

# Algorítmica avanzada

Grup 2

<b>Explicación del lenguaje de programación escogido</b>	<b>2</b>
<b>Explicación del diseño de los diferentes algoritmos y estrategias</b>	<b>3</b>
Laberinto	3
Backtracking	3
2- Determinar que será solución y su representación	3
3- Determinar el árbol de búsqueda	4
4- Codificar las funciones auxiliares	4
5 - Escribir el algoritmo final	5
Branch and Bound	7
Comprobar que el problema es resoluble con B&B	7
2. La solución será un registro compuesto por:	7
3. Determinar espacio de búsqueda	7
4. Codificar funciones auxiliares	8
Sopa de letras	12
Backtracking	12
Comprobar si el problema es resoluble con Backtracking	12
Determinar qué será solución y su representación	12
Determinar el árbol de búsqueda	12
Codificar las funciones auxiliares	13
Escribir el algoritmo final	16
Greedy	17
Definir el conjunto de candidatos	17
Indicar el criterio de selección de candidatos	17
Determinar qué será la configuración	17
Identificar bajo qué criterios una configuración es válida (completable/factible)	17
Indicar bajo qué criterios se habrá alcanzado la solución	17
Definir el criterio objetivo	18
Codificar el algoritmo Greedy resultante	18
<b>Análisis de resultados</b>	<b>21</b>
<b>Problemas observados</b>	<b>30</b>
<b>Dedicación total en horas</b>	<b>31</b>
Comprensión de las especificaciones	31
Análisis y Diseño	31
Implementación	31
Testing	32
Documentación	32
<b>Conclusiones</b>	<b>33</b>

# 1. Explicación del lenguaje de programación escogido

El lenguaje de programación elegido ha sido Java, por la parte de backend nos permite no preocuparnos de la gestión de la memoria y de los recursos del programa, más que usando variables y objetos (instancias de clases), ya que de esto se encarga el Garbage Collector; cuya función es limpiar las variables y los objetos, no usadas.

Esto es una gran ventaja respecto a otros lenguajes de programación, por ejemplo, el lenguaje C, donde te debes encargar de pedir la memoria, redimensionarla y al final de cada función y/o final del programa, limpiar la memoria y cerrar los recursos usados.

Por la parte de frontend / parte visual, se ha elegido el uso de Java, ya que nos provee de las herramientas Swing y AWT, las cuales permiten renderizar a través del código Java.

Al mismo tiempo, incorpora más integración y compatibilidad con la parte de backend, al construir y ejecutar el programa, al ser la parte de backend y de frontend con el mismo lenguaje de programación.

El emulador renderiza los resultados para ir viendo como va evolucionando el código en “tiempo real” siendo muy útil para ver el comportamiento del algoritmo, pero también para hacer el “debug” ya que al ver paso a paso como va implementando cada decisión hemos podido detectar en qué parte del código podíamos tener un error.

En cuanto a los algoritmos seleccionados para realizar el laberinto han sido:  
Backtracking y Branch & Bound.

Hemos elegido Backtracking porqué la forma de gestionar los caminos creemos que era muy adecuada y se podían realizar mejoras/poda que hacían que el algoritmo fuera muy eficiente.

Como segundo algoritmo para el laberinto, hemos elegido Branch & Bound para poder utilizar Branch & Bound el ejercicio tenía que ser resoluble con Backtracking pero también debía ser de minimización y que se pueda aplicar la poda basada en la mejor solución en curso y en este ejercicio se podía realizar sin problemas ya que se puede podar en cuanto la distancia recorrida es mayor a la de una solución anterior pero aún no se ha llegado a la solución.

## 2. Explicación del diseño de los diferentes algoritmos y estrategias

### Laberinto

#### Backtracking

1- ¿Comprobar si se puede resolver este problema utilizando backtracking?

- ¿Podemos representar la solución con una tupla?
  - o Si, podemos usar un array con tantas posiciones como casillas tiene la matriz ( $m*n$ ) en la que la casilla  $i$ -ésima del array guardará las coordenadas de la posición que ocupa el cursor en el nivel “ $k$ ” hasta llegar a la casilla “ $S$ ”
- ¿La solución se puede construir de forma incremental?
  - o Si, se decide de uno en uno cuál será el movimiento (coordenadas de la matriz) del cursor hasta llegar a la casilla “ $S$ ”
- ¿Existen las funciones auxiliares?
  - o Si,  
Solución: será solución cuando haya alcanzado la casilla del array que contiene una “ $s$ ”.

Se pueden unir las funciones completable y factible

Buena: comprueba si la solución o no solución ¡cumple todas las restricciones del problema que son:

- No puede pasar por el mismo sitio más de una vez
- No puede pasar por los muros
- Solo podemos movernos en las cuatro direcciones cardinales

2- Determinar que será solución y su representación

Tipos

Configuración = array [1..MAX\_MATRIZ\_CASILLAS] de caracteres

Fintipos

1	2	3	4	5	6	7
R	R	D	D	L	L	U

### 3- Determinar el árbol de búsqueda

- La anchura es igual a 4 (MAX\_TIPO\_MOV) que son todas las posibles elecciones que podemos realizar.
  - La altura es igual a MAX\_MATRIZ\_CASILLAS (M\*N) es la longitud máxima que puede tener el array.
  - En cada nivel “k” nos preguntamos qué movimiento realizamos y la distancia que llevamos recorrida.
- 
- Coste:  $O(4^{(M \cdot N)} + 1)$

### 4- Codificar las funciones auxiliares

prepararRecorridoNivel (x,k) -> x[k]:= 0

haySucesor(x,k) -> x[k] <= numDecisiones

siguienteHermano(x,k) -> x[k] := x[k]+1

solución (x,k) -> m[x\_actual][y\_actual]= EXIT

```
funcion buena (x:configuracion; k:entero; numDecisiones: entero)
devuelve booleano

var
    i,j,x,y,x_calculada,y_calculada: entero
    correcto: booleano

finvar

x_calculada:=1
y_calculada:=1
x:=1
y:=1

para i:=1 hasta k + 1
    opcion
        caso UP
            y--
        caso DOWN
            y++
        caso LEFT
            x--
        caso RIGHT
            x++
    finopcion
finpara
si matriuCells[y][x] := WALL hacer
    devuelve falso
```

```

finsi

para j:=1 hasta k
    opcion
        caso UP
            y_calculada-
        caso DOWN
            y_calculada++
        caso LEFT
            x_calculada--
        caso RIGHT
            x_calculada++
finopcion

si x:= x_calculada ^ y == y_calculada
    devuelve falso
finsi

finpara
devuelve cierto
finfuncion

proc tratarsolucion(x:configuracion; k:entero)
var
    i
finvar

i:=1
si k < vMejor v vMejor:= 0 hacer
    vmejor := k
    xMejor := x
finsi
soluciones[i] := x
i++

finproc

```

## 5 - Escribir el algoritmo final

```

proc Backtracking(x: configuracion; k: entero)
var
    numDecisiones: entero
fivar
numDecisiones:=1
x:=prepararRecorridoNivel(x,k)
mientras haySucesor(configuracion,k,numDecisiones) hacer
    x:=siguienteHermano(configuracio,k,numDecisiones)
    opcion
        caso EXIT: opcion
            caso buena(x,k,numDecisiones):
                tratarSolucion(x,k)
            finopcion
        caso k < N+1: opcion

