****

Universidad Nacional

Facultad de Ciencias Exactas y Naturales

Escuela de Informática

EIF-400: Paradigmas de Programación

Algoritmos para generación y solución eficiente de laberintos

Integrantes:

* Andrés Hernández Bravo
* Jose Joaquín Rojas Cortés
* Alejandro Vega Lacayo
* Javier Zamora Calvo

Profesor: Carlos Loría Sáenz

II Ciclo 2016

**Introducción**

El presente proyecto por desarrollar se define como la realización de una aplicación Cliente-Servidor bajo el modelo de SPA (Single Page Application), basado en el enfoque FRP (Functional Reactive Programming). Dicha aplicación debe generar laberintos de forma aleatoria por medio de un algoritmo que permita variar las dimensiones, complejidad y diseño del laberinto por generar. Luego de una amplia investigaciones decidimos usar el algoritmo llamado “Maze Generation Algorithm with Backtracking” que más adelante se detallará como debe ser. Asimismo explicaremos cómo implementamos este algoritmo usando JavaScript y comentaremos las razones por las que elegimos este algoritmo entre muchos otros que se pueden encontrar en la red.

**Maze Generation Algorithm with Backtracking**

Elegimos este algoritmo por la enorme simplicidad que tiene al ser un candidato perfecto para usar teoría de grafos y recursividad. A continuación explicaremos paso por paso cómo funciona y qué recursos necesita:

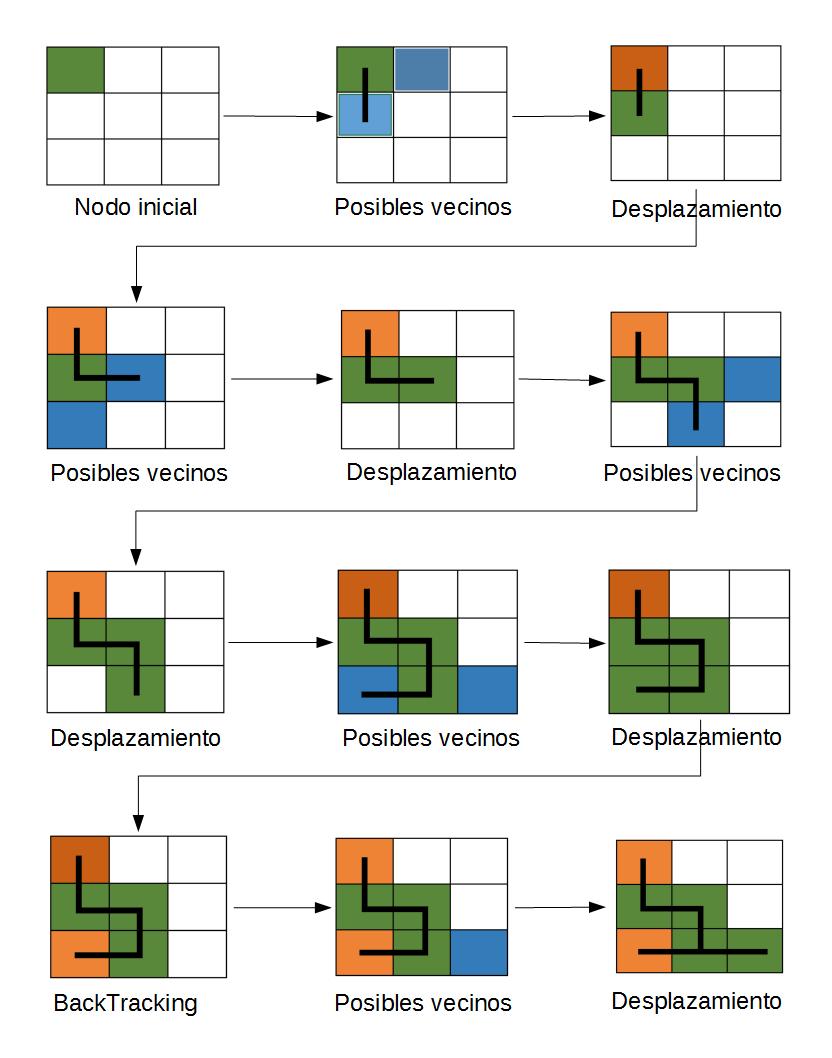
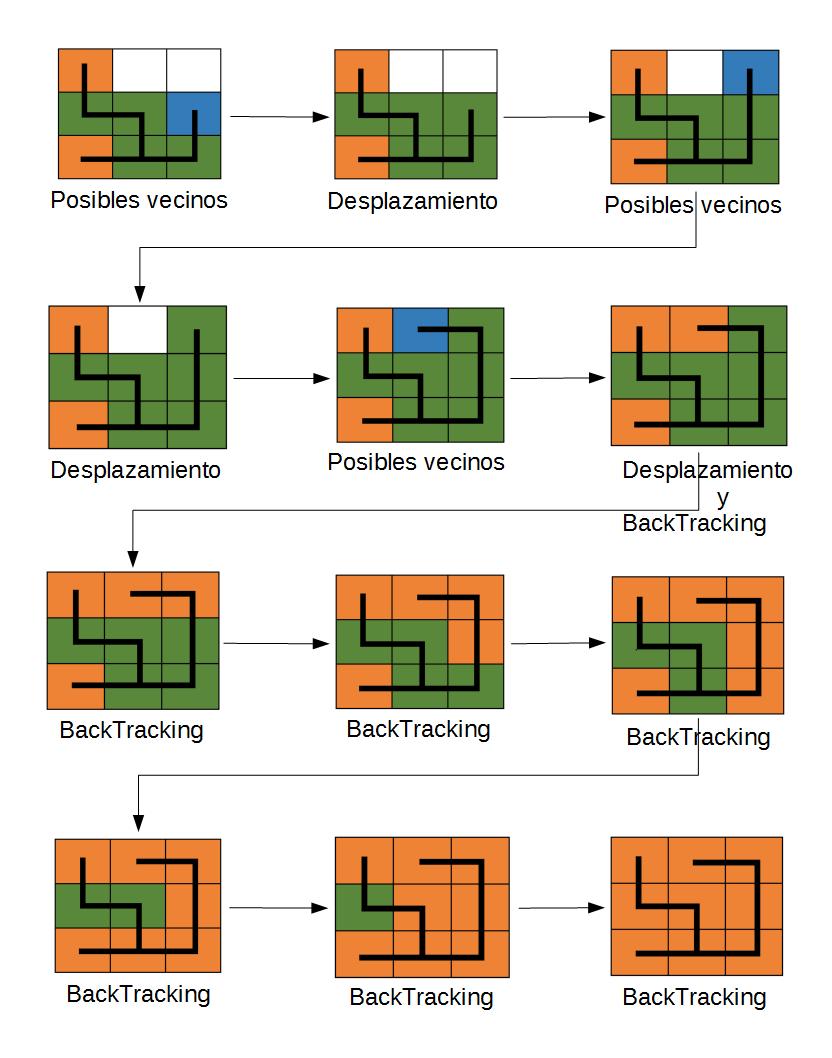
* ***Pasos a seguir:***

