Universidad Nacional de Costa Rica

Facultad de Ciencias Exactas y Naturales

Escuela de Informática

Paradigmas de Programación

Proyecto 1

Grupo 1

Omar Segura Villegas
Carlos Artavia Pineda
Fabián Hernández Chavarría
Andrey Campos Sánchez

Contenidos

1 Introducción **3**

2 Problema planteado **4**

3 Resolución **4**

4 Algoritmos importantes **4**

5 Conclusiones **8**

6 Referencias **9**

1 Introducción

El siguiente es el reporte del primer proyecto del curso Paradigmas de Programación, donde se desarrolló una aplicación tipo cliente-servidor utilizando el modelo SPA (single page application), y también utilizando diferentes herramientas y tecnologías como Node.js y Express.js para el servidor junto con la base de datos MongoDB.
Cabe resaltar que, se utilizó el patrón de diseño MVC (modelo, vista, controlador) tanto en el servidor como en el cliente donde se refleja una sólida estructura de proyecto, todo esto enfocado en los principios de programación funcional.

2 Problema planteado

- Generar en el cliente o servidor, a solicitud del usuario, un laberinto automáticamente de distinta complejidad usando algoritmos eficientes.
- Resolver un laberinto generado de manera manual o automática a solicitud del usuario mostrando en forma apropiada la solución.
- Salvar (persistir) el estado de la aplicación local o remotamente para uso futuro según el usuario lo demande.

3 Resolución

La manera de resolución fue la planteada en clase por el profesor, una aplicación web de tipo cliente-servidor, donde en el cliente se utilizó el estándar de paginas web HTML con el modelo-vista-controlador. En la vista se muestran una serie de botones, añadidos con bootstrap, framework de CSS, que responden a una serie de eventos que envían peticiones de forma asíncrona al servidor, lo cual hace que la página sea concurrente y no se bloquee el hilo de ejecución principal del navegador.
Para el manejo de eventos, se utilizó jQuery, una biblioteca de Javascript para simplificar la manera de interactuar con los elementos del DOM.
Al enviar las peticiones al servidor, se aplican promesas de ES6 para evitar el *código spaguetti* con los callbacks y para visualizar mejor el flujo de trabajo de los algoritmos.
Con respecto al servidor se empleó express como framework de Node.js para la infraestructura del servidor, y por último se implementó la persistencia de datos en MongoDB. A continuación un esquema que ilustra lo anteriormente mencionado.

4 Algoritmos importantes

La generación del laberinto se resolvió por medio del algoritmo de backtracking como idea principal, donde se tiene una clase "Maze" que a su vez contiene un grid que es un conjunto de objetos celda que se construyen a partir del nivel que el usuario seleccionó, por medio de tamaños predeterminados previamente.
El algoritmo va generando celdas en posición aleatoria hasta que se cumpla

con el tamaño del grid, hay que tomar en cuenta que cada celda tiene "paredes" que son los cuatro lados de la celda y por medio de estas se define si se dibujan o no con algoritmos de ayuda como la función "quitar paredes", que elimina las paredes necesarias para que tome forma de laberinto.

```
this.drawMaze = (grid, tamano) => {
  grid[0].visited = true;
  let backtracking = (actual, stack) => {
    let revisaVecino = next => {
      if(next){
        next.visited = true;
        stack.push(actual);
        this.quitarParedes(actual, next);
        backtracking(next, stack);
      }
      else if(stack.length > 0)
        backtracking(stack.pop(), stack);
    }
    revisaVecino(actual.checkNeighbors(grid, [], tamano));
  }
  backtracking(grid[0], []);
  return grid;
  }

  this.quitarParedes = (current, next)=>{ //Falta FP
    let x = current.i - next.i;
    if(x === 1){ //Si el vecino esta a la izquierda
      current.paredes[3] = false; //borrar izquierda
      next.paredes[1] = false;   //borrar derecha
    }
    else if(x === -1){
      next.paredes[3] = false; //borrar la izquierda
      current.paredes[1] = false; //borra la derecha
    }
    let y = current.j - next.j;
    if(y === 1){ //Si el vecino esta a abajo
      current.paredes[0] = false; //borrar bottom
      next.paredes[2] = false;   //borrar top
    }
    else if(y === -1){ //Si el vecino esta arriba
      next.paredes[0] = false; //borrar bottom
      current.paredes[2] = false; //borra top
    }
  }
```

