



Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

Ingeniería en Computación (INCO)

Inteligencia Artificial I

Sección: D02

NRC: 103847

Clave: I7038

Calendario: 2022B

"Practica 2 Puzzle búsqueda en amplitud"

Docente: Oliva Navarro Diego Alberto

Integrantes

Fernandez Roman Kevin Arturo: **216457949**

Flores Ontiveros Aide Sarahi **216587419**

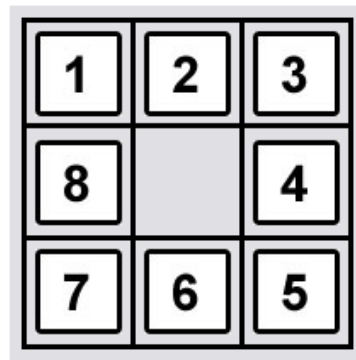
Madrigal Escoto Miguel Arturo **215767693**

Fecha de entrega: 17/11/2022

Objetivo

Resolver el problema 1 del documento Practica 2_SI. pdf utilizando la búsqueda en amplitud. Además, se debe realizar en equipos y adjuntar el código/ejecutable de los programas

Desarrollo



1	2	3
8		4
7	6	5

```
typedef vector<vector<int>> state;  
typedef pair<state,string> action;  
typedef map<action,action> map_action;  
typedef map<state,bool> map_visited;
```

Para simplificar la escritura del algoritmo y sus operaciones relacionadas, se definieron 4 tipos de datos para hacer referencia a ellos en donde sea necesario.

state: vector de vectores de enteros, para formar la matriz del tablero. Ej:

```
state start = {  
    {1,4,3},  
    {7,-1,6},  
    {5,8,2}
```

```
};
```

action: par para almacenar la matriz y el movimiento realizado en la pieza vacía (-1).

Ej:

```
state up = move(temp, "up");  
action after_up = make_pair(up, "up");
```

Dónde 'state up' es la matriz y 'up' denota un movimiento de la pieza vacía hacia arriba

map_action: mapa de acciones para almacenar los nodos y sus nodos adyacentes en forma de lista de lista de adyacencia. Ej:

```
map_action path;  
action after_up = make_pair(up, "up");  
path[after_up] = puzzle;
```

Dónde se agrega a la lista de adyacencia que el estado "after_up" es el padre del estado actual que se obtiene de un TDA cola.

map_visited: mapa para almacenar los nodos que han sido visitados, para que no volver a visitarlos y evitar perder tiempo en explorarlos y expandirlos, tal como lo sugiere el capítulo 3, apartado 3.5 del libro "Inteligencia Artificial Un Enfoque Moderno" de Russel y Norvig.

```
map_visited explored;  
if (!explored[puzzle.f]) {  
    explored[puzzle.f] = true;  
    ...  
    ...
```

Algoritmo búsqueda primero en amplitud (BFS)

BFS - De manera concisa, se entiende como estrategia sencilla en la que se expande primero el nodo raíz, a continuación se expanden todos los sucesores del nodo raíz, después sus sucesores, etc. En general, se expanden todos los nodos a una profundidad en el árbol de búsqueda antes de expandir cualquier nodo del próximo nivel. Posee una complejidad en tiempo y en espacio de $O(b^d)$. Donde b corresponde al factor de ramificación en el espacio de estados, y d , se refiere a la profundidad de la solución más superficial.