1. Se marcar una celda inicial en la matriz para que sea el punto de entrada al laberinto.
2. De la celda en la que nos ubicamos definimos cuáles son sus vecinos no diagonales (otras celdas a la par de la que nos encontramos) y que además dichos vecinos no hayan sido visitadas anteriormente.
3. Si existe al menos un vecino no visitado nos desplazamos marcando el camino, si existe más de uno el desplazamiento debe ser aleatorio.
4. Si nos ubicamos en una celda que no tenga vecinos no visitados, nos devolvemos por donde vinimos y regresamos al punto #2.

* ***En resumen:*** Nos desplazamos ente las celdas de la matriz mientras tengamos una celda no visitada a la que podemos movernos, mientras nos desplazamos vamos marcando el camino y si llegamos a un punto en el que no encontramos celdas por visitar, retrocedemos en busca de otros caminos.
* ***Requisitos:***

1. Se necesita definir una matriz (preferiblemente cuadrada).
2. Se necesita llevar un registro de movimiento para hacer backtracking.
3. Se necesita marcar el camino y las celdas que se van visitando.

* ***Ejemplo visual:***

*** ***

**Implementación del algoritmo a nivel de código.**

* ***Entidades.***
  + *Nodo:* Es la representación de cada una de las celdas de la matriz, se le llamó “Nodo” para hacer uso de los conceptos en la teoría de grafos. Entre los atributos del Nodo están los nodos con los que limita, un arreglo que define para dónde se puede desplazar el jugador (este arreglo se usa hasta que la matriz pasa por el algoritmo del laberinto) y coordenadas del eje X y Y además del tamaño en pixeles para luego dibujarse (estas coordenadas se usan en el canvas de HTML5, cada Nodo sabe en dónde dibujarse, pero no sabe cómo hacerlo).
  + *Matriz:* Es el conjunto de nodos (celdas) de un tamaño definido, nuestras matrices son cuadradas para no afectar el diseño de los laberintos puesto que no luciría agradable un laberinto rectangular, por ejemplo uno de cinco filas por dos columnas. Los atributos de la Matriz son el Nodo por el que comienza el laberinto, un arreglo que posee la solución del laberinto (este arreglo se genera mientras se ejecuta el algoritmo que genera el laberinto) y la dimisión que tiene dicha matriz.
* ***Algoritmo.***
  + *Generación de la matriz:* Básicamente enlazamos un conjunto de nodos uno por uno indicándole a cada Nodo cuáles son sus vecinos de forma recursiva. El O(n) de este método es “n \* n” (siendo “n” la dimensión).
  + *Generación del laberinto:* Pasamos una matriz ya terminada por el método que genera el laberinto, este método toma un Nodo como punto inicial y tomando en cuenta sus vecinos se desplaza marcando un camino (usando el arreglo dentro de los nodos), si se topa con un Nodo sin más opciones de desplazamiento hace backtracking usando una pila (no mutable) que lleva el registro de desplazamiento desde el Nodo por el que se comenzó hasta el actualmente visitado. El backtracking es el método que asegura buscar caminos previamente no tomados en cuenta y así no dejar ningún Nodo sin visitar. Cada vez que la pila de desplazamiento es usada para hacer backtracking evalúa su tamaño y así se define el final del laberinto asegurando ser el punto más largo de alcanzar desde el primer Nodo. Al final, la versión más extensa de esa pila es la solución para llegar desde el Nodo inicial hasta el final. La pila no es mutable puesto que pasa siendo copiada y el método es completamente recursivo por lo que las variables no tienen estados. El O(n) de este algoritmo tiende a “n \* n” (siendo “n” la dimensión de la matriz) puesto que recorre toda la Matriz Nodo por Nodo.
  + *Resuelve laberinto:* Mientras se genera el laberinto se guarda un arreglo con coordenadas que equivalen a la solución del mismo, el método “autoControl” sigue el camino que le indica el arreglo y así muestra la solución del laberinto. Usando promesas y setTimeout controlamos el tiempo en que se muestra el laberinto resolviéndose, pero el O(n) de este método es completamente lineal puesto que conoce la respuesta.
* ***Nota****:* Ya sea en la construcción de la matriz, la elaboración del laberinto o resolviendo el juego, la página nunca se pega puesto que usa promesas y workers.
* ***Visualización:***
  + *¿Cómo se dibuja el laberinto?*

Cada Nodo tiene un tamaño, eje X y eje Y como atributos, simplemente pasamos los Nodos (uno por uno) por un método (recursivo) que usando sus coordenadas y tamaño lo dibuja en una etiqueta canvas de HTML5 y tomando en cuenta las conexiones que tiene cada Nodo se marcan las líneas entre los nodos a los que no se puede desplazar. El O(n) de este algoritmo que dibuja el laberinto es “n \* n” (siendo “n” la dimensión de la matriz) puesto que se recorre la Matriz dibujando sus nodos uno por uno.

* ***Análisis del O(n).***
  + Como ya se mencionó en la previa explicación, usamos tres algoritmos con O(n) igual a “n \* n” (siendo “n” la dimensión de la matriz), por lo tanto el O(n) final de la construcción y presentación del laberinto tiende a “3 \* n \* n”.

**Conclusiones**

A partir de distintas búsquedas de algoritmos para generar laberintos como: “el algoritmo de Kruskal Aleatorio”, en el cual se crea una lista de todas las paredes y un conjunto para cada celda, con el fin de recorrer cada pared de forma aleatoria, y si las paredes divididas pertenecen a grupos distintos, retira la pared actual y la une de manera aleatoria. Asimismo, se tomó en consideración otro algoritmo: “el algoritmo de división recursiva”; el cual divide espacios, llamados cámaras, colocando paredes y, aleatoriamente, abriendo caminos entre estas, finalmente se continúa con este mismo proceso hasta llegar a un punto central indivisible el cual equivale al final del laberinto.

Al final de la investigación se resolvió usar “el algoritmo de Backtracking Recursivo”, debido a que es unos de los algoritmos de generación de laberintos más impredecibles que, actualmente, son de dominio público y que se adapta a las necesidades de este proyecto.

**Referencias**

Easy Learn Tutorial (2014). *Maze Generation Algorithm - Recursive Backtracker*. Recuperado de: https://www.youtube.com/watch?v=elMXlO28Q1U.

Buck J., (2011). *The Buckblog*. Recuperado de http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm.

Buck J., (2011). *The Buckblog*. Recuperado de http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm.