ESCUELA POLITECNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

ALGORITMOS (SIC324-GR2)

GRUPO: No. 11

DEBER: Proyecto Final

Alumnos: Borja Grace, Burbano Renato, Díaz Danny

PROFESORA: Dra. María Gabriela Pérez H.

FECHA DE ENTREGA: 03-02-2019

***Proyecto final**:* Póster e informe

Tema: Resolución de 8-puzzle utilizando búsqueda A*

Objetivo general del proyecto:

Desarrollar las técnicas algorítmicas necesarias para dar solución al problema de encontrar los menores pasos para resolver un puzzle y una vez resuelto, diseñar un póster científico que presente los resultados obtenidos del diseño e implementación de las técnicas algorítmicas que le permitieron encontrar la solución a dicho problema.

Objetivos:

- Crear una interfaz amigable con el usuario que le permita cargar imágenes que luego serán divididas en 9 partes para que se forme un 8-puzzle.
- Potenciar el trabajo en equipo.
- Aplicar técnicas algorítmicas, para dar solución a problemas de la vida real. Desarrollar habilidades y destrezas en programación

Marco teórico:

1. **Busqueda con A***

En inteligencia artificial el tema de búsquedas es central, dado que, por ejemplo, realizar acciones mecanizadas o resolver problemas, se reduce a buscar en un espacio de estados como se explicaba en el apartado anterior. En esa disciplina se estudian búsquedas ciegas (búsqueda primero en amplitud, primero en profundidad, profundidad iterativa, de costo uniforme, etc.) y búsquedas inteligentes (búsqueda avara, A*, IDA*, A* restricta por memoria simplificada, ascenso de cima (hill-climbing), etc.)

Relacionado con la búsqueda del óptimo está el problema del control de la búsqueda, control planteado por Newell y Simon que ha generado una abundancia de trabajos en el campo de la inteligencia artificial. Se trata de elegir entre búsquedas heurísticas lo suficientemente buenas (no perfectas) como para que se pueda dar por concluida la búsqueda con una aceptable respuesta al problema en un lapso aceptable de tiempo.

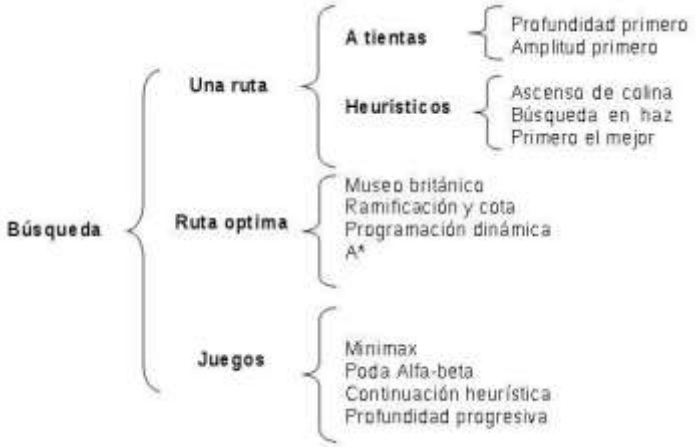
No se discute que las búsquedas aumentan "explosivamente" cuando el espacio de problema se vuelve demasiado vasto por bifurcación de nodos a buscar o por incorporación de más variables. Un control de búsqueda basado en técnicas mediocres también llega a proponer una respuesta adecuada, aunque en un tiempo demasiado largo. En un modelo de mundo o en un contexto con más y más variables que participan y que no se reducen a un número manejable por descarte, surge un problema de control de la búsqueda: ella se vuelve "explosiva". El problema del control de búsqueda (por ejemplo, el problema del operador a elegir, el problema de la planificación, etc.) aún está casi sin resolver.

1.1. El papel de la búsqueda en la Inteligencia Artificial

En Inteligencia Artificial (IA) los términos resolución de problemas y búsqueda se refieren a un núcleo fundamental de técnicas que se utilizan en dominios como la deducción, elaboración de planes de actuación, razonamientos de sentido común, prueba automática de teoremas, etc. Aplicaciones de estas ideas generales aparecen en la práctica totalidad de los sistemas inteligentes, como por ejemplo en los programas que tratan de entender el lenguaje natural, en los programas que tratan de sintetizar un conjunto de reglas de clasificación en un determinado dominio de actuación, o en los sistemas que realizan inferencias a partir de un conjunto de reglas.

1.2. Clasificación

Para elaborar una clasificación de los sistemas de búsqueda se tienen muchas clasificaciones tantas como investigadores y autores en inteligencia artificial existen, en el módulo se ha tratado de organizar esta información para ofrecer un panorama lo más amplio posible para que el estudiante abarque la mayor cantidad de información, los nombres de los algoritmos y métodos de solución en unos casos tienen diferencias que se aclaran en el transcurso del documento. La siguiente clasificación se puede tomar como genérica para tener una idea de las posibilidades de búsqueda.



1.3. Espacios de búsqueda

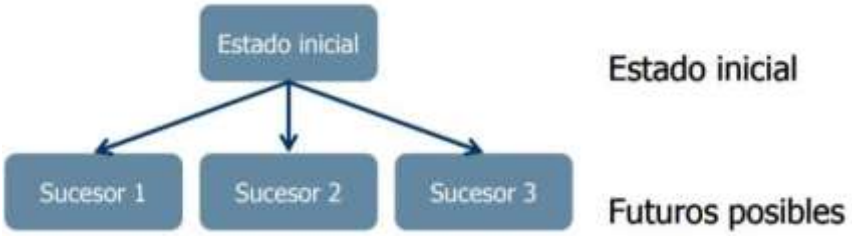
Representación matemática de un problema de búsqueda (nodos: estados; arcos: operadores): Grafo teórico que representa todas las posibles transformaciones del sistema aplicando todos los operadores posibles recursivamente. Debido a su complejidad exponencial, que requeriría una cantidad inviable de memoria y tiempo, el grafo del espacio de estados no puede generarse por completo.

Debido a la complejidad exponencial del grafo implícito, se irá generando, paso a paso, una porción del grafo conforme avance el proceso de búsqueda. El grafo explícito es el subgrafo del grafo implícito del grafo implícito que se va generando durante el proceso de búsqueda de una secuencia de operadores que resuelva nuestro problema (camino solución).

Nodo raíz: Estado inicial.

Hijos de un nodo: Posibles sucesores (nodos correspondientes a estados resultantes de la aplicación de un operador al nodo padre).

Los nodos del árbol representan estados, pero corresponden a PLANES mediante los cuales se alcanzan dichos estados



Ejemplo: 8-puzzle

En el problema del 8 En el problema del 8-puzzle hay que encontrar puzzle hay que encontrar un camino desde la configuración inicial hasta una determinada configuración final.

2. Backtracking

Cuando un problema no tiene un método algorítmico para resolverse, en general la única forma posible de resolverlo es la búsqueda exhaustiva de soluciones entre todas las posibilidades del problema; este método se conoce como fuerza bruta: se generan todos los casos posibles y se testean uno a uno hasta encontrar las soluciones necesarias (a veces basta con encontrar una, en otras ocasiones hay que encontrar todas ellas, o quedarse con la mejor). Sin embargo, en

muchos de estos problemas no es necesario crear completamente un caso para ver si es una solución o no.

Cuando resolver un problema puede hacerse por etapas se puede comprobar paso a paso si se está creando una solución o si se han tomado decisiones que no conseguirán resolver el problema: en cada etapa se estudian las propiedades cuya validez ha de examinarse con objeto de seleccionar las adecuadas para proseguir con el siguiente paso.

En su forma básica Backtracking se asemeja a un recorrido en profundidad del árbol de expansión; se recorre en preorden: primero se evalúa el nodo raíz o actual, y después todos sus hijos de izquierda a derecha. Por tanto, hasta que no se termina de generar una solución parcial (sea válida o no) no se evalúa otra solución distinta. En los nodos del nivel k del árbol se encuentran las soluciones parciales formadas por k etapas o decisiones. Hay que recordar que el árbol no está almacenado realmente en memoria, solo se recorre a la vez que se genera, por lo que todo lo que quiera conservarse (decisiones tomadas, soluciones ya encontradas al problema, etc.) debe ser guardado en alguna estructura adicional. Finalmente, debido a que la cantidad de decisiones a tomar y evaluar puede ser muy elevada, si es posible es conveniente tener una función que determine si a partir de un nodo se puede llegar a una solución completa, de manera que utilizando esta función se puede evitar el recorrido de algunos nodos y por tanto reducir el tiempo de ejecución.