```

```

    caso buena(x, k, numDecisiones) :
        Backtracking(x, k+1)
    finopcion
    finopcion
    numDecisiones:=numDecisiones+1
finmientras
finproc

```

## 6. Mejoras en la eficiencia

Reducir el espacio explorado del árbol de búsqueda con poda basada en la mejor solución en curso (PBMSC)

```

proc BacktrackingWithOpt(x: configuracion; k: entero)
    var
        numDecisiones: entero
    fivar
        numDecisiones:=1
        x:=prepararRecorridoNivel(x, k)
    mientras haySucesor(configuracion, k, numDecisiones) hacer
        x:=siguienteHermano(configuracion, k, numDecisiones)
    opcion
        caso EXIT: opcion
            caso buena(x, k, numDecisiones) :
                tratarSolucion(x, k)
            finopcion
        caso k < N+1: opcion
            caso buena(x, k, numDecisiones) :
                si k < vMejor || vMejor == 0 hacer
                    Backtracking(x, k+1)
                finsi
            finopcion
        finopcion
        numDecisiones:=numDecisiones+1
    finmientras
finproc

```

## Branch and Bound

1. Comprobar que el problema es resoluble con B&B

El problema cumple los requisitos para resolverse con Backtracking y es compatible con la aplicación de la PBMSC.

Es un problema en el que hay que minimizar el recorrido a realizar y presenta un coste aditivo por niveles en cuanto a la distancia recorrida.

Por lo que puede ser resuelto con Branch & bound

2. La solución será un registro compuesto por:

- Un array de MAX\_MATRIZ\_CASILLAS (cantidad de casillas que forman el laberinto, altura por anchura). Las casillas almacenarán la dirección a la que se dirige el cursor por cada decisión tomada.
- Un entero “k” que nos indicará el nivel en el que nos encontramos.

Representación:

```
tipo
    configuración = registro
        d: array [1..MAX_MATRIZ_CASILLAS] de dirección
        k: entero
    finregistro
fintipo
```

3. Determinar espacio de búsqueda

- La anchura es igual a 4 (MAX\_TIPO\_MOV) que son todas las posibles elecciones que podemos realizar.
  - La altura es igual a MAX\_MATRIZ\_CASILLAS (M\*N) es la longitud máxima que puede tener el array.
  - En cada nivel “k” nos preguntamos qué movimiento realizamos y la distancia que llevamos recorrida.
- 
- Coste:  $O(4^{(M*N)} + 1)$

#### 4. Codificar funciones auxiliares

```
Funcion configuracionRaiz() devuelve configuración

var

    x:configuración

finvar

    x.k :=0

devuelve x

finfuncion

funcion expande (x: configuracion) devuelve <array[1..N] de
configuración, entero>

var

    hijos: array [1..numDecisiones] de configuración

    i,j: entero

finvar

para i:=1 hasta numdecisiones

    para j:=1 hasta x.k

        hijos[i].d[j] := x.d[j]

    +finpara

    hijos[i].d[x.k+1] := i

    hijos[i].k := x.k + 1

finpara

devuelve <hijos, numdecisiones>

finfuncion

funcion solución (x: configuracion) devuelve booleano
```

```

devuelve  x.[x.k]= EXIT

finfuncion

funcion buena (x:configuracion; k:entero; numDecisiones: entero)
devuelve booleano

var
    i,j,x,y,x_calculada,y_calculada: entero
    correcto: booleano

finvar

x_calculada:=1
y_calculada:=1
x:=1
y:=1

para i:=1 hasta k
    opcion
        caso UP
        y--
        caso DOWN
        y++
        caso LEFT
        x--
        caso RIGHT
        x++
    finopcion
finpara
si matriuCells[y][x] := WALL hacer
    devuelve falso
finsi

para j:=1 hasta k-1
    opcion
        caso UP
            y_calculada-
        caso DOWN
            y_calculada++
        caso LEFT
            x_calculada--
        caso RIGHT
            x_calculada++
    finopcion

    si x:= x_calculada ^ y == y_calculada
        devuelve falso
    finsi

finpara
devuelve cierto
finfuncion

```

```

Función valorParcial (x:configuracion) devuelve entero

devuelve valor(x)

finfuncion

```

```

función valorEstimado (x:configuracion) devuelve entero

devuelve valor(x)

finfuncion

```

```

funcion B&B ()devuelve configuracion

```

```

var

```

```

    nodosVivos: colaPrioridad

```

```

    x, xMejor: configuracion

```

```

    numHijos, vMejor, i: entero

```

```

    hijos: array[1..MAX_MATRIU_CASELLES] de
configuracion

```

```

finvar

```

```

nodosVivos := COLA_PRIORIDAD.crea()

```

```

x:=configuracionRaiz()

```

```

COLA_PRIORIDAD.encola(nodosVivos,x,+infinito)

```

```

vMejor := +infinito

```

```

mientras ¬COLA_PRIORIDAD.vacia(nodosVivos) hacer

```

```

    x:= COLA_PRIORIDAD.primer(nodosVivos)

```

```

    COLA_PRIORIDAD.desencola(nodosVivos)

```

```

    <hijos, numHijos> := expande(x)

```

```

para i=1 hasta numHijos

opcion

caso solucion(hijos[i])

    si buena(hijos[i]) entonces

        si valor(hijos[i])< vMillor
            vMejor:= valor(hijos[i])

            xMejor:= hijos[i]

        finsi

    finsi

    caso ¬solucion(hijos[i])

        si buena(hijos[i]) entonces

            si valorParcial(hijos[i]<vMejor)

                COLA_PRIORIDAD.encola(nodosVi
                vos,hijos[i],valorEstimado[hijos[i
                ]])

            finsi

        finsi

    finopcion

finpara

finmientras

COLA_PRIORIDAD.destruye(nodosVivos)

devuelve xMejor

finfuncion

```

# Sopa de letras

## Backtracking

### 1. Comprobar si el problema es resoluble con Backtracking

- ¿Podemos representar la solución con una tupla?

Sí, podemos usar un array con n posiciones para indicar las coordenadas de las letras..

- ¿La solución se puede construir de forma incremental?

Sí, se decide si la letra actual coincide con las de la palabra en el mismo orden consecutivo..

- ¿Existen las funciones auxiliares?

Sí.

solución(): comprobar si se ha decidido poner cada uno de los n caracteres de la palabra..

factible() y completable(): pueden agruparse en una misma función buena(), quedan letras por revisar o no, siempre que encaje una letra de la matriz se guarda su posición, hay que validar que sean consecutivas.

