ЛАБОРАТОРНАЯ РАБОТА №8-9

РЕАЛИЗАЦИЯ НА ЯЗЫКЕ CLOJURE ГЕНЕРАЦИИ ЛАБИРИНТА И ВИЗУАЛИЗАЦИИ ПОСЛЕДОВАТЕЛЬНОГО ЕГО ПРОХОЖДЕНИЯ

Индивидуальный вариант задаёт размеры лабиринта — (40+№)x(40+№).

Генерация лабиринтов

Рассмотрим еще один пример: алгоритм генерации лабиринтов Уилсона (Wilson's maze generation algorithm)[1].

Алгоритм Уилсона – это алгоритм «вырезания». Он берет исходный «лабиринт», состоящий из клеток, окруженных с четырех сторон стенами, и вырезает в нем проходы, удаляя некоторые стены. Этот алгоритм имеет следующий принцип действия:

1. Выбирается случайная клетка и помечается как «посещенная».
2. Выбирается случайная клетка, еще не помеченная как «посещенная» – если такой клетки нет, возвращается готовый лабиринт.
3. Из вновь выбранной клетки случайным образом прокладывается маршрут, пока не будет встречена клетка, помеченная как «посещенная» – если во время прокладки маршрута одна и та же клетка встречается на пути несколько раз, для нее всегда запоминается направление, в котором эта клетка покидалась в последний раз.
4. Все клетки, встретившиеся во время прокладки маршрута, помечаются как «посещенные» и удаляются стены, соответствующие «направлениям выхода» из каждой клетки в последний раз.
5. Возврат к шагу 2.

В общем случае для представления лабиринта алгоритм генерации лабиринта использует матрицу, каждый элемент которой представляет собой битовую маску, указывающую, какие стены вокруг данной клетки остались на месте. Проницательный читатель может заметить, что при таком подходе состояние каждой стены хранится в двух местах – в смежных клетках, для которых эта стена является общей.

Кроме того, алгоритм Уилсона требует запоминать направление выхода для каждой клетки – еще один источник сложности, если попытаться впихнуть всю информацию в битовую маску, или «состояние клетки». Существует еще ряд причин, почему алгоритм Уилсона считается сложным для реализации. Однако, вооруженные языком Clojure и знанием особенностей «реляционного» моделирования, мы можем минимизировать сложности с помощью множеств, ассоциативных массивов и естественных идентификаторов!

Кому интересна тема генераторов лабиринтов, я рекомендую обратиться к серии иллюстрированных статей Джеймиса Бака (Jamis Buck) по адресу: http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap.

Если внимательно изучить описание этого алгоритма, можно выделить несколько основных его элементов: клетки, признак, что клетка «посещалась», сам лабиринт, случайный маршрут и направление выхода. Посмотрим, как лучше представить их на языке Clojure.

На данный момент для вас не будет сюрпризом, что клетки должны быть представлены вектором с координатами, `[x y]`.

Стоп! Если представление клетки сводится к ее координатам, как тогда хранить дополнительный признак, что клетка «посещалась»? Ответ прост: этот признак будет храниться отдельно от координат, потому что он не принадлежит к ним, координаты – это лишь идентификатор (естественный). Следовательно, необходимо создать множество координат посещавшихся клеток.

Сам лабиринт состоит из клеток, окруженных стенами, и каждая стена разделяет две смежные клетки, поэтому стены должны быть представлены парами координат смежных ячеек, то есть, вектор `[[0 0] [1 0]]` в этом случае мог бы обозначать стену между клетками с координатами `[0 0]` и `[1 0]`. Беда в том, что ту же стену можно представить вектором `[[1 0] [0 0]]`. Поскольку порядок следования ячеек в парах не имеет значения, эти пары следует хранить в виде неупорядоченной коллекции... такой как множество. Таким образом множество `#{[0 0] [1 0]}` будет являться уникальным естественным идентификатором единственной стены. Лабиринт – это множество стен, поэтому вполне естественно будет представить его в виде множества[1]!

Маршрут представить совсем несложно – это обычная последовательность координат клеток. Однако в данном случае понятие *последовательность* имеет более широкое толкование: для этой цели подойдет любой последовательный тип, включая сами последовательности, а также векторы и списки.

