ZTile Manual

Linus Arver

ver. 0.1.0-8-ge9c05fd

2014-01-07 23:30:25 -0800

Contents

Part I

Usage

Chapter 1

Introduction

ZTile is a library designed for handling square and hexagonal tiles in a game map.

1.1 Terminology

A map is made up of square or hexagonal **tiles**.

1.2 GHCi

You can test out some of the random graphs generated by ZTile by invoking GHCi on the `ZTile.Test` module as follows:

```
ZTile.Test> :m +Data.GraphViz
ZTile.Test> rng <- createSystemRandom
ZTile.Test> g <- randWGraph 10 0.3 [] rng :: IO (Gr Int Int)
ZTile.Test> preview g
```

This will create a random graph of 10 vertices, with each vertex having a 30% probability of having edges to all other vertices in the graph, and will output that graph with GraphViz's `preview` function, which pretty-prints the graph in a window using vector graphics. You can move around the graph with your mouse's middle mouse button, and also zoom in and out with the scroll wheel.

Chapter 2

ztile-test

Compile and run this file to test the functions in `Ztile/Test.lhs`.

```
module Main where

import Test.Framework (defaultMain)
import System.Random.MWC (withSystemRandom)

import qualified ZTile.Test as T1

main :: IO ()
main = withSystemRandom
    $ \g -> defaultMain
        [ T1.tests g
        ]
```

Part II

Source Code

Chapter 3

ZTile

```
{-# LANGUAGE RecordWildCards #-}

module ZTile where

import Data.List (intercalate, intersperse)

type Tile = (Int, Int)
```

The `Tile` type is based on a (x, y) coordinate system. This simple system can represent both square and hex tiles. For hex tiles, the third z-axis coordinate can be calculated on the fly using the x and y values, using the formula

$$z = (-x) - y \tag{3.1}$$

.

```
data Direction
    = DXPlus
    | DXMinus
    | DYPlus
    | DYMinus
    deriving (Eq, Show)

go :: Tile -> [Direction] -> Tile
go = foldl go'

go' :: Tile -> Direction -> Tile
go' (x, y) d = case d of
    DYPlus -> (x, y + 1)
    DYMinus -> (x, y - 1)
    DXPlus -> (x + 1, y)
    DXMinus -> (x - 1, y)
```

The `Direction` type represents the four ways a `Tile` can change by the smallest amount. A `Tile` can change by adding or subtracting from its x value or y value. By isolating these four possible operations into a single data type, we can more easily reason about changes to a `Tile` elsewhere.

```
data Plane
    = FlatSq
    | FlatHex
    deriving (Eq, Show)
```

The `Plane` represents the type of arrangement of tiles possible. Currently, only two arrangements are possible — `FlatSq` and `FlatHex`. `FlatSq` is a flat plane composed of regular squares (like a chess board); `FlatHex` is a flat plane composed of hexagons, but aligned so that perfect East/West movement is possible by simply changing the x value in the `Tile`.

```
data TileGeom = TileGeom
    { tgPlane :: Plane
    , tgSizeX :: Int
    , tgSizeY :: Int
    , tgTiles :: [Tile]
    } deriving (Eq)

instance Show TileGeom where
    show TileGeom{..}
        = "TileGeom { tgPlane = " ++ show tgPlane ++ ",\n"
        ++ "tgSizeX = " ++ show tgSizeX ++ ",\n"
        ++ "tgSizeY = " ++ show tgSizeY ++ ",\n"
        ++ "tgTiles =\n" ++ showTileGeom ++ "\n}"
        where
        showTileGeom = intercalate "\n" . map showTGRow $ reverse [1..tgSizeY]
        showTGRow yIdx = indent ++ intersperse ' ' (map (const 'x') [1..tgSizeX])
            where
            indent = if odd yIdx
                then " "
                else ""
```

The `TileGeom` type, or simply `TG`, is the core data type offered by ZTile. The `tgPlane` parameter describes which `Plane` type is used. The `tgSizeX` and `tgSizeY` parameters state the size, in x and y coordinate space, the tiles take up. Lastly, the `tgTiles` parameter lists every `Tile` in `TG`.