### 2. Determinar qué será solución y su representación

#### tipos

```
configuracion = array [1..CASELLAS_PALABRA] de Point
fintipos
```

### 3. Determinar el árbol de búsqueda

- La anchura es igual a filas por columnas (MAX\_FILAS\*MAX\_COLUMNAS).
  - La altura es igual a la longitud de la palabra.
- 
- Coste:  $O(\text{MAX\_FILAS} * \text{MAX\_COLUMNAS}^{(\text{LONGITUD\_PARAULA})+1})$

#### 4. Codificar las funciones auxiliares

```
prepararRecorridoNivel(x,k) -> x[k].x=-1; x[k].y=-1  
haySucesor(x,k) -> x < LONGITUD_MATRIU ^ y < LONGITUD_MATRIU  
siguienteHermano(x,k,coordx,coordy) -> x[k].setloaction(coordx,coordy)
```

```
solucion(x,k) -> !(k < LONGITUD_PALABRA)
```

```
funcion buena(x: configuracio; k entero) devuelve booleano
```

```
var
```

```
    encontrada: BOOL
```

```
    j : entero
```

```
finvar
```

```
encontrada:=TRUE
```

```
j:=0
```

```
si (matriuCells[configuracionFinal[k].getY()][  
configuracionFinal[k].getX()] != referent.charAt(k)) hacer  
    devolver FALSO  
finsi
```

```
si (k == 1) hacer
```

```
    si (configuracionFinal[k - 1].getX() + 1 ==
```

```
        configuracionFinal[k].getX() ^ configuracionFinal[k -  
1].getY() + 1 == configuracionFinal[k].getY()) hacer
```

```
        devuelve TRUE
```

```
finsi
```

```
    si (configuracionFinal[k - 1].getX() + 1 ==
```

```
        configuracionFinal[k].getX() ^ configuracionFinal[k -  
1].getY() == configuracionFinal[k].getY()) hacer
```

```
        devuelve TRUE
```

```
finsi
```

```
    si (configuracionFinal[k - 1].getX() ==
```

```
        configuracionFinal[k].getX() ^ configuracionFinal[k -
```

```
1].getY() + 1 == configuracionFinal[k].getY()) hacer
```

```

devuelve TRUE

finsi

devuelve FALSO

finsi

si (k > 1) hacer

    si (configuracionFinal[0].getX() + 1 ==

        configuracionFinal[1].getX() ^ configuracionFinal[0].getY()

        + 1 == configuracionFinal[1].getY()) hacer

        j := 2

        sino (configuracionFinal[0].getX() + 1 ==

            configuracionFinal[1].getX() ^ configuracionFinal[0].getY()

            == configuracionFinal[1].getY()) hacer

            j := 0

            sino (configuracionFinal[0].getX() ==

                configuracionFinal[1].getX() ^ configuracionFinal[0].getY()

                + 1 == configuracionFinal[1].getY()) hacer

                j := 1

                finsi

                para i:=1,i<k,i++ hacer

                    opcion

                    caso 0 : // Derecha

                        si (configuracionFinal[i].getX() + 1 ==

                            configuracionFinal[i + 1].getX() ^

                            configuracionFinal[i].getY() ==

                            configuracionFinal[i + 1].getY()) hacer

                            encontrada := TRUE

                        sino

                            devuelve FALSO

```

```

    finsi

    caso 1: // Abajo

        si (configuracionFinal[i].getX() ==
            configuracionFinal[i + 1].getX() ^
            configuracionFinal[i].getY() + 1 ==
            configuracionFinal[i + 1].getY()) hacer

            encontrada := TRUE

        sino

            devuelve FALSO

        finsi

    caso 2: // Diagonal

        si (configuracionFinal[i].getX() + 1 ==
            configuracionFinal[i + 1].getX() ^
            configuracionFinal[i].getY() + 1 ==
            configuracionFinal[i + 1].getY()) hacer

            encontrada := TRUE

        sino

            devuelve FALSO

        finsi

    finopcion

    finpara

    devuelve TRUE

finfuncion

```

```

funcion tratarSolucion(x : configuracion,k: entero)
    si !encontrado hacer
        encontrado:=CIERTO
        configuracion[0]:= xl[0].gety()
        configuracion[1]:= xl[1].getx()
        configuracion[2]:=xl[LONGITUD_PALABARA-1].gety()
        configuracion[3]:=xl[LONGITUD_PALABARA-1].getx()

```

finsi  
finfuncion

5. Escribir el algoritmo final

```
funcion (configuracionFinal: Point[],k: entero, referent: String )  
    var  
        x: entero  
        y: entero  
    finvar  
  
    x:=1  
    y:=1  
  
    configuracionFinal:=prepararRecorridoNivel(configuracionFinal,k)  
    mientras haySucesor(configuracionFinal,k,x,y) hacer  
        configuracionFinal := siguienteHermano(configuracionFinal,k,x,y)  
        si solucion(configuracionFinal,k) hacer  
            si buena(configuracionFinal,k, referent) hacer  
                tratarSolucion(configuracionFinal,k)  
            finsi  
        sino  
            si buena(configuracionFinal,k, referent) hacer  
                backtracking(configuracionFinal,k+1,referent)  
            finsi  
        finsi  
        si x<MATRIZ_CASELLES hacer  
            x:=x+1  
        sino  
            x:=0  
            y++  
        finsi  
    finmientras  
finfuncion
```

## Greedy

### 1. Definir el conjunto de candidatos

El conjunto de candidatos estará formado por el array de las posiciones x,y de la primera letra y de la última letra.

### 2. Indicar el criterio de selección de candidatos

Encontrar de letra en letra hasta salir la palabra referida, de esta manera primero al encontrar la primera letra, guardamos coordenadas, seguimos analizando de lectura normal, hasta salir la última letra, habrá que guardar la posición de la esta última letra.'

### 3. Determinar qué será la configuración

Podemos representar la solución mediante un array de números enteros. En cada casilla se almacenará la x e y siendo las 2 primeras casillas, las coordenadas de la primera y las 2 siguientes de la última letra'.

tipos

```
configuracion = array [1..4] de entero  
fintipos
```

### 4. Identificar bajo qué criterios una configuración es válida (completable/factible)

La selección de una configuración será factible si de manera natural, se entiende de izquierda a derecha, en diagonal o hacia abajo, las letras formadas por la palabra referida están consecutivas.

### 5. Indicar bajo qué criterios se habrá alcanzado la solución

Se alcanzará la solución cuando se haya considerado todas las letras de la palabra referida<

### 6. Definir el criterio objetivo

El valor de la solución obtenida es la suma de las casillas con valor CIERTO del array de configuración.