3. Algoritmos voraces

El propósito de un algoritmo voraz es encontrar una solución, es decir, una asociación de valores a todas las variables tal que el valor de la función objetivo sea óptimo. Para ello sigue un proceso secuencial en el que a cada paso toma una decisión (decide qué valor del dominio le ha de asignar a la variable actual) aplicando siempre el mismo criterio (función de selección). La decisión es localmente óptima, es decir, ningún otro valor de los disponibles para esa variable lograría que la función objetivo tuviera un valor mejor, y luego comprueba si la puede incorporar a la secuencia de decisiones que ha tomado hasta el momento, es decir, comprueba que la nueva decisión junto con todas las tomadas anteriormente no violan las restricciones y así consigue una nueva secuencia de decisiones factible.

En el siguiente paso el algoritmo voraz se encuentra con un problema idéntico, pero estrictamente menor, al que tenía en el paso anterior y vuelve a aplicar la misma función de selección para tomar la siguiente decisión. Esta es, por tanto, una técnica descendente. Pero nunca se vuelve a reconsiderar ninguna de las decisiones tomadas. Una vez que a una variable se le ha asignado un valor localmente óptimo y que hace que la secuencia de decisiones sea factible, nunca más se va a intentar asignar un nuevo valor a esa misma variable.

Un algoritmo voraz determina el mínimo número de monedas que debe devolverse en el cambio. En la figura se muestran los pasos que un ser humano debería seguir para emular a un algoritmo voraz para acumular 36 céntimos usando sólo monedas de valores nominales de 1, 5, 10 y 20. La moneda del mayor valor menor que el resto debido es el óptimo local en cada paso. Nótese que en general el problema de devolución del cambio requiere programación dinámica o

programación lineal para encontrar una solución óptima. Sin embargo, en muchos sistemas monetarios, incluyendo el euro y el dolar estadounidense, son casos especiales donde en la estrategia del algoritmo voraz da con la solución óptima.

Desarrollo de la práctica:

1. Primero se definió una función H que cuenta por cada pieza desfasada, cuantos pasos le falta para estar en su posición correcta.

```
public static int H(int[][] matriz_actual) {

    int suma = 0;

    int n = matriz_actual.length, m = matriz_actual[0].length;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            int valor = matriz_actual[i][j];
            if (valor == 0) {
                continue;
            } else {
                int f = (int) Math.ceil((double) valor / 3) - 1;
                int c = (valor - 1) % 3;

                //System.out.println("Eje Y:" + Math.abs(i-f) + ", Eje X:" + Math.abs(j-c));
                suma += Math.abs(i - f) + Math.abs(j - c);
            }

        }
    }
```

Utilizamos la lógica de 'posición final en x menos posición actual en x' y lo mismo para el eje vertical, de esa forma se generó una fórmula matemática que encuentra los valores correctos de forma rápida y los suma iterativamente para obtener un peso final.

2. Creamos funciones de apoyo que nos ayudarán al momento de generar un árbol con backtracking:

```
public static int[] obtenerPosicionVacio(int[][] matriz_actual) {...18 lines }

public static int valorMinimoVector(int[] vector) {...9 lines }

public static void imprimeMatriz(String extra, int[][] matriz) {...12 lines }

public static void copiarMatriz(int[][] copia, int[][] original) {...10 lines }

public static boolean sePuedeMover(int direccion, int direccion_anterior, int i, int j) {...14 lines }

public static int obtenerFallos(int[][] m1) {...13 lines }
```

- obtenerPosicionVacio .- busca en la matriz un cero, y retorna la posición en filas y columnas.
- valorMinimoVector .- obtiene el menor número en un arreglo.
- imprimeMatriz .- por cuestiones de depuración imprime la matriz, asi podemos analizar que sucede.
- copiarMatriz .- copia una matriz a otra, evitando el paso por referencia innecesario.

- sePuedeMover .- verifica si el cero puede moverse en una dirección, no puede moverse por ejemplo hacia arriba cuando su posición es (0,0) ya que la matriz no tiene una posición (-1,0). Eso causaría error.
- obtenerFallos .- cuenta en la matriz los números que no están en su posición correcta.

