWidth {
                        lt.setNode(k, iRoot, jRoot,
                           config.direct(VERTICAL))
                        lt.areaStep(int(k), lt.Spec)
                }
                lt.NextNode += 1
                return true
        }
}

func (t *LayoutTree) clearNode(i, left, right uint16) {
        node := &t.Stacks[i]
        node.Left = 0
        node.Right = 0
        //direction is a bool, no value for empty
}

func (t *LayoutTree) setNode(i, left, right uint16,
   config Join) {
```

```
        node := &t.Stacks[i]
        node.Direction = config.direction()
        node.Left = left
        node.Right = right
}

func (t *LayoutTree) setChild(i, parent uint16, rot bool)
    {
        if i < t.Nboards {
                t.Picks[i].Parent = parent + 1
                t.Picks[i].Rot = rot
        } else {
                t.Stacks[i-t.Nboards].Parent = parent + 1
        }
}

func (lt *LayoutTree) rotationOnMaxWidth(i uint16, rot
   bool) (fixed bool) {
        if i > lt.Nboards || lt.Spec.MaxWidth == 0 {
                return rot
        }
        leaf := lt.Spec.Boards[i]
        if rot {
                leaf = leaf.rotated()
        }
        if leaf.Width > lt.Spec.MaxWidth {
                return !rot
        } else {
                return rot
        }
}

func (lt *LayoutTree) fixRotationConfig(i, j uint16,
   config Join) Join {
```

```
        fixed := JOIN.direct(config.direction())
        if lt.rotationOnMaxWidth(i, config.irot()) {
                fixed = fixed.irotated()
        }
        if lt.rotationOnMaxWidth(j, config.jrot()) {
                fixed = fixed.jrotated()
        }
        return fixed
}

type Fitness func(t *LayoutTree) uint

//Processes an area state from start to (non including)
   end.
//Assumes state has already been computed from 0 to start
   -1
func (t *LayoutTree) areaStep(i int, spec *CutSpec) {
        stack := t.Stacks[i]
        first := t.getBoard(stack.Left, spec.Boards, t.
           Areas)
        second := t.getBoard(stack.Right, spec.Boards, t.
           Areas)
        switch stack.Direction {
        case VERTICAL:
                t.Areas[i] = first.Vstack(second)
        case HORIZONTAL:
                t.Areas[i] = first.Hstack(second)
        }
}

//It'd be better to decouple area calculation from tree
   building
//but wee somehow need to track if the layout falls
   outside the
```

```
//spec limits (maxWidth)
func (t *LayoutTree) Area() uint {
        return t.Areas[len(t.Areas)-1].Area()
}

func (t *LayoutTree) Height() uint {
        return t.Areas[len(t.Areas)-1].Height
}

var _ Fitness = (*LayoutTree).Area
var _ Fitness = (*LayoutTree).Height

type Rect struct {
        X, Y, Width, Height uint
}
type Drawer struct {
        lt     *LayoutTree
        state []Board
}

type Drawing struct {
        Boxes []Rect
        Sheet Rect
}

func NewDrawer(lt *LayoutTree) *Drawer {
        return &Drawer{lt: lt, state: lt.Areas}
}

func (d *Drawer) Draw() *Drawing {
        nboards := len(d.lt.Spec.Boards)
        boxes := make([]Rect, nboards)
        for i, board := range d.lt.Spec.Boards {
                if d.lt.Picks[i].Rot {
```

```
                        board = board.rotated()
                }
                boxes[i].Width = board.Width
                boxes[i].Height = board.Height
        }
        d.DrawWithOffset(2*nboards-2, Board{0, 0}, boxes)
        totalArea := d.lt.Areas[nboards-2]
        sheet := Rect{0, 0, totalArea.Width, totalArea.
           Height}
        return &Drawing{Boxes: boxes, Sheet: sheet}
}

func (d *Drawer) DrawWithOffset(i int, offset Board,
   boxes []Rect) {
        nboards := len(d.lt.Spec.Boards)
        if i < nboards {
                boxes[i].X = offset.Width
                boxes[i].Y = offset.Height
        } else {
                stack := d.lt.Stacks[i-nboards]
                d.DrawWithOffset(int(stack.Left), offset,
                    boxes)
                leftOffset := d.lt.getBoard(stack.Left, d
                   .lt.Spec.Boards, d.state)
                if stack.Direction == VERTICAL {
                        offset = Board{offset.Width,
                           offset.Height + leftOffset.
                           Height}
                } else {
                        offset = Board{offset.Width +
                           leftOffset.Width, offset.
                           Height}
                }
                d.DrawWithOffset(int(stack.Right), offset
```

```
                , boxes)
        }
}

func (t *LayoutTree) getBoard(i uint16, orig []Board,
   state []Board) Board {
        if i < t.Nboards {
                if t.Picks[i].Rot {
                        return orig[i].rotated()
                } else {
                        return orig[i]
                }
        } else {
                return state[i-t.Nboards]
        }
}

//Gets the root of a leaf's tree. The root is encoded as:
//0 if the leaf is th root
//j+1 if the node j is the root
//The leaf is identified by its index in the layoutTree
   picks slice.
func (t *LayoutTree) getLeafRoot(i uint16) uint16 {
        if pick := &t.Picks[i]; pick.Parent == 0 {
                return i
        } else {
                pick.Parent = t.getNodeRoot(pick.Parent)
                return pick.Parent + t.Nboards - 1
        }
}

//Gets the root of a node's tree. The node is identified
   by
//it's 1-based index in the layoutTree stacks slice.
```

```
//returns the 1-based index of root node
//returns: 1-based node-index
func (t *LayoutTree) getNodeRoot(i uint16) uint16 {
        if node := &t.Stacks[i-1]; node.Parent == 0 {
                return i
        } else {
                node.Parent = t.getNodeRoot(node.Parent)
                return node.Parent
        }
}
```