### 7. Codificar el algoritmo Greedy resultante

```
funcion Greedy(matriuCells: array de [1..MAX_CASILLAS] de  
array[1..MAX_CASILLAS]) devuelve configuración  
var  
    s: configuración  
    i: entero  
    x: entero  
    encontrado: booleano  
    y: entero
```

```

finvar
para y=1 hasta celdasMatriz + 1 ^ ~ encontrado
    para x=1 hasta celdasMatriz + 1 ^ ~ encontrado
        si matriuCells[y][x] = palabra[0]
            si y + LONGITUD_PALABRA < celdasMatriz
                si heuristic(x, y, 1)
                    configuracio[0] := y
                    configuracio[1] := x
                    configuracio[2] := y +
LONGITUD_PALABRA
                    configuracio[3] := x
                    encontrado := cierto
                fisi
            fisi
        fisi
        si x + LONGITUD_PALABRA < celdasMatriz
            si heuristic(x, y, 2)
                configuracio[0] := y
                configuracio[1] := x
                configuracio[2] := y
                configuracio[3] := x + LONGITUD_PALABRA
                encontrado := cierto
            fisi
        fisi
    si y + LONGITUD_PALABRA < celdasMatriz ^ x + LONGITUD_PALABRA <
celdasMatriz
        si heuristic(x, y, 3)
            configuracio[0] := y
            configuracio[1] := x
            configuracio[2] := y + LONGITUD_PALABRA
            configuracio[3] := x + LONGITUD_PALABRA
            encontrado := cierto
        fisi
    fisi
finpara
finpara
fifuncion

```

```

funcion heuristic(x, y, direccion: entero) devuelve booleano

opcion(algoritmo)
caso 1
    opcion(direccion)
caso 1

```

```

si matriuCells[y + LONGITUD_PALABRA][x] =
palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    caso 2
si matriuCells[y][x + LONGITUD_PALABRA] =
palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    caso 3
si matriuCells[y + LONGITUD_PALABRA][x + LONGITUD_PALABRA] =
palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    fiopcion
    ficaso
    caso 2
        opcion(direccion)
            caso 1
si matriuCells[y + 1][x] = palabra[1] ^ matriuCells[y +
LONGITUD_PALABRA - 1][x] = palabra[LONGITUD_PALABRA - 1] ^
matriuCells[y + LONGITUD_PALABRA][x] = palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    caso 2
si matriuCells[y][x + 1] = palabra[1] ^ matriuCells[y][x +
LONGITUD_PALABRA - 1] = palabra[LONGITUD_PALABRA - 1] ^
matriuCells[y][x + LONGITUD_PALABRA] = palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    caso 3
si matriuCells[y + 1][x + 1] = palabra[1] ^ matriuCells[y +
LONGITUD_PALABRA - 1][x + LONGITUD_PALABRA - 1] =
palabra[LONGITUD_PALABRA - 1] ^ matriuCells[y +
LONGITUD_PALABRA][x + LONGITUD_PALABRA] =
palabra[LONGITUD_PALABRA]
    devuelve cierto
fisi
    ficaso
    fiopcion
    fiopcion
    ficaso
    caso 3

```

```

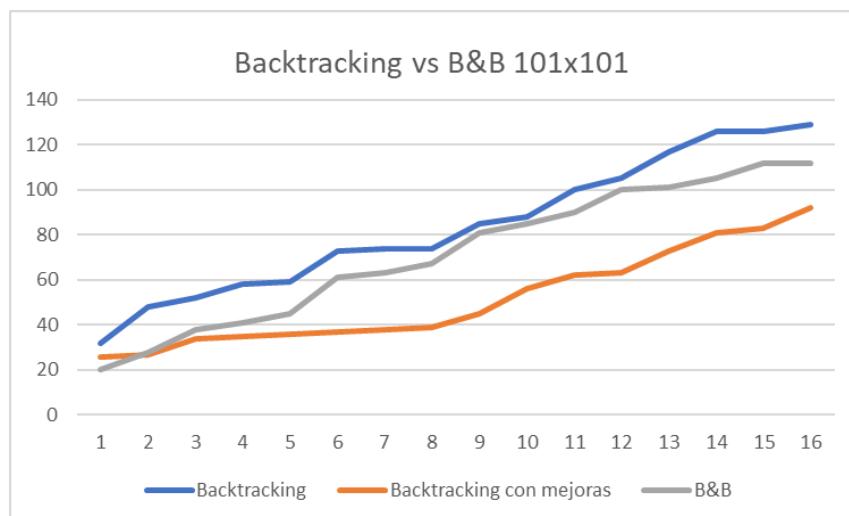
opcion(direccion)
caso 1
    para i=1 hasta LONGITUD_PALABRA + 1
        si matriuCells[ y + i][x] <>
palabra[i]
            devuelve falso
        fisi
        fipara
        devuelve cierto
    ficaso
caso 2
    para j=1 hasta LONGITUD_PALABRA + 1
        si matriuCells[y][x + j] <>
palabra[j]
            devuelve falso
        fisi
        fipara
        devuelve cierto
    ficaso
caso 3
    para i=1,j=1 hasta LONGITUD_PALABRA + 1
        si matriuCells[y + i][x + j] <>
palabra[i]
            devuelve falso
        fisi
        fipara
        devuelve cierto
    ficaso
fiopcion
ficaso
fiopcion
devuelve falso
fifucion

```

### 3. Análisis de resultados

Tamaño	101 (tiempo en ms)		
Laberinto	Laberinto BKT	Laberinto BKT poda	Laberinto B&B
	85	92	101
	52	62	90
	100	83	100
	74	34	41
	126	39	112
	48	35	28
	88	37	67
	105	45	38
	126	63	112
	59	81	81
	73	26	20
	117	36	63
	58	56	45
	74	38	61
	32	73	85
	129	27	105

En esta cuadrícula se muestran los resultados de 16 intentos realizando los 2 tipos de algoritmos y backtracking con PBSC en una tablero de 101x101 en el que se puede comprobar con bastante facilidad cuánto mejora el rendimiento cuando usamos Backtracking con Poda y Branch & Bound respecto a Backtracking tradicional, es bastante obvio ya que reducimos muchísimo la cantidad de pasos que realizamos. También podemos observar que la heurística aplicada en Branch & Bound no hace que mejore las estadísticas comparando con Backtracking con PBSC.



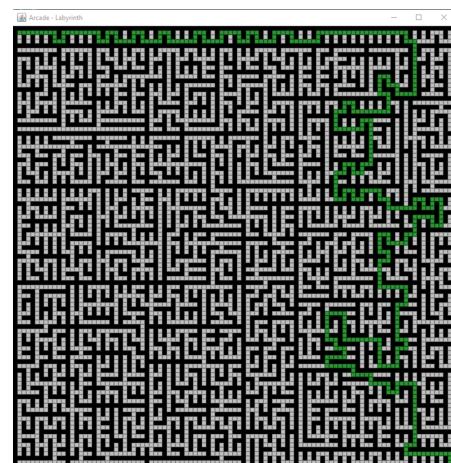
En el gráfico podemos ver los resultados de forma ordenada en el que la vertical son los tiempos en milisegundos y la horizontal son los intentos realizados en 101x101 casillas y se observa mejor que Backtracking con PBSC rinde a un nivel bastante superior a los otros.