3. Creamos una función que genera un árbol y verifica cual rama tiene el menor número de pasos.

```
public static void generaArbol(int[][] matriz_actual, ArrayList<String> camino, int direccion_anterior,
        int pos_cero_j, int cota, int nivel) {

    if (obtenerFallos(matriz_actual) == 0) {
        System.out.println("POSIBLE RESPUESTA ENCONTRADA EN EL NIVEL: " + (nivel - 1));
        //imprimeMatriz("FINAL", matriz_actual);
        cota_g = cota;
        cantidad_pasos = camino.size();
        for (String l : camino) {
            //se imprime el camino al revez para que sea humanamente entendible
            if (l.compareToIgnoreCase("0") == 0) {
                System.out.print("↓ ");
            } else if (l.compareToIgnoreCase("1") == 0) {
                System.out.print("← ");
            } else if (l.compareToIgnoreCase("2") == 0) {
                System.out.print("↑ ");
            } else if (l.compareToIgnoreCase("3") == 0) {
                System.out.print("→ ");
            }

        }
        camino_final = new ArrayList<>(camino);
        System.out.println();
        return;
    }else if(nivel >1000){
        JOptionPane.showMessageDialog(null, "El problema no se puede resolver");
    }
```

La primera parte es una condición de salida para la recursividad que en caso que la matriz tenga sus elementos en perfecta posición, guarda los pasos que dio y reescribe la lista 'camino_final' con esos pasos además de establecer una cota. La cota será siempre menor a causa del código que continúa después.

Si el nivel supera los 1000, imprime un mensaje de error, ya que la matriz enviada no se puede resolver.

```
int[] fallos = {100, 100, 100, 100};
int[][][] matrices = {new int[matriz_actual.length][matriz_actual[0].length], new int[matriz_actual.

if (sePuedeMover(0, direccion_anterior, pos_cero_i, pos_cero_j)) {
    copiarMatriz(matrices[0], matriz_actual);
    int aux = matrices[0][pos_cero_i - 1][pos_cero_j];
    matrices[0][pos_cero_i - 1][pos_cero_j] = 0;
    matrices[0][pos_cero_i][pos_cero_j] = aux;
    fallos[0] = H(matrices[0]);
    //fallos[0] = H(matrices[0]);
    //imprimeMatriz("arriba , fallo: " + fallos[0], matrices[0]);
}
```

Creamos un arreglo que guardará todos los valores/pesos que causaría el movernos en las 4 direcciones si es que se puede mover.

El 0 representa arriba, 1 representa derecha, 2 abajo, 3 izquierda.

Si se puede mover en cualquier dirección entonces crea una matriz con el movimiento ya establecido es decir si muevo para arriba la matriz:
123
405
678

Quedaría:
103
425
678

Es importante que aquí se mueve el espacio vacío.

Lo mismo se hace para las otras tres direcciones.

Al final se guarda con la función H, el peso de ese movimiento.