И последний элемент – направление выхода. Направление выхода определяется клеткой, откуда осуществляется выход, и клеткой, куда производится вход, то есть, направление – это пара координат клеток `[from to]`, но, в отличие от стен, порядок следования координат имеет значение, поэтому для преставления направления выхода будут использоваться векторы.

---

[1] Лабиринт в этом случае будет представлять собой множество двухэлементных множеств с координатами клеток. Не позволяйте структуре данных вызывать головокружение: не заглядывайте на всю глубину вложенности, в каждый конкретный момент думайте только об одном уровне.

Теперь, определившись со структурами данных, можно перейти к программному коду:

```
(defn maze
  "Возвращает случайный вырезанный лабиринт; стены - это множество
   2-элементных множеств #{a b}, где a и b - координаты.
   Возвращаемый лабиринт - это множество удаленных стен."
  [walls]
  (let [paths (reduce (fn [index [a b]]                                    ❶
                        (merge-with into index {a [b] b [a]}))
                {} (map seq walls))                                        ❷
        start-loc (rand-nth (keys paths))]                                 ❸
    (loop [walls walls
           unvisited (disj (set (keys paths)) start-loc)]                  ❹
      (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]            ❺
        (let [walk (iterate (comp rand-nth paths) loc)                     ❻
              steps (zipmap (take-while unvisited walk) (next walk))]      ❼❽
          (recur (reduce disj walls (map set steps))                       ❾
            (reduce disj unvisited (keys steps))))
        walls))))
```

❶ `paths` – это индекс (ассоциативный массив) с направлениями переходов из одной клетки в другую (в виде векторов, см. ❻).

❷ `(map seq walls)` преобразует множество стен в последовательность, чтобы их можно было деструктурировать с помощью `[a b]`[1].

❸ Конструктивно `(keys path)` содержит все координаты, таким образом `rand-nth` возвращает координаты начальной клетки для построения маршрута.

❹ Вместо *множества посещавшихся клеток*, в программе используется его дополнение – *множество не посещавшихся клеток*, потому что в противном случае программный код получился бы сложнее (см. ❺ и ❼).

❺ Здесь вызов `seq` преследует две цели: убедиться, что множество непустое, и получить последовательное представление, чтобы можно было задействовать `rand-nth`. Если бы вместо *не посещавшихся* использовалось множество *посещавшихся* клеток, вызов `(seq unvisited)` пришлось бы заменить на `(seq (remove visited (keys paths)))`.

❻ Вызов `(iterate (comp rand-nth paths) loc)` выполняет бесконечный обход клеток в случайных направлениях: в вызов передаются координаты клетки, к ним применяется `paths`, чтобы получить вектор смежных клеток, из которых с помощью `rand-nth` выбирается одна. Если бы `paths` возвращала множества, а не последовательность (например, вектор), тогда пришлось бы использовать форму `(comp rand-nth seq paths)`.

❼ `(take-while unvisited walk)` обеспечивает обход, пока не будет встречена посещавшаяся клетка (но не включает ее). Если бы в реализации использовалось множество посещавшихся клеток, тогда вместо `(take-while unvisited walk)` пришлось бы использовать `(take-while (complement visited) walk)`.

❽ `(next walk)` – выполняется до бесконечности, но `(take-while unvisited walk)` нет, поэтому `zipmap` получает только первые n элементов `(next walk)` (где n – это `(count (take-while unvisited walk))`). Первые n элементов, возвращаемых `(next walk)`, – это последовательность координат точек случайного маршрута без начальной точки, но с первой попавшейся на пути посещавшейся клеткой. Поскольку две последовательности оказываются сдвинуты относительно друг друга на один элемент, каждая получившаяся пара ключ/значение превращается в определение *направления*. При создании ассоциативного массива из этих пар, для каждого данного ключа будет сохраняться только самое последнее *направление выхода. Элементы в окончательном ассоциативном массиве хранят последние направления выхода* для каждой клетки, встретившейся во время обхода.

❾ `(map set steps)` преобразует направления (элементы ассоциативного массива) в стены в лабиринте (множества), которые нужно удалить.