The custom `Show` instance is there to make `TG`s look easier to the human eye; in particular, the `showTileGeom` function displays the `TG` with basic ASCII-art.

```
class ZTile a where
    tiles :: a -> [Tile]
    adjacent :: a -> Tile -> [Tile]
    adjacent' :: a -> Tile -> [Tile]
    distance :: a -> Tile -> Tile -> Int
    contains :: a -> Tile -> Bool
    size :: a -> (Int, Int)
```

The ZTile class defines a set of common functions that a tile map should support:

- `tiles`: List all tiles.
- `adjacent`: Given a vertex, return all tiles that share a common edge.
- `adjacent'`: Given a vertex, return all tiles that share a common edge *or vertex*. For square tiles, this would check diagonal tiles as well. For hex tiles, as they always share a common edge, this function is the same as `adjacent`.
- `distance`: The minimum distance between two tiles, if there are no obstructions.
- `contains`: Checks if a given tile exists in the tile map.
- `size`: Return the size of the tile map, in (x, y) form.

The `ZTile` class instance for `TileGeom` is relatively straightforward. The highlight is the ease in which we describe the `adjacent` and `adjacent'` functions with the help of the `Direction` type we defined in the beginning.

```
instance ZTile TileGeom where
    tiles = tgTiles
    adjacent TileGeom{..} idx = filter (flip elem tgTiles) $ case tgPlane of
        FlatSq -> map (go' idx)
            [ DXPlus
            , DXMinus
            , DYPlus
            , DYMinus
            ]
        FlatHex -> map (go idx)
            [ [DXPlus]
            , [DXMinus]
            , [DYPlus]
            , [DYMinus]
            , [DYPlus, DXMinus]
            , [DYMinus, DXPlus]
            ]
    adjacent' tg@TileGeom{..} idx = case tgPlane of
        FlatSq -> filter (flip elem tgTiles) $ map (go idx)
            [ [DXPlus]
            , [DXMinus]
            , [DYPlus]
            , [DYMinus]
            , [DXPlus, DYPlus]
            , [DXMinus, DYMinus]
            , [DYPlus, DXMinus]
            , [DYMinus, DXPlus]
            ]
        FlatHex -> adjacent tg idx
    distance TileGeom{..} (x1, y1) (x2, y2) = case tgPlane of
```

```
        FlatSq -> max (abs $ x2 - x1) (abs $ y2 - y1)
        FlatHex -> maximum
            [ abs $ x2 - x1
            , abs $ y2 - y1
            , abs $ z2 - z1
            ]
            where
            z2 = (-x2) - y2
            z1 = (-x1) - y1
    contains TileGeom{..} = flip elem tgTiles
    size TileGeom{..} = (tgSizeX, tgSizeY)
```

Notice how the `distance` function uses the equation at 3.1 to determine the z coordinate distance between two tiles.

```
flatPlaneInit :: Plane -> Int -> Int -> TileGeom
flatPlaneInit p = case p of
    FlatSq -> flatSqInit
    FlatHex -> flatHexInit

flatSqInit :: Int -> Int -> TileGeom
flatSqInit x y
    | x < 1 || y < 1 = TileGeom
        { tgPlane = FlatSq
        , tgSizeX = 1
        , tgSizeY = 1
        , tgTiles = [(0, 0)]
        }
    | otherwise = TileGeom
        { tgPlane = FlatSq
        , tgSizeX = x
        , tgSizeY = y
        , tgTiles = [ (x', y') | x' <- [0..(x - 1)], y' <- [0..(y - 1)] ]
        }

flatHexInit :: Int -> Int -> TileGeom
flatHexInit x y
    | x < 1 || y < 1 = TileGeom
        { tgPlane = FlatHex
        , tgSizeX = 1
        , tgSizeY = 1
        , tgTiles = [(0, 0)]
        }
    | otherwise = TileGeom
        { tgPlane = FlatHex
        , tgSizeX = x
        , tgSizeY = y
        , tgTiles = buildHexes x y
```

```
        }

-- E.g., for a size x=4 and y=3, we get:
--
--  x x x x     <- row 2, an even row, so we shift the tiles left by 1 unit
--   x x x x
--  x x x x     <- row 0
buildHexes :: Int -> Int -> [Tile]
buildHexes xWidth yHeight = snd $ foldl step (0, []) ys
    where
    ys = [0..(yHeight - 1)]
    step (x, acc) y = (x', acc ++ map (flip (,) y) xs)
        where
        xs = [x..(x + (xWidth - 1))]
        -- If we encounter an odd row, decrement the starting x index for the
        -- next iteration (an even row).
        x' = if odd y
            then x - 1
            else x
```

The `flatPlaneInit` function initializes a `TG` based on the given `Plane` and (x, y) size. The helper functions `flatSqInit` and `flatHexInit` do the real work. The `flatHexInit` function's real work is done with `buildHexes`, which carefully sets each row of hex tiles with the correct x coordinate.

```
flatSqDefault :: TileGeom
flatSqDefault = flatSqInit 19 19

genTiles :: Plane -> Int -> Int -> [Tile]
genTiles p x y = case p of
    FlatSq -> tgTiles $ flatSqInit x y
    FlatHex -> tgTiles $ flatHexInit x y
```

These are some miscellaneous functions. The 19×19 size in `flatSqDefault` is an homage to the game of Go.

Chapter 4

ZTile/PathFinding

For this module, we borrow terms from computer science when describing the shortest path problem. We speak of vertices, edges, and graphs. Vertices are the points, or nodes, where we can visit (e.g., cities). Edges connect two vertices together (e.g., roads). A graph is the collection of vertices and edges; more specifically, for purposes of the `dijkstra` algorithm, it only deals with edges that are non-negative.

The `Data.List.Key` module is from the utility-ht package.

```
module ZTile.PathFinding where

import Data.List
import qualified Data.List.Key as K
import Data.Map ((!), fromList, fromListWith, adjust, keys, Map, notMember)

import ZTile.Util

data Weight
    = Finite Int
    | Infinity
    deriving (Eq, Show)
```

For pathfinding problems, we are interested in distances (the length of edges) between vertices in a graph. The weight can be either `Infinity` for unvisited vertices, or `Finite Int` for visited ones.

```
instance Ord Weight where
    compare (Finite a) (Finite b) = compare a b
    compare (Finite _) Infinity = LT
    compare Infinity (Finite _) = GT
    compare Infinity Infinity = EQ

instance Num Weight where
    Finite a + Finite b = Finite (a + b)
    Finite _ + _ = Infinity
    Infinity + _ = Infinity

```

```
    Finite a * Finite b = Finite (a * b)
    Finite _ * _ = Infinity
    Infinity * _ = Infinity

    abs (Finite a) = Finite (abs a)
    abs _ = Infinity

    signum (Finite a) = Finite (signum a)
    signum _ = Infinity

    fromInteger a = Finite (fromInteger a)
```

Because we need to perform basic math operations on the `Weight` type, we define the `Ord` and `Num` typeclass instances here.

```
type TileId = Int
type WTile = (TileId, Weight)
```

We define a `WTile` type here for weighted tiles, which could perhaps be used by the user of this package. The idea is that each tile will have a weight associated with it, and that moving from tile A to tile B will incur a movement cost that is the interpolation between the weight of A and B divided by 2, or some other scheme. It is up to the user to decide how to determine the values of the weights between two tiles, and to generate the `[(a, a, Int)]` list required to feed to `buildGraph`.