La implementación del algoritmo BFS para el problema propuesto(8-puzzle), recibe dos parámetros, un estado inicial(start), y un estado objetivo(target) de tipo state, para el que se buscará el camino más óptimo. Posteriormente, dentro del bloque de la función, se definen 4 variables que ayudan al control del flujo del algoritmo, un entero que almacena el costo del camino, una cadena de salida, que se encarga de almacenar el recorrido entre los estados, dos contenedores asociativos(maps) para almacenar los estados expandidos/explorados y la ruta/camino que el algoritmo está siguiendo, una variable que representa el 8-puzzle, y una cola que almacenará valores de tipo action.

```
void bfs(state start, state target){
    int path_cost = 0;
    string out = "";
    map_visited explored;
    map_action path;

    action puzzle;
    queue<action> q;
```

Posteriormente, se encola un par ordenado del parámetro del estado inicial(start), con una cadena que indica el movimiento. Aunado a lo anterior, al hashmap de la ruta(path), se agrega como clave, un vector compuesto nuevamente por el estado inicial, y una cadena que indica el movimiento a realizar(en este caso, al tratarse del estado inicial, se declara "initial"), y como llave, el mismo vector.

Después, a manera informativa, se imprime en pantalla que ha comenzado la expansión de nodos mediante BFS.

```
//{ state, "up"|"down"|"left"|"right" }  
q.push(make_pair(start,"initial"));  
  
path[{start,"initial"}] = {start,"initial"};  
  
cout << "Expandiendo con BFS" << ENDL;
```

A continuación, una iteración infinita comienza, se asigna a la variable puzzle el siguiente elemento de la cola(q.front()), de igual forma, este es eliminado de la misma, mediante q.pop(), posteriormente, se incrementa en una unidad el costo de la ruta, y se concatena a la variable out, el movimiento a realizar, accediendo al segundo elemento(que almacena el movimiento en una cadena) del par ordenado del tipo de dato action, además de agregar igualmente, el estado actual, haciendo uso de la función get_state(puzzle.f), a la que se le envía como parámetro el primer elemento que contiene un vector de vectores de enteros.

```
while (true){  
    puzzle = q.front(); q.pop();  
    ++path_cost;  
  
    out += "Mov: " + puzzle.s + ENDL;  
    out += get_state(puzzle.f) + ENDL;
```

Luego, se tiene la parte medular del algoritmo, una estructura de control selectiva, en la que si se cumple que el primer elemento del par ordenado contenido en puzzle es igual al estado objetivo, se imprime en pantalla que la solución ha sido encontrada, así como el número de nodos expandidos y la variable out, que almacena la secuencia de estados requerida para llegar desde el estado inicial hasta el estado objetivo, una vez hecho lo anterior, se ejecuta la sentencia break, y se entiende que el algoritmo ha terminado su ejecución satisfactoriamente, puesto que ha encontrado una ruta.

```

        // Si ya llegó al estado meta
        if (puzzle.f == target){
            cout << "Solucion Encontrada" << ENDL;
            cout << "Nodos expandidos: " << to_string(path_cost) <<
ENDL << ENDL;
            cout << out;
            break;
        }

```

En el caso de que la condición previa no se cumpla, se realiza en el bloque de la sentencia else, una nueva condición, en la que se accede al hashmap que almacena si un estado ya ha sido explorado/visitado, en caso de que no se tenga registro, se procede a marcar como explorado, y se inicial la expansión sobre este, creando una variable state temporal que nos facilitara el conocer el estado actual y operar con este sin afectar el camino/ruta recorrida hasta el momento.

A continuación, dado que existen 4 movimientos posibles, se realizan 4 condicionales para cada expansión posible en el espacio de estados, siendo que cuando el segundo elemento del par ordenado puzzle cumpla con ciertas condiciones de las estructuras selectivas, tales estados serán agregados a la cola para su posterior visita/exploración. Para esto, una vez que se cae en algún caso, se asigna a la variable temp el valor del primer elemento de la variable puzzle(que contiene el estado actual) y se crea una variable de tipo estado a la que se asigna el resultado de llamar a la función move(temp, <movimiento>), que simula el posible estado, una vez hecho lo anterior, se compara si el resultado previamente obtenido es distinto del almacenado en temp, de ser así se crea un par de tipo action con el estado creado y el movimiento tomado, una vez hecho lo anterior, se añade como clave y valor al hashmap que almacena la ruta y finalmente, se encola a la cola, para su posterior expansión.

```

        // Si aun no ha llegado
        else {
            if (!explored[puzzle.f]){