A continuación mostramos los caminos que ha realizado cada uno de los algoritmos en su mejor y peor caso

#### Backtracking

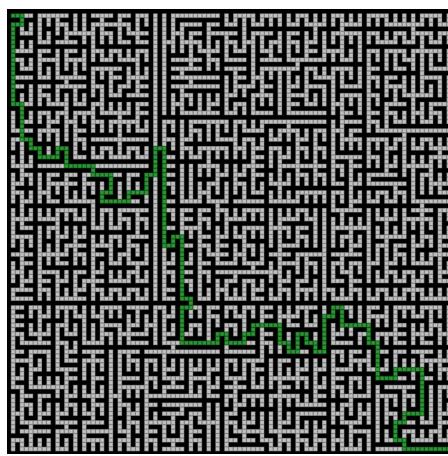


32 ms

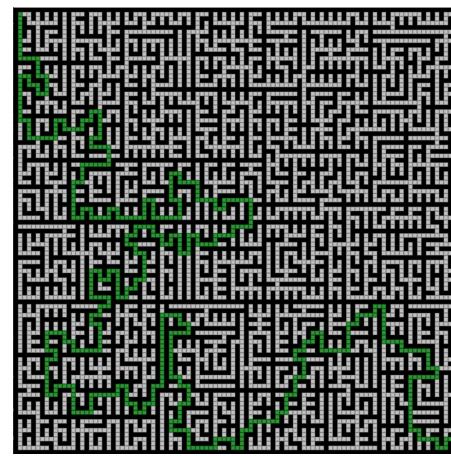


129 ms

#### Backtracking con PBSC

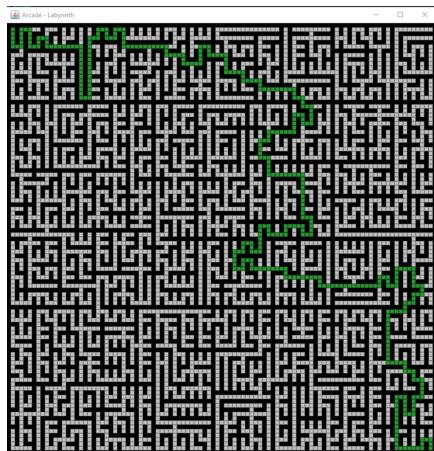


26 ms

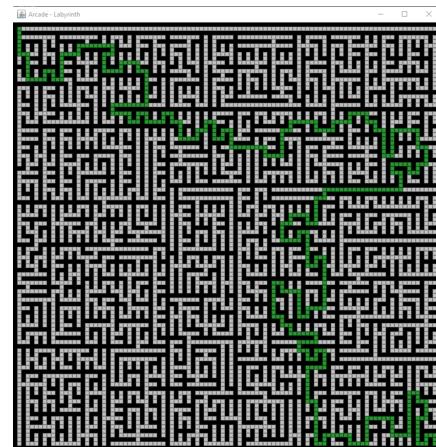


92 ms

## Branch&Bound



20 ms



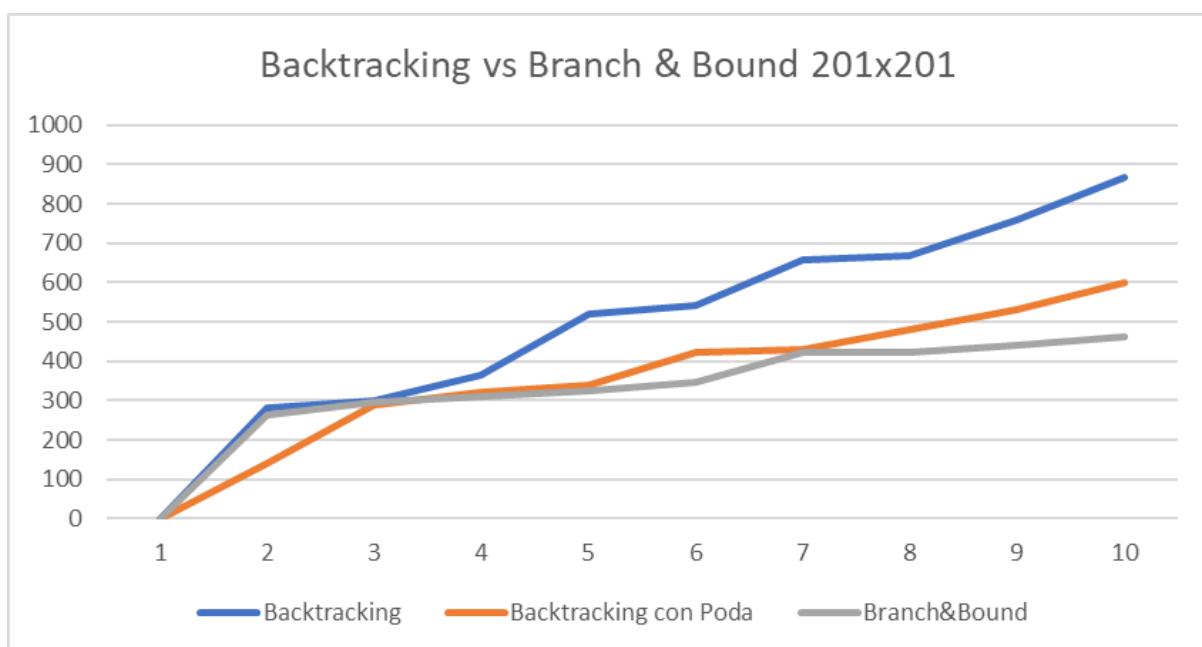
105 ms

De estos tableros las conclusiones que podemos sacar son bastantes de las que destacaremos las siguientes:

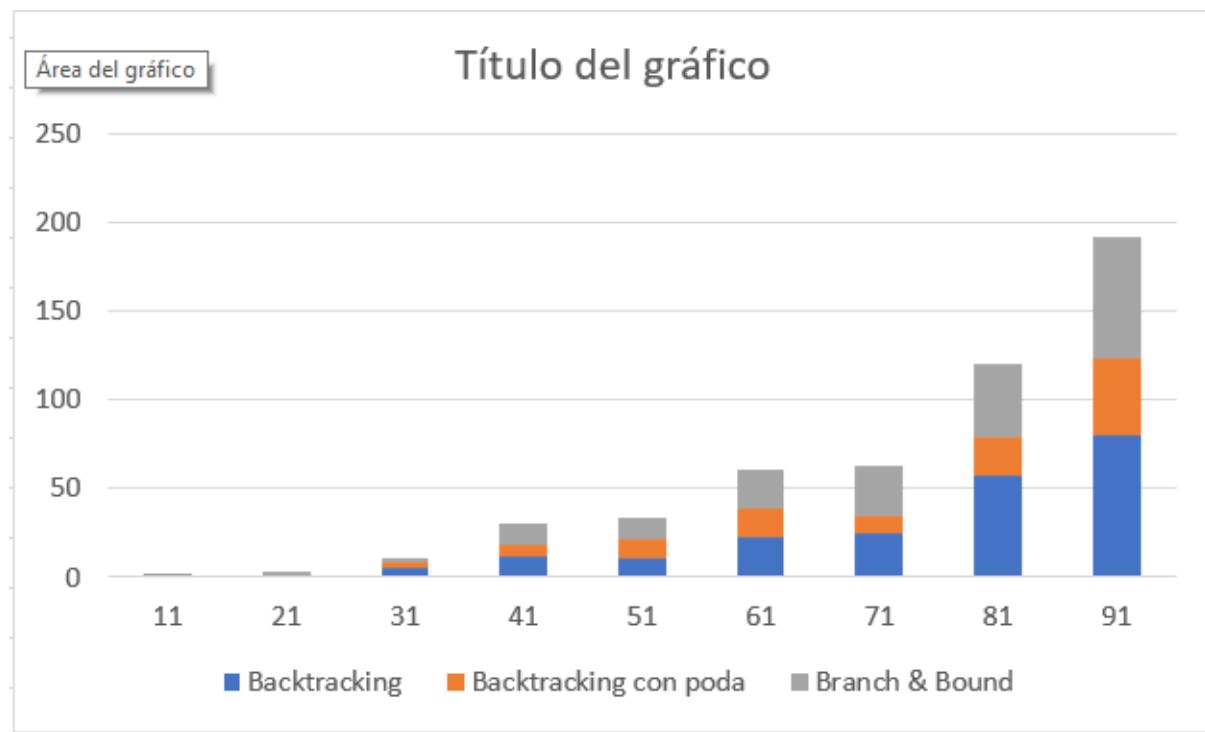
- Es muy importante la dirección que elegimos y el orden en el que las ponemos en nuestro código ya que determinará mucho el resultado obtenido.
- Al ser un tablero de dimensiones tan grandes el tiempo varía mucho entre ejecuciones ya que el camino puede ser muy diverso y puede favorecer o no al algoritmo codificado
- Las 3 ejecuciones más rápidas de los 3 algoritmos tienen una forma muy similar lo que nos da a entender que este tipo de problema combinatorio al tener fija la entrada y la salida el mejor camino suele ser la diagonal.