4.1 Dijkstra's Algorithm

```
buildGraph :: Ord a
    => [(a, a, Int)]
    -> Either (String, [(a, a)]) (Map a [(a, Weight)])
buildGraph edges
    | length es /= length (nub es)
        = Left ("duplicate edge weight definitions", es \\ nub es)
    | any (<0) ws
        = Left
            ( "negative weights detected"
            , map getEdge $ filter ((<0) . getWeight) edges
            )
    | otherwise = Right
        . fromListWith (++)
        $ concatMap (\(a, b, w) -> [(a, [(b, Finite w)]), (b, [])]) edges
    where
    getEdge = fstSnd3
    getWeight = thd3
    es = map getEdge edges
    ws = map getWeight edges
```

`buildGraph` generates the graph structure we will be working with, where each vertex has a list of neighboring vertices. It takes a list of directed edges, in the format (v_1, v_2, w); e.g., if it is given an edge `(LA, NY, 1000)`, this is interpreted as an arrow pointing from LA to NY (and *only* in this direction), with a weight of 1000 units.

We also check if the given list of edges makes sense, in that

- it does not contain any duplicate edge definitions, and
- it does not contain any negative weights (because Dijkstra's algorithm cannot handle negative weights).

```
dijkstra :: Ord a => Map a [(a, Weight)] -> a -> Map a (Weight, Maybe a)
dijkstra graph source = dijkstra' graph wverts verts
    where
    verts = keys graph
    wverts = fromList $ map setVertex verts
    setVertex v =
        ( v
        , (if v == source then Finite 0 else Infinity, Nothing)
        )
```

Dijkstra's algorithm. The return type is another `Map` type, where each vertex has a final weight (i.e., distance) associated with it (the shortest distance from the source vertex), as well as a `Maybe a` type, which holds the previous vertex traveled to reach this vertex from the source. Unreachable vertices will have the value `(Infinity, Nothing)`.

```
dijkstra' :: Ord a
    => Map a [(a, Weight)]
    -> Map a (Weight, Maybe a)
    -> [a]
    -> Map a (Weight, Maybe a)
dijkstra' _ finished [] = finished
dijkstra' graph finished unvisited
    = dijkstra' graph (foldl' readjust finished uNeighbors)
    $ delete u unvisited
    where
    u = K.minimum (fst . (finished !)) unvisited
    uNeighbors = graph ! u
    neighborWeight = fst (finished ! u)
    readjust vxmap (neighbor, weight)
        = adjust (min (weight + neighborWeight, Just u)) neighbor vxmap
```

We examine one unvisited vertex at a time, until the set of all unvisited vertices becomes empty (at every iteration, we call `delete` to remove the minimum-distance vertex `u` from it). `K.minimum` has type `minimum :: Ord b => (a -> b) -> [a] -> a`. That is, `u` is the vertex with the shortest distance to the source that is in the unvisited set.

```
shortestPath :: Ord a => a -> a -> Map a [(a, Weight)] -> [a]
shortestPath source dest graph
```

```
    | source == dest = [] -- ignore self-loops
    | notMember dest graph = []
    | Infinity == fst (dijkstra graph source ! dest) = [] -- dest is unreachable
    | otherwise = reverse $ traceBack dest
    where
    traceBack x = x : maybe [] traceBack (snd $ dijkstra graph source ! x)
```

To retrieve the shortest path, we simply reverse our direction from the destination.

Chapter 5

ZTile/Test

```
module ZTile.Test where

import Control.Monad
import Control.Monad.Primitive
import Data.Graph.Inductive as GI
import Data.Graph.Inductive.Example (genLNodes)
import Data.List
import Data.Maybe
import Data.Tuple
import qualified Data.Vector as V
import System.Random.MWC as MWC
import System.Random.MWC.CondensedTable
import Test.Framework
import Test.Framework.Providers.QuickCheck2
import Test.QuickCheck as Q
import Test.QuickCheck.Monadic as Q

import ZTile.PathFinding
import ZTile.Util
```