```

```

explored[puzzle.f] = true;

state temp;
// Mover hacia arriba
if (puzzle.s != "down"){
    temp = puzzle.f;
    state up = move(temp, "up");
    if (up != puzzle.f){
        action after_up = make_pair(up, "up");
        path[after_up] = puzzle;
        q.push(after_up);
    }
}

// Mover hacia la izquierda
if (puzzle.s != "right"){
    temp = puzzle.f;
    state left = move(temp, "left");
    if (left != puzzle.f){
        action after_left = make_pair(left, "left");
        path[after_left] = puzzle;
        q.push(after_left);
    }
}

// Mover hacia abajo
if (puzzle.s != "up"){
    temp = puzzle.f;
    state down = move(temp, "down");
    if (down != puzzle.f){
        action after_down = make_pair(down, "down");
        path[after_down] = puzzle;
        q.push(after_down);
    }
}

```



```

        fore(i,0,3){
            fore(j,0,3){
                // Buscar la posición vacia (-1)
                if(temp[i][j] == -1){
                    // Validar que se pueda mover hacia arriba
                    if (i != 0){
                        // Mover el que esta arriba de -1 a la
posición del -1
                        temp[i][j] = temp[i-1][j];

                        // Mover el -1 hacia abajo
                        temp[i-1][j] = -1;
                    }
                }
            }
        }
    }
    if (mov == "down"){
        //Recorrer la matriz del estado temporal
        // i = rows, c = cols
        fore(i,0,3){
            fore(j,0,3){
                // Buscar la posición vacia (-1)
                if(temp[i][j] == -1){
                    // Validar que se pueda mover hacia abajo
                    if (i != 2){
                        // Mover el que esta abajo de -1 a la
posición del -1
                        temp[i][j] = temp[i+1][j];

                        // Mover el -1 hacia abajo
                        temp[i+1][j] = -1;
                    }
                }
            }
        }
    }
    if (mov == "left"){
        //Recorrer la matriz del estado temporal
        // i = rows, c = cols
        fore(i,0,3){
            fore(j,0,3){
                // Buscar la posición vacia (-1)
                if(temp[i][j] == -1){
                    // Validar que se pueda mover hacia la izquierda
                    if (j != 0){
                        // Mover el que esta izquierda de -1 a la
posición del -1
                        temp[i][j] = temp[i][j-1];

                        // Mover el -1 hacia la izquierda
                        temp[i][j-1] = -1;
                    }
                }
            }
        }
    }
}

```

```

    }
    }
}

if (mov == "right"){
    //Recorrer la matriz del estado temporal
    // i = rows, c = cols
    fore(i,0,3){
        fore(j,0,3){
            // Buscar la posición vacia (-1)
            if(temp[i][j] == -1){
                // Validar que se pueda mover hacia la derecha
                if (j != 2){
                    // Mover el que esta derecha de -1 a la
posición del -1

                    temp[i][j] = temp[i][j+1];

                    // Mover el -1 hacia la derecha
                    temp[i][j+1] = -1;
                }
            }
        }
    }
}

return temp;
}

```

Esta función es de gran utilidad como complemento en el algoritmo de BFS, ya que es la que se encarga de validar si es que la pieza vacía en el tablero (-1) puede moverse a la derecha, izquierda, arriba o abajo.

Además, esta función recibe dos parámetros: un estado del tablero y un string que indica la dirección del movimiento de la pieza. Por ejemplo:

	1	2
3	4	5
6	7	8

Se tiene el siguiente estado durante la expansión de nodos de BFS, donde la pieza vacía se encuentra en la parte superior izquierda. Como puede observarse, esta pieza puede moverse únicamente hacia abajo o hacia la derecha.