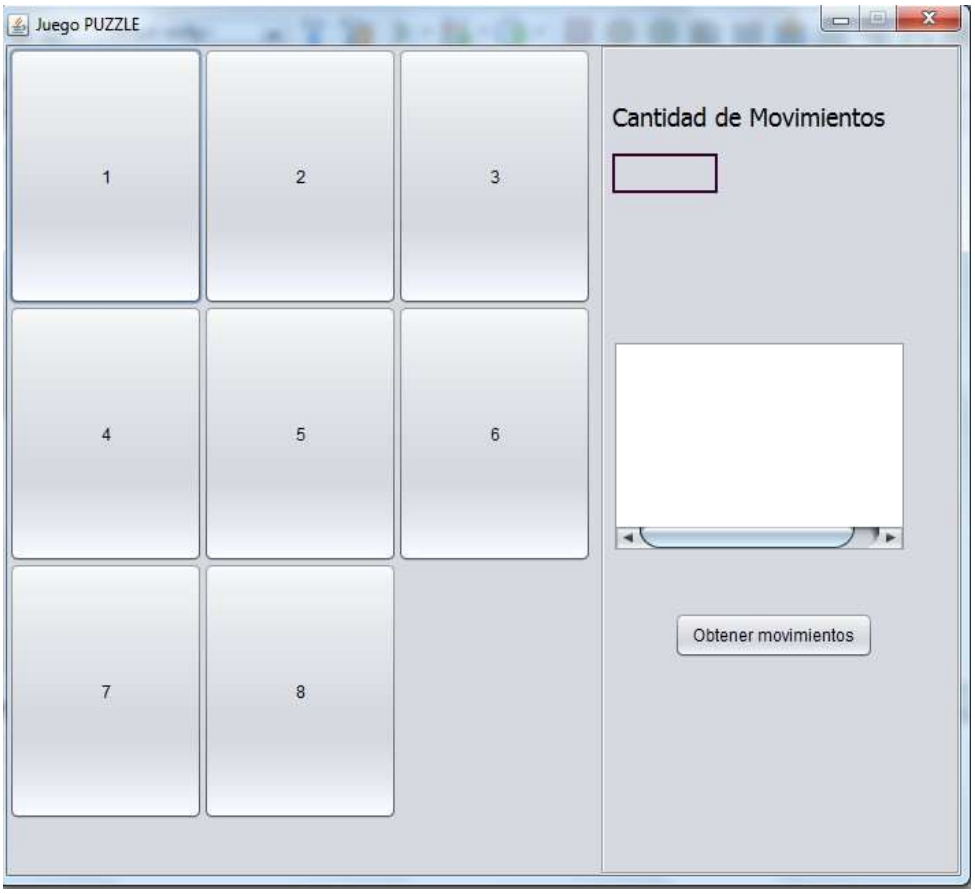
```
for (int i = 0; i < 4; i++) {
    if (fallos[i] == minimo && (minimo + nivel) < cota_g && cantidad_pasos > camino_local.size()) {

        //if (!esBucle(matrixes, matrices[i])) {
        int[] pos_cero = obtenerPosicionVacio(matrices[i]);

        //imprimeMatriz("cota: " + cota + " , fallas: " + minimo + " NIVEL " + nivel + ", Se genera
        camino.add("" + i);
        //System.out.println("1: " + camino.size() + " 2: " + camino_local.size());
        //matrixes_local.add(new Matriz(matrices[i]));
        generaArbol(matrices[i], camino, i, pos_cero[0], pos_cero[1], minimo + nivel, nivel + 1);
        //matrixes_local.remove(new Matriz(matrices[i]));

        camino = new ArrayList<>(camino_local);
        //} else {
        //    imprimeMatriz("bucle para, en el nivel: " + nivel, matrices[i]);
        //}
    }
}
```

Utilizando como referencia el mínimo valor de H, se analiza TODAS las posibilidades para los nodos con ese número. Siempre y cuando H + G (nivel) sea menor a la cota establecida por cuestiones de optimización.

Al generar una nueva ramificación se envían nuevos valores a considerar para la lista de movimientos.

Con esto analizamos todos los caminos viendo siempre mínimos valores.

4. Creación de la interfaz inicial



Se utilizaron botones con un texto que representa a los valores de la matriz con el fin de depurar más adelante. Botones para que reaccionen a los clicks que se den.

```
public void intercambio(JButton b1, JButton b2, int[] a, int[] b) {
    int aux = 0;
    b2.setText(b1.getText());
    b2.setIcon(b2.getIcon());
    b2.setVisible(true);
    b1.setVisible(false);
    aux = this.matriz[a[0]][a[1]];
    this.matriz[a[0]][a[1]] = this.matriz[b[0]][b[1]];
    this.matriz[b[0]][b[1]] = aux;
    for (int i = 0; i < matriz.length; i++) {
        for (int j = 0; j < matriz[0].length; j++) {
            System.out.print(matriz[i][j] + " ");
        }
        System.out.print("\n");
    }
    cont++;
    contador.setText(Integer.toString(cont));
}
```

En esta función intercambio se recibe el botón pulsado y el botón oculto en donde se ve un espacio vacío. a y b son arreglos que ayudan para obtener la posición de los botones en el espacio 3x3 de una matriz.

Cuando se da clic a un botón se oculta el mismo, y el espacio en blanco se reactiva.