5.1 Random Graph Generation

```
type Vertex = Int
```

We abstract away all vertices to just integers.

```
data GraphProperty
    = GpNoLoops
    | GpNoBidirs
    | GpDAG
    deriving (Eq, Show)
```

We classify the kind of graph properties we are interested in.

```
randWGraph :: (DynGraph g, PrimMonad m)
    => Int -> Double -> [GraphProperty] -> MWC.Gen (PrimState m) -> m (g Int Int)
randWGraph n p gps rng
    | n < 1 = return GI.empty
    | p <= 0.0 = return $ mkGraph (genLNodes 1 n) []
    | p >= 1.0 = prepareGraph =<< mkEdges' n gps rng
    | otherwise = prepareGraph =<< randInclude rng p =<< mkEdges' n gps rng
    where
    prepareGraph es = return . mkGraph (genLNodes 1 n)
        =<< mapM addWeight
        =<< incIdxs es
    incIdxs = return . map (\(v1, v2) -> (v1 + 1, v2 + 1))
    addWeight (v1, v2) = do
        w <- genFromTable table rng
        return (v1, v2, w)
    table = tableFromWeights . V.fromList . zip [1..] $ map (1/) [0.5,1..5]
```

This is our baseline Erdős-Rényi random graph generation algorithm, with some caveats. The original algorithm considers every single possible edge in the graph, and adds it into the graph if the random number is less than p. The edges considered in our version in `randWGraph` depend on the `[GraphProperty]` list `gps`. The `GraphProperty` types are defined as follows:

- `GpDAG`: all edges point from a smaller vertex to a bigger one, and thus the appearance looks like a DAG (directed acyclic graph);
- `GpNoLoops`: all edges' endpoints point to different vertices (i.e., there is no edge from a vertex back to itself);
- `GpNoBidirs`: if there is an edge (v_1, v_2), then there is no edge (v_2, v_1) (i.e., there can only be 1 edge between any two vertices, if at all).

If the `[GraphProperty]` list is empty, we consider every single possible edge like in the original Erdős-Rényi algorithm.

The other parameters to this function are the standard ones – `n` for the total number of vertices, and `p` for the percentage in which an arbitrary edge will be created. The `incIdxs` function simply increments all vertices in the graph by 1, to make it compatible with FGL's vertex indexing scheme, which counts starting from 1, not 0. The `table` variable makes it so that there is a greater likelihood to select weights with lower numbers than those with higher numbers, and also, the range of weights is 1 to 10 (as the length of `[0.5,1..5]` is 10).

```
mkEdges':: PrimMonad m
    => Int -> [GraphProperty] -> MWC.Gen (PrimState m) -> m [(Vertex, Vertex)]
mkEdges' n gps rng
    | elem GpDAG gps = return $ edgesDAG n
    | allElem [GpNoLoops, GpNoBidirs] gps = randFstSnd rng $ edgesSimple n
    | elem GpNoLoops gps = return $ edgesNoLoops n
```

```
    | elem GpNoBidirs gps = return $ edgesDAGselfLoops n
    | otherwise = return $ edgesAll n

edgesAll :: Int -> [(Vertex, Vertex)]
edgesAll n = [(x, y) | x <- [0..(n - 1)], y <- [0..(n - 1)]]

edgesNoLoops :: Int -> [(Vertex, Vertex)]
edgesNoLoops = filter (uncurry (/=)) . edgesAll

edgesDAG :: Int -> [(Vertex, Vertex)]
edgesDAG n = thd3 $ foldl' step ((0, 1), 1, []) [1..(div (n * (n + 1)) 2)]
    where
    step (e@(x, y), yStart, acc) _
        | y == n = ((x + 1, yStart + 1), yStart + 1, acc)
        | otherwise = ((x, y + 1), yStart, e:acc)

edgesDAGselfLoops :: Int -> [(Vertex, Vertex)]
edgesDAGselfLoops n = edgesDAG n ++ selfLoops
    where
    selfLoops = [(x, x) | x <- [0..(n - 1)]]

edgesSimple :: Int -> [((Vertex, Vertex), (Vertex, Vertex))]
edgesSimple n = zip (edgesDAG n) . map swap $ edgesDAG n
```