Для проверки этой замечательной реализации необходимо создать две вспомогательные функции: `grid`, создающую заготовку лабиринта, где каждая клетка окружена четырьмя стенами, и `draw`, отображающая лабиринт (в данном случае с помощью Swing `JFrame`), как показано на рис. 3.10:

```
(defn grid
  [w h]
  (set (concat
         (for [i (range (dec w)) j (range h)] #{[i j] [(inc i) j]})
         (for [i (range w) j (range (dec h))] #{[i j] [i (inc j)]}))))

(defn draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
      (doto (proxy [javax.swing.JPanel] []

              (paintComponent [^java.awt.Graphics g]
                (let [g (doto ^java.awt.Graphics2D (.create g)
                          (.scale 10 10)
                          (.translate 1.5 1.5)
                          (.setStroke (java.awt.BasicStroke. 0.4)))]
                  (.drawRect g -1 -1 w h)
                  (doseq [[[xa ya] [xb yb]] (map sort maze)]
                    (let [[xc yc] (if (= xa xb)
                                    [(dec xa) ya]
                                    [xa (dec ya)])]
                      (.drawLine g xa ya xc yc))))))
        (.setPreferredSize (java.awt.Dimension.
                             (* 10 (inc w)) (* 10 (inc h))))))
    .pack
    (.setVisible true)))

(draw 40 40 (maze (grid 40 40)))
```

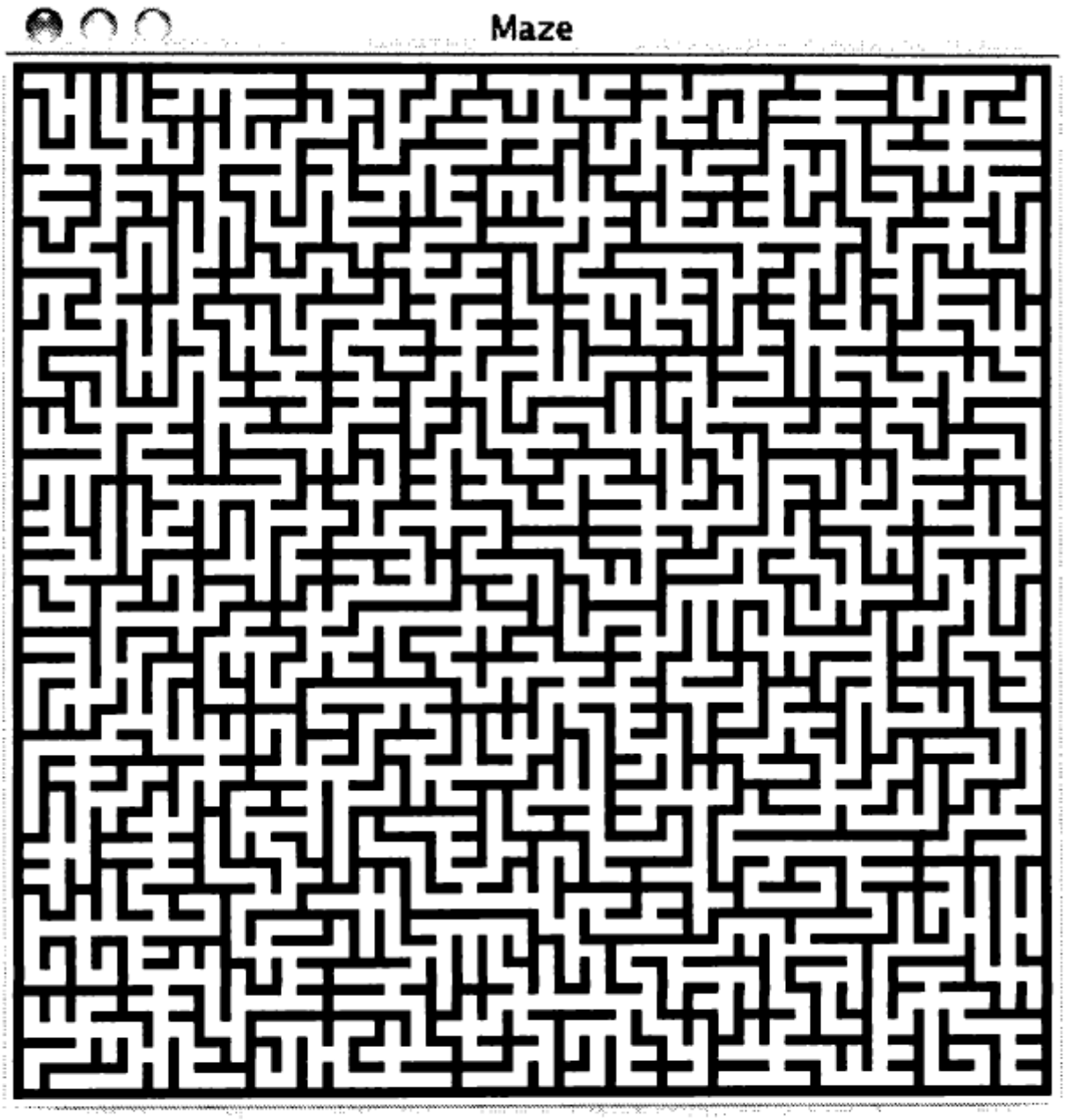

Рис. 3.10. Вид сгенерированного лабиринта

Истинный алгоритм Уилсона

В действительности мы немного схитрили и функция `maze` не является точной реализацией алгоритма Уилсона.

Когда при прокладке случайного маршрута встречается клетка, имеющаяся на графе (в лабиринте), мы добавляем целое дерево, состоящее из всех клеток, встретившихся на пути, а не ветвь дерева, тянущуюся от начальной точки до конечной. Очевидно, что наш алгоритм быстрее, потому что за один раз он добавляет в лабиринт больше клеток. Однако выигрышной особенностью алгоритма Уилсона является то обстоятельство, что все лабиринты являются одинаково вероятностными.

Эмпирическим путем (сравнением распределения лабиринтов, сгенерированных обоими алгоритмами, нашим и Уилсона) установлено, что эта особенность присуща и нашему варианту, но мы не доказали это формально и не определяли его сложность относительно времени. Это мы оставляем читателям, как самостоятельное упражнение[1].

За нашей хитростью стоит интересная история: мы почти случайно выбрали этот алгоритм – его проще было реализовать на языке Clojure, и он так и просился, чтобы реализовать его.

Реализация истинного алгоритма Уилсона мало чем отличается. Она содержит всего две дополнительные строки:

```
(defn wmaze
  «Оригинальный алгоритм Уилсона.»
  [walls]
  (let [paths (reduce (fn [index [a b]]
                        (merge-with into index {a [b] b [a]}))
                {} (map seq walls))
        start-loc (rand-nth (keys paths))]
    (loop [walls walls unvisited (disj (set (keys paths)) start-loc)]
      (if-let [loc (when-let [s (seq unvisited)] (rand-nth s))]
        (let [walk (iterate (comp rand-nth paths) loc)
              steps (zipmap (take-while unvisited walk) (next walk))
              walk (take-while identity (iterate steps loc))      ❶
              steps (zipmap walk (next walk))]                    ❷
          (recur (reduce disj walls (map set steps))
            (reduce disj unvisited (keys steps))))
        walls))))
```

❶ Трассирует только одну «ветвь» случайного маршрута, начиная от клетки `loc`.

❷ Преобразует маршрут в ассоциативный массив элементов `[from-loc to-loc]`.

Говоря формальным языком, алгоритм Уилсона генерирует связующие деревья графов. В оригинальной статье[1], приводится псевдокод реализации, в котором, несмотря на его императивность, используются множества и ассоциативные массивы, но он опирается на синтетические идентификаторы (номера узлов). Эта реализация по-прежнему расценивается как весьма выразительная и ясная, но была забыта большинством разработчиков генераторов лабиринтов.