Tamaño	201x201 (tiempo en ms)		
Laberinto	Laberinto BKT	Laberinto BKT poda	Laberinto B&B
	282	140	262
	299	289	294
	366	322	311
	520	338	323
	540	422	347
	658	430	421
	669	480	422
	759	531	442
	867	601	461
	996	603	478

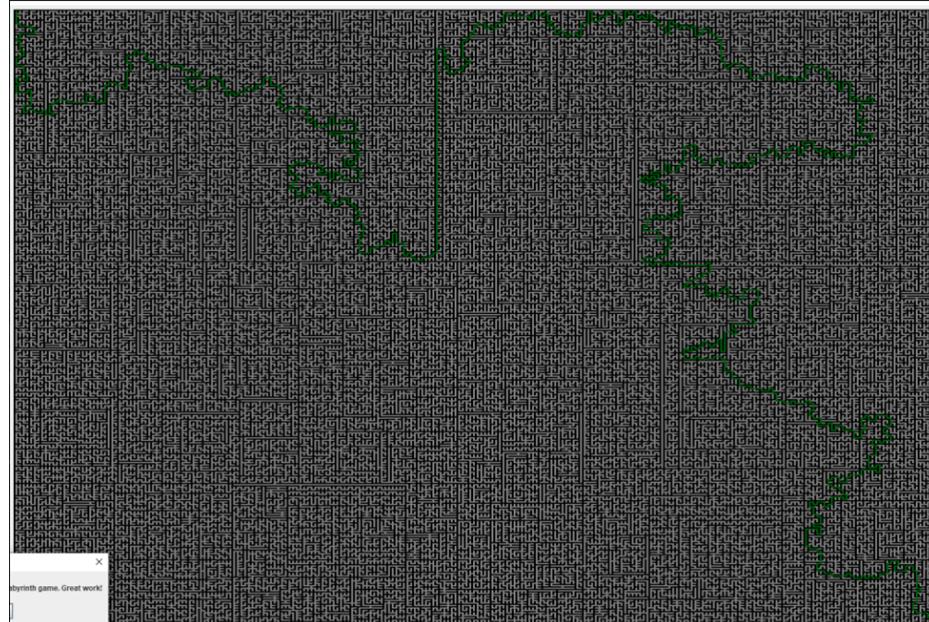
En esta cuadrícula se muestran los resultados de 10 intentos realizando los 2 tipos de algoritmos y backtracking con PBSC en una tablero de 201x201



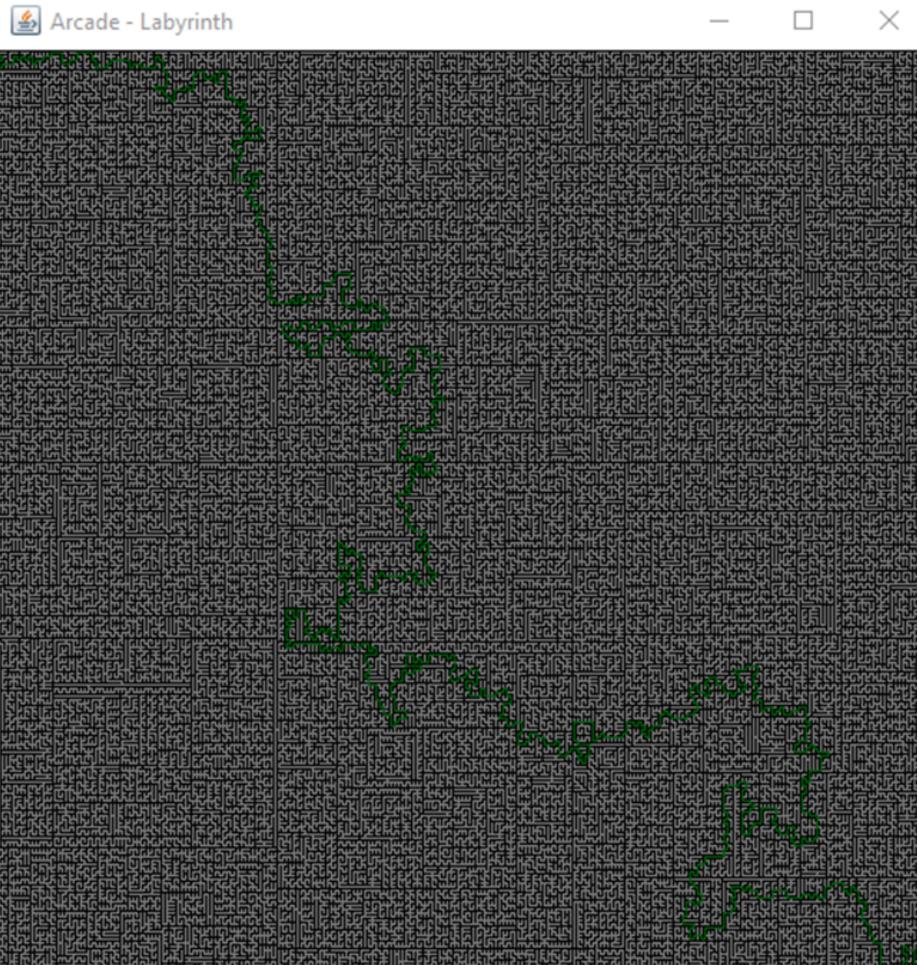
En el gráfico podemos ver los resultados de forma ordenada en el que la vertical son los tiempos en milisegundos y la horizontal son los intentos realizados en 201x201 casillas y se observa mejor que Backtracking con PBSC rinde a un nivel bastante superior a los otros aunque en este caso también podemos comprobar como el peor de los casos en B&B es menor que en los otros 2