5. Método para dividir la imagen original, en cada ficha.

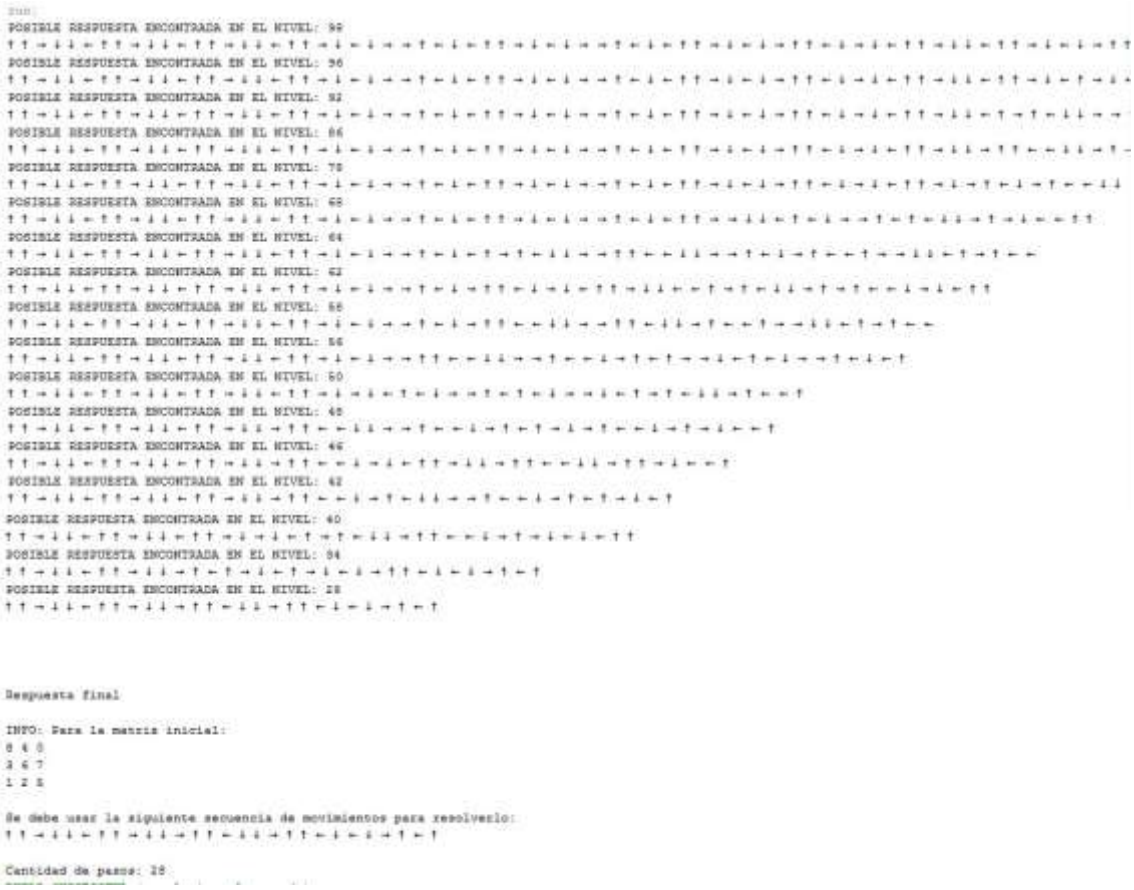
```
public void crearImg(Image imagenPrincipal){
    for(int i = 0; i < 3; i++){
        for (int j = 0; j  < 3; j++){
            if(i==2 && j==2){
                ImageIcon icon = new ImageIcon(getClass().getResource("/imagenes/fond.png"));
                imagen = createImage(new FilteredImageSource(icon.getImage().getSource(),
                    new CropImageFilter(0, 0, 150, 150)));
            }else{
                imagen = createImage(new FilteredImageSource(imagenPrincipal.getSource(),
                    new CropImageFilter(j*(width/3), i*(width/3) , width/3, width/3)));
            }
            imgList.add(imagen);
        }
    }
}
```

A partir de la imagen original que mide 450x450 pixeles, se la divide y se va creando imágenes de 150x150 pixeles, para lo cual se utiliza dos lazos for con dos variables que harán 9 iteraciones cada una, para poder recalcular el tamaño de las nuevas imágenes y de esta manera poder agregar a las fichas.

Análisis de resultados:

El algoritmo H en O(1) calcula cuantos pasos debe dar una pieza en total el coste de ese algoritmo es de O(3*3-1) = O(8) ya que se analizan hasta n piezas siendo n siempre 8 (el espacio vacío se ignora.