В качестве награды, возможно вы заметили, что, что подобно функции `step` выше, `maze` не зависит от фактического значения, определяющего клетку[2]. Как следствие, `maze` может генерировать лабиринты с любой топологией: гексагональные, N-мерные и так далее. В качестве доказательства такой универсальности, ниже представлены функции создания пустых и отображения готовых лабиринтов с гексагональной сеткой, изображенного на рис. 3.11:

```
(defn hex-grid
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0) (* 2 w) 2)]
                        [x y]))
        deltas [[2 0] [1 1] [-1 1]]]
    (set (for [v vertices d deltas f [+ -]
               :let [w (vertices (map f v d))]
               :when w] #{v w}))

(defn- hex-outer-walls
  [w h]
  (let [vertices (set (for [y (range h) x (range (if (odd? y) 1 0) (* 2 w) 2)]
                        [x y]))
        deltas [[2 0] [1 1] [-1 1]]]
    (set (for [v vertices d deltas f [+ -]
               :let [w (map f v d)]
               :when (not (vertices w))] #{v (vec w)}))
(defn hex-draw
  [w h maze]
  (doto (javax.swing.JFrame. "Maze")
    (.setContentPane
```

[1] Дэвид Брюс Уилсон (David Bruce Wilson), «Generating Random Spanning Trees More Quickly than the Cover Time» http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.8598.

[2] Если оно не равно `nil` или `false`.

```
    (doto (proxy [javax.swing.JPanel] []
            (paintComponent [^java.awt.Graphics g]
              (let [maze (into maze (hex-outer-walls w h))
                    g (doto ^java.awt.Graphics2D (.create g)
                        (.scale 10 10)
                        (.translate 1.5 1.5)
                        (.setStroke (java.awt.BasicStroke. 0.4
                                     java.awt.BasicStroke/CAP_ROUND
                                     java.awt.BasicStroke/JOIN_MITER)))
                    draw-line (fn [[[xa ya] [xb yb]]]
                                (.draw g
                                  (java.awt.geom.Line2D$Double.
                                   xa (* 2 ya) xb (* 2 yb))))]
                (doseq [[[xa ya] [xb yb]] (map sort maze)]
                  (draw-line
                    (cond
                      (= ya yb) [[(inc xa) (+ ya 0.4)]
                                 [(inc xa) (- ya 0.4)]]
                      (< ya yb) [[(inc xa) (+ ya 0.4)] [xa (+ ya 0.6)]]
                      :else [[(inc xa) (- ya 0.4)]
                             [xa (- ya 0.6)]])))))))
      (.setPreferredSize (java.awt.Dimension.
                          (* 20 (inc w)) (* 20 (+ 0.5 h))))))
  .pack
  (.setVisible true)))

(hex-draw 40 40 (maze (hex-grid 40 40)))
```

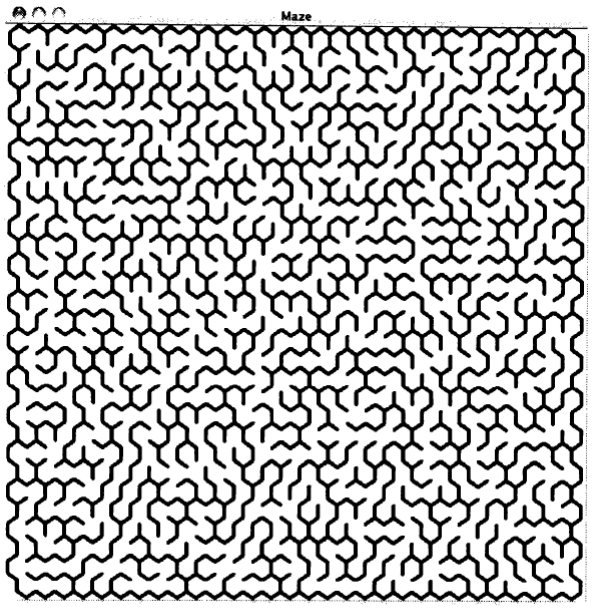

Рис. 3.11. Вид сгенерированного лабиринта

ЗАДАНИЕ

1. Реализовать лабиринт Уилсона на языке Clojure для двух видов клеток — квадратных и гексогональных.
2. Реализовать алгоритм прокладывания пути из левой верхней точки до правой нижней. Осуществиить её динамическое отображение.