`mkEdges'` returns the set of edges with the given desired properties. The real workhorses are the various edge generation functions. `edgesDAG` is the most complex one; there are two main ideas behind it: first, do not generate self-loop edges, and second, given any two vertices, choose the edge direction that goes from the smaller vertex to the greater one. Here are some sample values:[1]

Vertices	Edge list length	Edges
2	1	[(0, 1)]
3	3	[(0,1),(0,2),(1,2)]
4	6	[(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]
5	10	[(0,1),(0,2),(0,3),(0,4),(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]

`edgesDAG` thus generates all unique edges, in the sense that (v_2, v_1) is discounted as a duplicate of (v_1, v_2). An interesting tidbit is that this list grows in the same manner as that of triangular numbers – hence the formula

$$\text{EdgeListLength} = \frac{n(n+1)}{2}$$

used to generate the list fed into `foldl'`.

[1]The contents of each list have been reversed for clarity.

```
randInclude :: (Ord a, PrimMonad m, Variate a)
    => MWC.Gen (PrimState m) -> a -> [b] -> m [b]
randInclude rng p = foldM f []
    where
    f acc edge = do
        theta <- uniform rng
        return $ if theta < p
            then edge:acc
            else acc
```

`randInclude` randomly selects items from a list, based on a threshold percentage `p`.

```
randFstSnd :: PrimMonad m => MWC.Gen (PrimState m) -> [(a, a)] -> m [a]
randFstSnd rng = foldM f []
    where
    f acc pair = do
        theta <- uniformR ((0, 1) :: (Int, Int)) rng
        return $ (if theta == 0 then fst else snd) pair:acc
```

`randFstSnd` randomly chooses between either the first or second item in a tuple.

5.2 Tests

We now test ZTile's `shortestPath` function, to see if it matches FGL's version. Because there can be multiple shortest paths in a graph, we only check to see if the chosen path lengths are the same in `prop_shortestPath`.

```
tests :: GenIO -> Test
tests g = testGroup "Dijkstra"
    [ testProperty "prop_shortestPath" $ prop_shortestPath g
    ]

prop_shortestPath :: GenIO -> Property
prop_shortestPath rng = monadicIO $ do
    g <- Q.run $ randWGraph 10 0.3 [GpNoLoops] rng :: PropertyM IO (Gr Int Int)
    unless (null (fglPath g) && null (ztPath g)) $
        assert (ztPathCost g == fglPathCost g)
    where
    (a, b) = (1, 10)
    fglPath = sp a b
    fglPathCost = Finite . spLength a b
    ztPath g = shortestPath a b g'
        where
        g' = (\(Right x) -> x) . buildGraph $ labEdges g
    ztPathCost g = Finite . sum . map (getEdgeWeight g) $ pathEdges g
    getEdgeWeight g e = fromJust $ lookup e edges'
        where
```

```
        edges' = map (\(x, y, w) -> ((x, y), w)) $ labEdges g
    pathEdges g = zip (ztPath g) . drop 1 $ ztPath g
```

The condition that always returns without any result if both `fglPath` and `ztPath` are empty is only there to work around a likely bug in GHC 7.6.3, which makes the code here crash with “Prelude.head: empty list”.

Chapter 6

ZTile/Util

```
module ZTile.Util where

allElem :: Eq a => [a] -> [a] -> Bool
allElem members clan = all (flip elem clan) members

fst3 :: (a, b, c) -> a
fst3 (a, _, _) = a

snd3 :: (a, b, c) -> b
snd3 (_, b, _) = b

thd3 :: (a, b, c) -> c
thd3 (_, _, c) = c

fstSnd3 :: (a, b, c) -> (a, b)
fstSnd3 (a, b, _) = (a, b)
```