Entonces, por ejemplo, para **mover la pieza vacía a la derecha**:

```
// Validar que se pueda mover hacia la derecha
if (j != 2){
    // Mover el que esta derecha de -1 a la
    posición del -1
    temp[i][j] = temp[i][j+1];

    // Mover el -1 hacia la derecha
    temp[i][j+1] = -1;
}
```

Se hace una validación, la cual consiste en que el índice que recorre las columnas (j) sea diferente de 2, ya que si fuera el caso la pieza vacía estaría en la última columna y eso indica que no hay una casilla disponible para colocar ahí la pieza vacía.

Una vez realizada esta validación, se trae el número que está a la derecha a la casilla vacía ($\text{temp}[i][j] = \text{temp}[i][j+1]$) y en la casilla donde estaba ese número se pone el -1 ($\text{temp}[i][j+1] = -1$);

Esto se aplica a los cuatro tipos de movimientos, en donde a cada uno le corresponde una validación e intercambio de posiciones específico. Cabe destacar que, si la validación falla y no se puede mover la pieza vacía a 'x' dirección, la función retorna el estado actual sin ninguna modificación, lo cual sirve en BFS para verificar si el estado actual ha cambiado si se le aplica un movimiento a la casilla vacía y expandir dicho nodo, puesto que si no ha cambiado el estado actual quiere decir que la pieza no se puede mover en 'x' dirección, por lo que estaríamos expandiendo un nodo repetido.

reconstruct path (map_action path, action s, action begin)

```
void reconstruct_path(map_action path, action s, action begin) {
    action curr = s;
    stack<action> stack_path;
    while (true) {
        stack_path.push(curr);
        if (curr == begin) break;

        stack_path.push(path[curr]);
        if (path[curr] == begin) break;

        curr = path[path[curr]];
    }

    cout << "Movimientos: " << sz(stack_path)-1 << ENDL << ENDL;

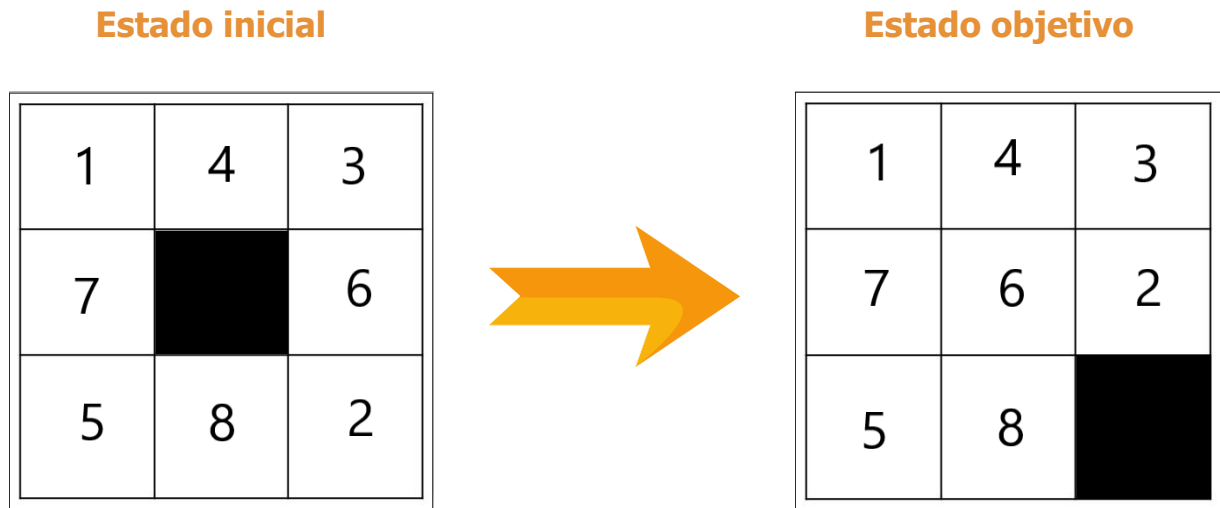
    while(!stack_path.empty()) {
        curr = stack_path.top(); stack_path.pop();
        cout << "Mov: " << curr.s << ENDL;
        cout << get_state(curr.f) << ENDL;
    }
}
```