Figura 1: clase Maze.js

```
    this.checkNeighbors = (grid, neighbors, dif) => {
      let top = grid[index(this.i,this.j-1,dif)];
      let rigth = grid[index(this.i+1,this.j,dif)];
      let bottom = grid[index(this.i,this.j+1,dif)];
      let left = grid[index(this.i-1,this.j,dif)];

      if(top && !top.visited)
        neighbors.push(top);
      if(rigth && !rigth.visited)
        neighbors.push(rigth);
      if(bottom && !bottom.visited)
        neighbors.push(bottom);
      if(left && !left.visited)
        neighbors.push(left);

      //Retorna un vecino aleatorio, sino tiene devuelve undefined
      return (neighbors.length > 0) ? neighbors[Math.floor(Math.random() * neighbors.length)] : undefined;
    }

    this.checkWalls = (grid, neighbors, dif) => {
      let top = grid[index(this.i,this.j-1,dif)];
      let rigth = grid[index(this.i+1,this.j,dif)];
      let bottom = grid[index(this.i,this.j+1,dif)];
      let left = grid[index(this.i-1,this.j,dif)];

      if(!this.paredes[0] && top && !top.visited)
          neighbors.push(top) //Ingresa el de arriba
      if(!this.paredes[1] && rigth && !rigth.visited)
          neighbors.push(rigth) //Ingresa el derecha
      if(!this.paredes[2] && bottom && !bottom.visited)
          neighbors.push(bottom) //Ingresa el de abajo
      if(!this.paredes[3] && left && !left.visited)
          neighbors.push(left) //Ingresa el de left

      return (neighbors.length > 0) ? neighbors[Math.floor(Math.random() * neighbors.length)] : undefined;
    }
  }
```

Figura 2: clase Cell.js

Ahora bien, lo anterior es la parte lógica ya que para la generación gráfica se utiliza otro algoritmo que por medio de canvas pintan el laberinto en la página, este lo que hace es revisar para cada celda si “existen” la paredes y las dibuja.

```
let mostrar = (cell,ancho) =>{
  let x = cell.i * ancho;
  let y = cell.j * ancho;
  let ctx = getCanvasContext('canvas');
  if(cell.paredes[0]) //Top
    line(ctx,x,y,x+ancho,y);
  if(cell.paredes[1]) //Rigth
    line(ctx,x+ancho,y,x+ancho,y+ancho);
  if(cell.paredes[2]) //Bottom
    line(ctx,x+ancho,y+ancho,x,y+ancho);
  if(cell.paredes[3]) //Left
    line(ctx,x,y+ancho,x,y);
  if(cell.path == 1){
    ctx.rect(x,y,ancho,ancho);
    ctx.fillStyle = '#A6AEBF';
    ctx.fill();
  }
}
```

Figura 3: Método de muestra hacia el canvas

La solución automática utiliza el mismo algoritmo de backtracking y va guardando en una variable path si la celda fue visitada, hasta que esté en la última posición del grid. Luego para dibujarlo se verifica si cada celda del grid tiene la variable path en 1 o verdadera y se pinta con el método mostrar que está anteriormente, a continuación el algoritmo de resolver el laberinto automáticamente.

```
this.solveMaze = (grid,tamano) => {
  grid[0].visited = true;
  let solvebacktracking = (actual, stack) => {
    let revisaVecino = next => {
      if(next){
        next.visited = true;
        stack.push(actual);
        this.makePath(actual,next);
        solvebacktracking(next,stack);
      }
      else if(actual.i == tamano-1 && actual.j == tamano-1) {return grid;}
      else{
        actual.path = 0;
        solvebacktracking(stack.pop(), stack);
      }
    }
    revisaVecino(actual.checkWalls(grid , [], tamano));
  }
  solvebacktracking(grid[0], []);
  grid[0].path = 1;
  return grid;
}
this.makePath = (current,next) => {current.path = 1; next.path = 1; }
}
```

Figura 4: Método de solución del laberinto

Por último para la solución manual del laberinto se utilizó un método que responde a los eventos generados por las flechas, y este pinta la celda en la que se encuentra, para ello, se utiliza una clase cursor para saber la posicion X y Y en la que se encuentra el jugador.

5 Conclusiones

La mayor comodidad que sintió el grupo de trabajo fue por poder trabajar con Javascript tanto en el servidor como en el cliente, ya que no hay que hacer conversiones de tipos de objetos como pasaba en tecnologías como servlets de Java.
También, fue muy provechoso la investigación de nuevas tecnologías como Node.js y Express.js, que están con buen auge en el mercado. Además, los fundamentos de programación funcional hacen "ponerse los lentes" y enfocarse en un paradigma al que no se estaba acostumbrado, al principio genera conflictos pero estos mismos resuelven en aprendizaje.

6 Referencias

1. NodeJs Foundation. (2016). NodeJS Docs. 11/8/2016, de Node.js Foundation. Sitio web: https://www.nodejs.org/en/
2. NodeJs Foundation. (2016). Express-Node. 11/8/2016, de Node.js Foundation. Sitio web: http://www.expressjs.com/es/
3. MongoDB Inc.. (2016). MongoDB. 22/8/2016, de MongoDB Sitio web: https://www.mongodb.com/
4. Mozilla Developer Network. (2015-2016). Using Fetch. 15/8/2016, de Mozilla Developer Network Sitio web: https://www.developer.mozilla.org/
5. W3School. (2016). Bootstrap. 29/8/2016, de School Sitio web: http://www.w3schools.com/bootstra