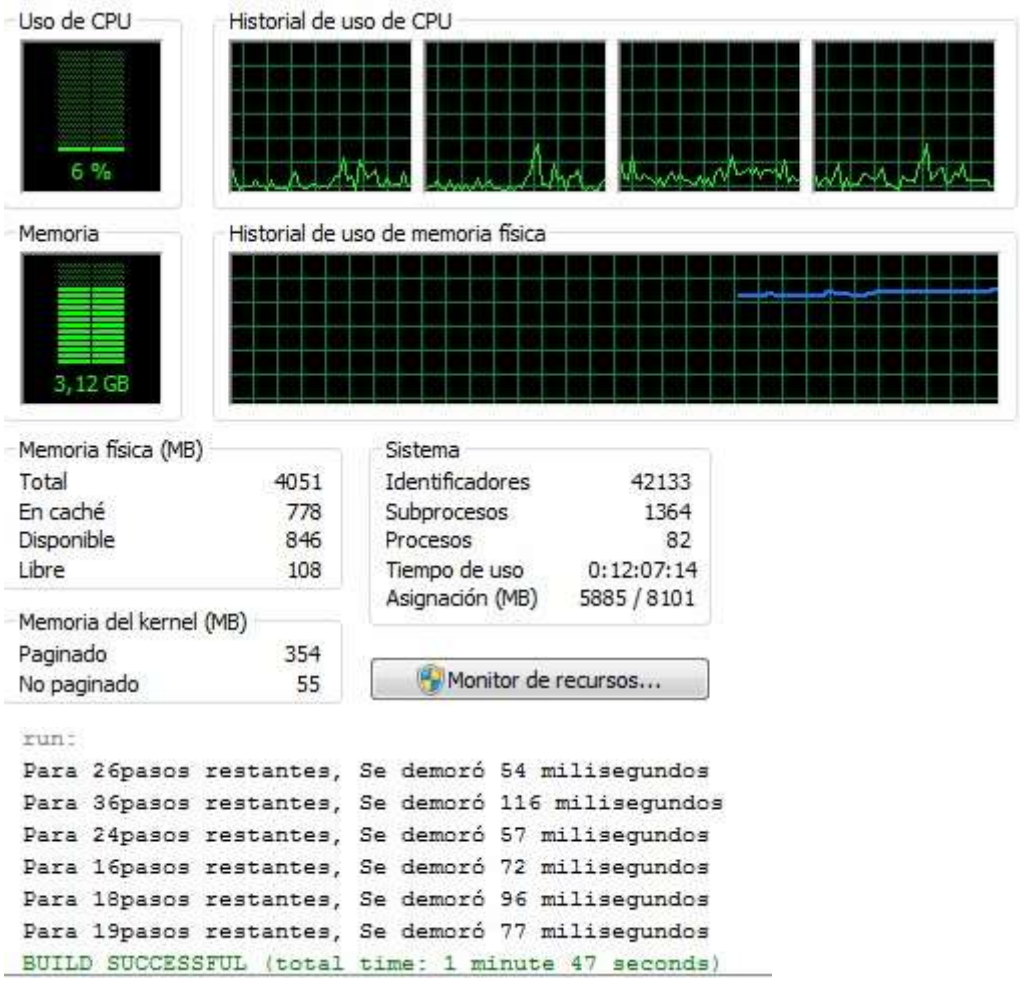
Al generar el árbol con los pasos a resolver inicialmente obtenemos una gran cantidad de pasos, luego ramifica lo que quedó pendiente siempre verificando que el número de pasos sea menor. De esta forma el camino final se sobreescribe muchas veces siempre optimizándose:



Las flechas indican como se debe mover ‘la matriz’, si se mueve hacia arriba entonces la pieza debajo del espacio vacío subirá.