Media de 3 ejecuciones de 11 a 91 con los tres algoritmos del laberinto



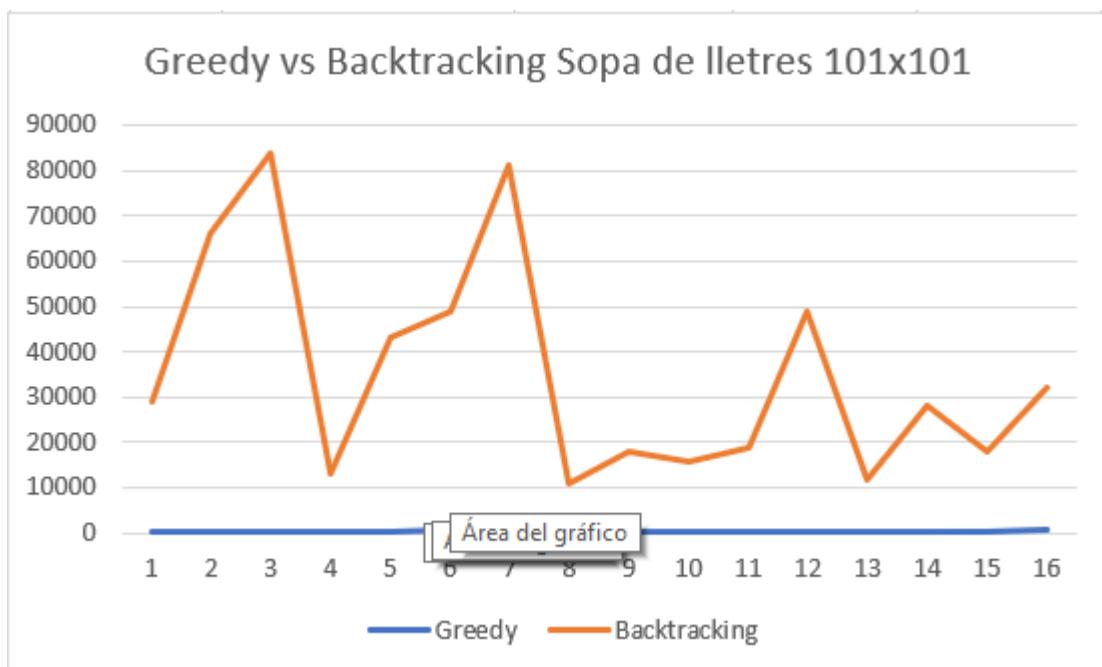
Test realizado Backtracking de 501x501 casillas 23489 microsegundos



Test realizado Branch & Bound de 501x501 casillas 22043 microsegundos

Tamaño	101 (tiempo en microsegundos)	
Sopa de letras	470	29000
	445	66000
	464	84000
	336	13000
	261	43000
	557	49000
	336	81000
	269	11000
	443	18000
	222	16000
	354	19000
	282	49000
	538	12000
	340	28000
	183	18000
	664	32000

En esta cuadrícula se muestran los resultados de 16 intentos realizando los 2 tipos de algoritmos backtracking y greedy en una tablero de 101x101

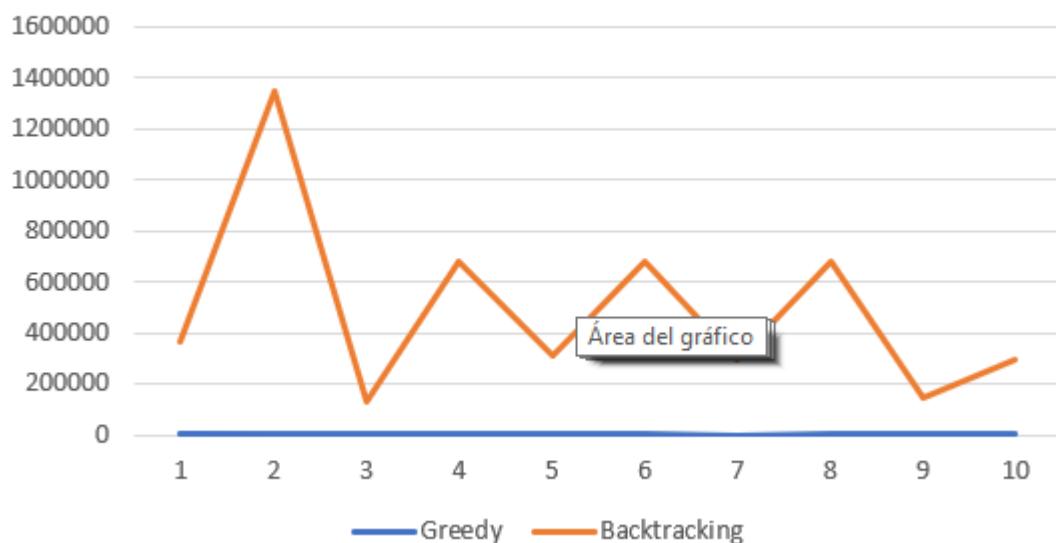


En este grafico observamos que la diferencia entre los algoritmos greedy y backtracking que hemos creado es abismal.

Tamaño	201 (tiempo en microsegundos)	
Sopa de letras	1318	367000
	1046	1349000
	832	129000
	352	680000
	492	309000
	490	683000
	266	295000
	399	680000
	778	145000
	289	295000

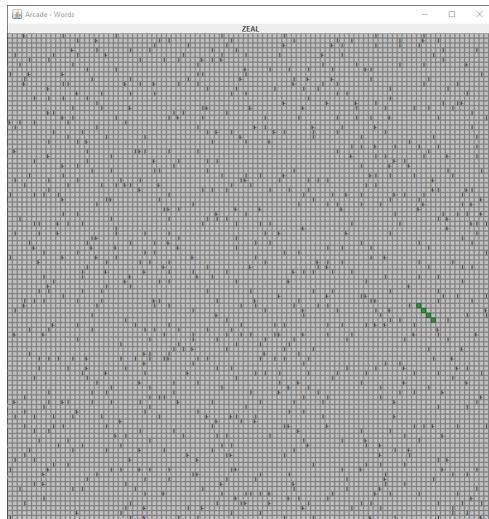
En esta cuadrícula se muestran los resultados de 10 intentos realizando los 2 tipos de algoritmos backtracking y greedy en una tablero de 201x201

Greedy vs Backtracking Sopa de lletres 101x101

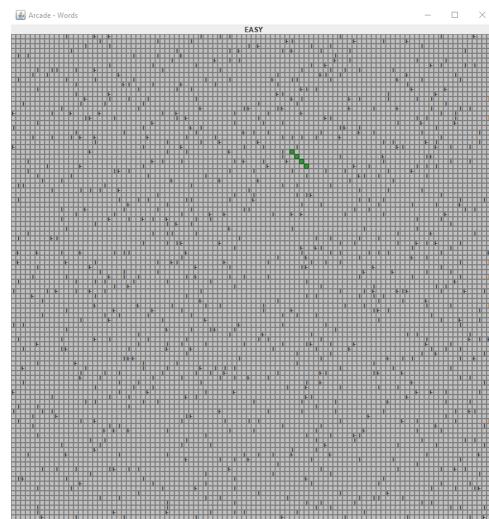


Este gráfico no nos permite ver con exactitud pero la diferencia se va haciendo más grande a medida que el tamaño del tablero crece