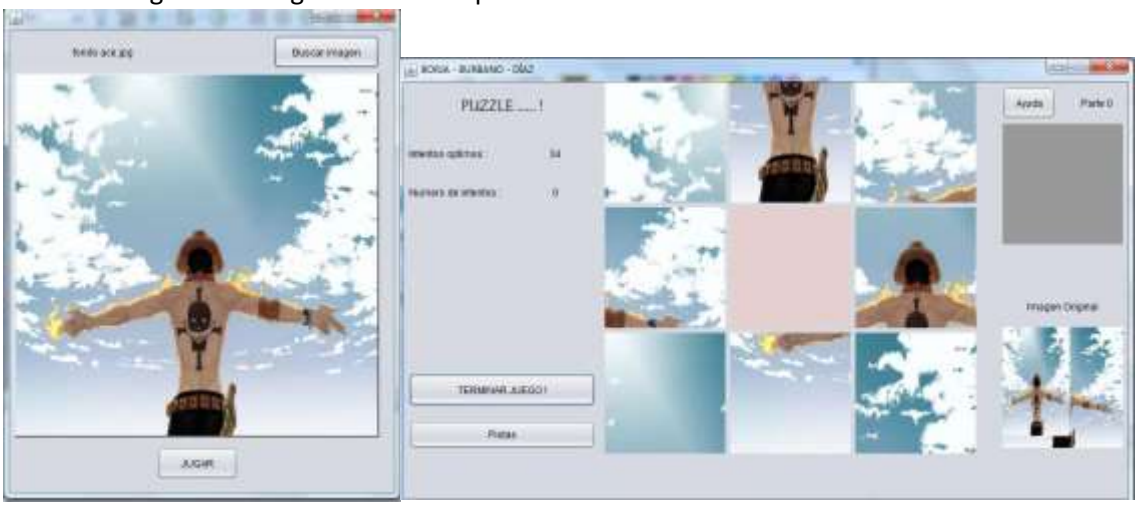
Los tiempos de ejecución para encontrar la solución a matrices que se resuelven con ciertos intentos óptimos son:

El uso de CPU aumenta en promedio un 2-3% en el peor de los casos a 4%.



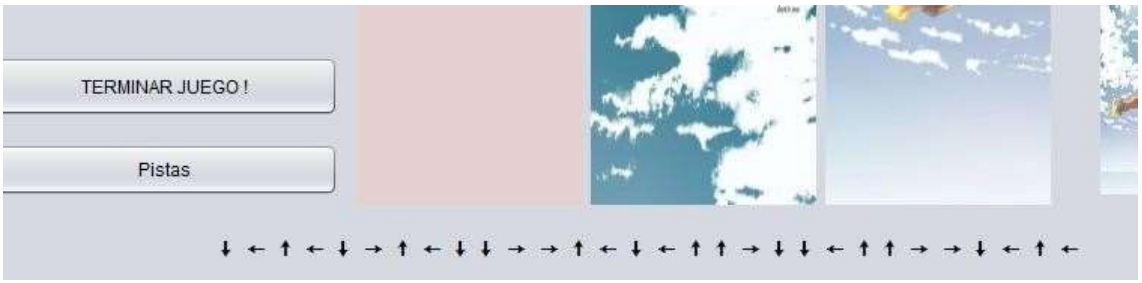
Es importante notar que para 19 pasos se demoró 77 milisegundos y para 18 pasos se demoró MÁS, no es error del contador; esto se da porque el tiempo de ejecución también depende de cómo se encuentra el procesador en ese momento y que picos está dando.

Podemos cargar una imagen de nuestra preferencia



El algoritmo desordena la matriz y carga las imágenes según la posición referenciada.

El algoritmo guarda como era la posición original y a que parte corresponde, para que con imágenes confusas el usuario pueda resolver el problema.



Al obtener las pistas podemos saber en qué dirección debemos mover para resolver todo el acertijo.

Conclusiones:

- Combinar varias técnicas vistas en el curso otorgan un algoritmo más óptimo pero mucho más complejo.
- La recursividad puede generar caminos ya explorados creando así un bucle.

- La complejidad del programa aumenta notoriamente cuando se trata de una matriz 4x4.

Recomendaciones:

- Siempre crear condiciones de salida a las funciones recursivas.
- Crear una animación que resuelva el problema como método de depuración.
- Darle un tiempo límite a la búsqueda de una solución para que si es una matriz imposible de resolver o con muchos pasos nos avise en lugar de marcar un error.

Bibliografía:

[1] [Véase en Línea] https://www.scoop.it/t/configuracion-de-servidoresweb/p/4043791047/2015/05/17/algoritmos-de-fuerza-bruta

[2] [Véase en Línea] https://es.wikipedia.org/wiki/Algoritmo_voraz

[3] ALGORITMOS VORACES, María Teresa Abad Soriano Departamento L.S.I., curso 2007/2008. Últimas vez visto [En línea]: http://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf

[4] Búsqueda en Inteligencia Artificial, Fernando Berzal, Últimas vez visto [En línea]: https://elvex.ugr.es/decsai/iaio/slides/A3%20Search.pdf

[5] Nils J. Nilsson: Principles of Artificial Intelligence. Morgan Kaufmann, 1986.