### Backtracking 101x101 (Mejor y peor caso de 16 intentos)

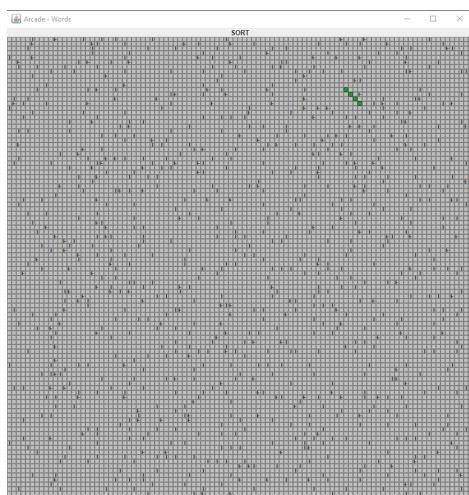


11000 microsegundos

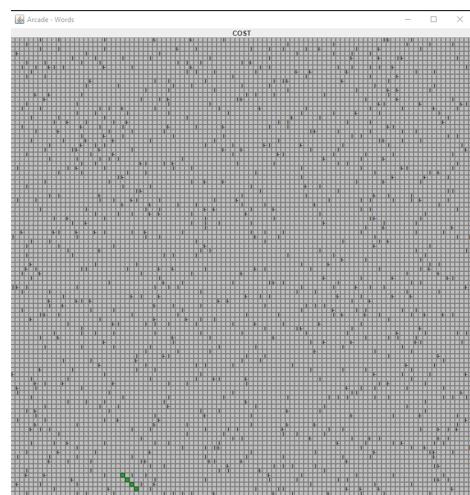


84000 microsegundos

### Greedy 101x101 (Mejor y peor caso de 16 intentos)



183 microsegundos



664 microsegundos

## 4. Problemas observados

Como parte negativa, hemos encontrado dos pequeños problemas con el emulador ya que con un tablero con grandes dimensiones 300 o más, el software tardaba una eternidad en generar dicho tablero con el consiguiente coste de tiempo para realizar las pruebas pertinentes, el otro problema que hemos encontrado ha sido que al generar tableros pares se producía un camino doble que distorsionaba los resultados de manera significativa.

En cuanto a la realización del algoritmo de Backtracking para la sopa de letras, nos hemos encontrado con dificultades a la hora de definir que es altura y que es anchura ya que hemos intentado realizarlo de muchas maneras y la única que hemos encontrado que funciona correctamente es asignar como anchura todas las casillas de la matriz

## 5. Dedicación total en horas

A este proyecto se han dedicado aproximadamente unas 30 horas en total. La mayor parte se ha dedicado a la codificación y el análisis y diseño del proyecto.

### Comprensión de las especificaciones

Tiempo dedicado a comprender el objetivo y la escala del proyecto. Al principio del proyecto, para tener una idea general sobre lo que se tenía que alcanzar y cuál era el mejor camino para conseguir el objetivo.

### Análisis y Diseño

Al principio del proyecto, gracias a tener una idea general del objetivo a alcanzar debido a la comprensión de las especificaciones, hemos dedicado un tiempo para poder planificar los pasos a seguir en los siguientes pasos del proyecto con unos bocetos de los algoritmos para la estructura general y poder empezar a programar.

### Implementación

Este apartado es el que más tiempo ha tomado para nosotros. En este apartado se ha escrito todo el código necesario utilizando los conocimientos adquiridos en la asignatura para poder ejecutar el proyecto una vez se han juntado todos los módulos a medida que se iban finalizando e incorporando a la rama.

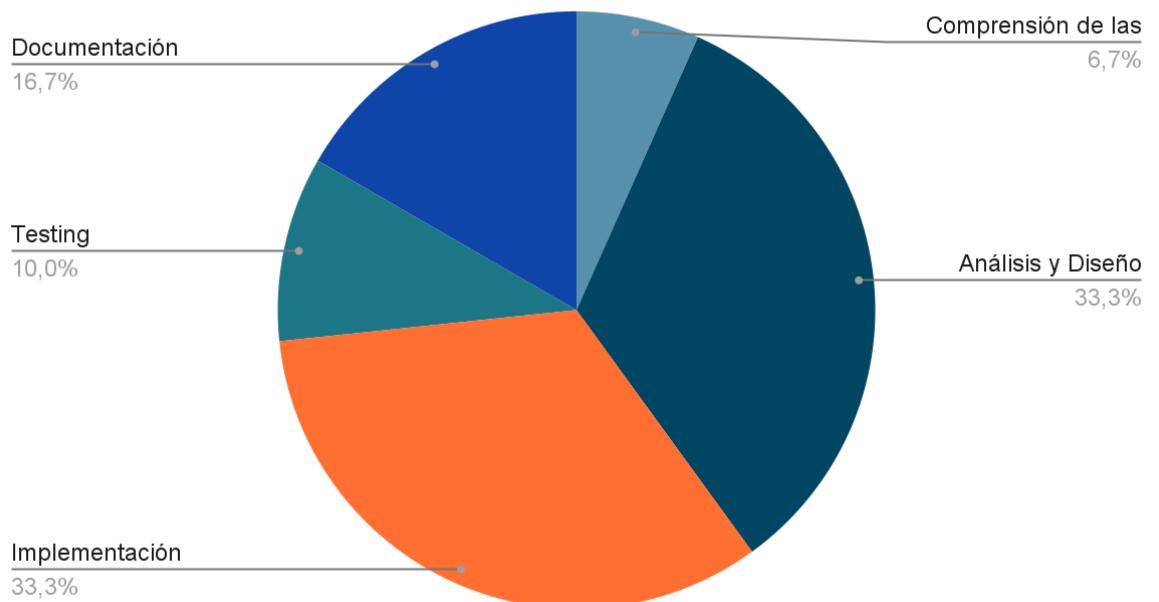
### Testing

En este apartado se incluye el tiempo empleado para asegurar que el proyecto funciona correctamente y arreglar los apartados que no han funcionado correctamente en un primer momento para que funcionaran con una calidad adecuada. Testear que la lógica cumple el objetivo y que la vista muestre los pasos seguidos hasta el resultado.

## Documentación

En este apartado se incluye la creación de la memoria junto con el análisis de resultado, los cuales se han ido realizando desde el comienzo del proyecto y se han actualizando a medida que el proyecto ha ido evolucionando.

Points scored



## 6. Conclusiones

Las conclusiones extraídas del proyecto son muchas, pero a continuación nos gustaría citar las que consideramos más importantes y las que nos han aportado un mayor valor. En primer lugar tenemos la metodología de trabajo de los algoritmos. Después de terminar el proyecto estamos muy contentos, porque los algoritmos no resultan fáciles de plantear al escenario y te obligan a mantener una constancia en el trabajo, tener más a mano la evolución del proyecto cada dos semanas y poder diversificar y organizar mucho mejor el trabajo entre nosotros.

Además, la magnitud del proyecto implica tener que aplicar todos los conceptos aprendidos durante la teoría y práctica de la asignatura. Y sobre todo consolidar la teoría, puesto que en el fondo donde realmente se aprende es practicando y equivocándose.

Por último decir que estamos contentos en global del resultado del proyecto, de cómo se ha organizado el proyecto, de cómo hemos aprendido los conceptos, de saber trabajar en grupo y ayudarnos mutuamente, una buena estructura e implementación del código, además de conseguir unos conocimientos consolidados acerca de la recursividad y sus algoritmos según para qué escenario van mejor o cuales no.