El siguiente código tiene por objetivo la reconstrucción del camino generado por la BFS para llegar del estado inicial al final, es decir, mostrar las operaciones realizadas en cada uno de los estados para alcanzar el estado objetivo. Esto es posible ya que el método recibe un mapa con la expansión de cada nodo (lista de adyacencia), el último movimiento realizado junto con su estado y el estado inicial, el cual como tal no tiene un movimiento ya que es el primero.

Lo que se busca es utilizar un TDA pila para almacenar la ruta, la cual está en orden inverso, por lo que ésta estructura de datos será de utilidad para mostrar los nodos y los pasos en orden.

Caso 1

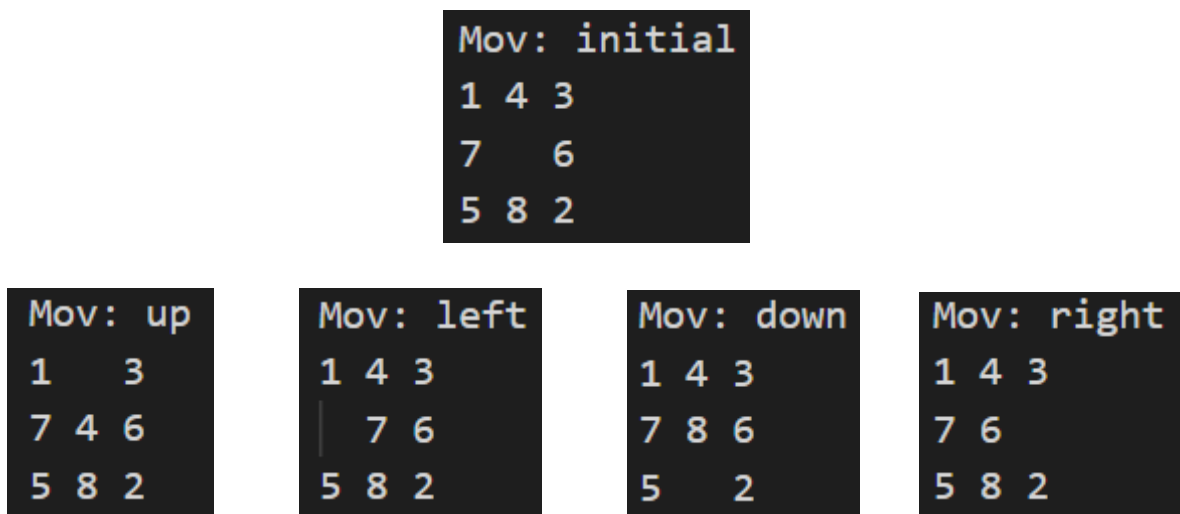
El caso consiste en llevar el puzzle de un estado inicial a un estado objetivo moviendo el cuadro vacío (arriba, izquierda, abajo, derecha) y luego comparar si este ya ha llegado a la configuración final:



La expansión de los nodos haciendo uso de la búsqueda en amplitud en la implementación de nuestro código es la siguiente:

Nodos expandidos: 13

Una vez tomado nuestro estado inicial existen 4 posibles movimientos (arriba, izquierda, abajo, derecha) que serían los nodos expandidos del segundo nivel:

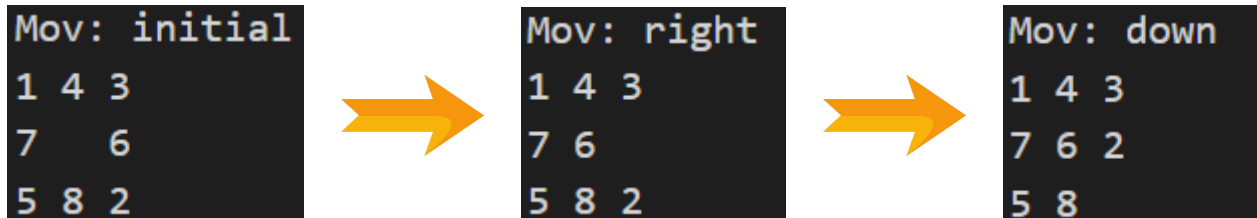


Una vez expandidos todos los nodos del segundo nivel pasamos al tercer y último nivel donde se expanden (arriba, izquierda, abajo, derecha) los siguientes nodos:

Mov: left	Mov: right	Mov: up	Mov: down	Mov: left	Mov: right	Mov: up	Mov: down
1 3 7 4 6 5 8 2	1 3 7 4 6 5 8 2	4 3 1 7 6 5 8 2	1 4 3 5 7 6 8 2	1 4 3 7 8 6 5 2	1 4 3 7 8 6 5 2	1 4 7 6 3 5 8 2	1 4 3 7 6 2 5 8

Por último, los pasos para llegar del estado inicial al estado objetivo son los siguientes:

Movimientos: 2



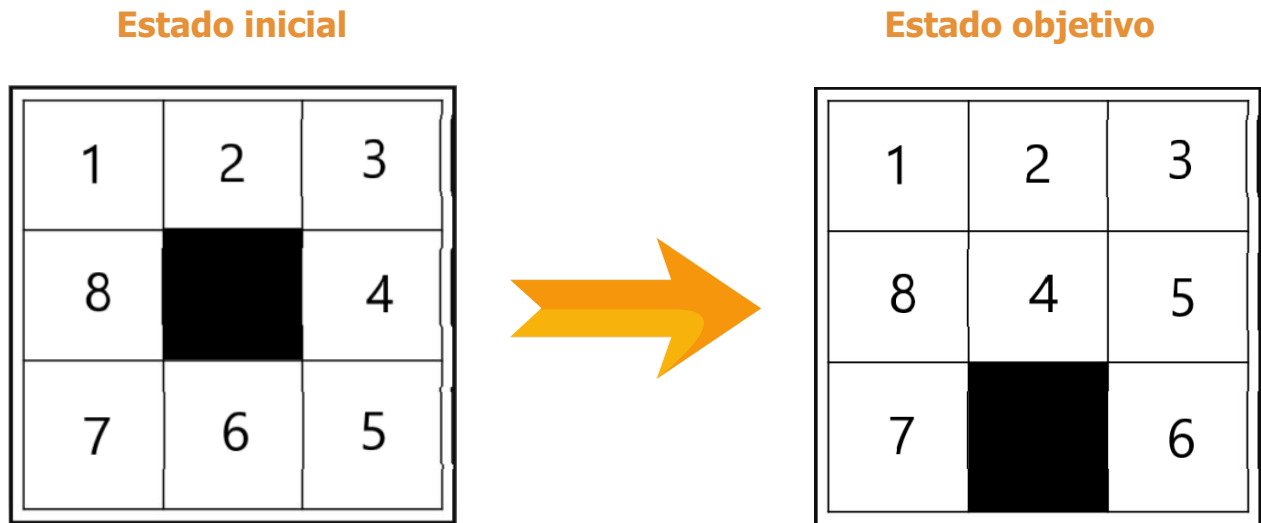
De esta forma se concluye este caso donde desde un estado inicial a un estado final se hace uso de la búsqueda en amplitud para buscar el camino y poder realizar los movimientos necesarios para llegar de la configuración inicial a la objetivo.

Se llegó al estado objetivo:

1	4	3
7	6	2
5	8	

Caso 2

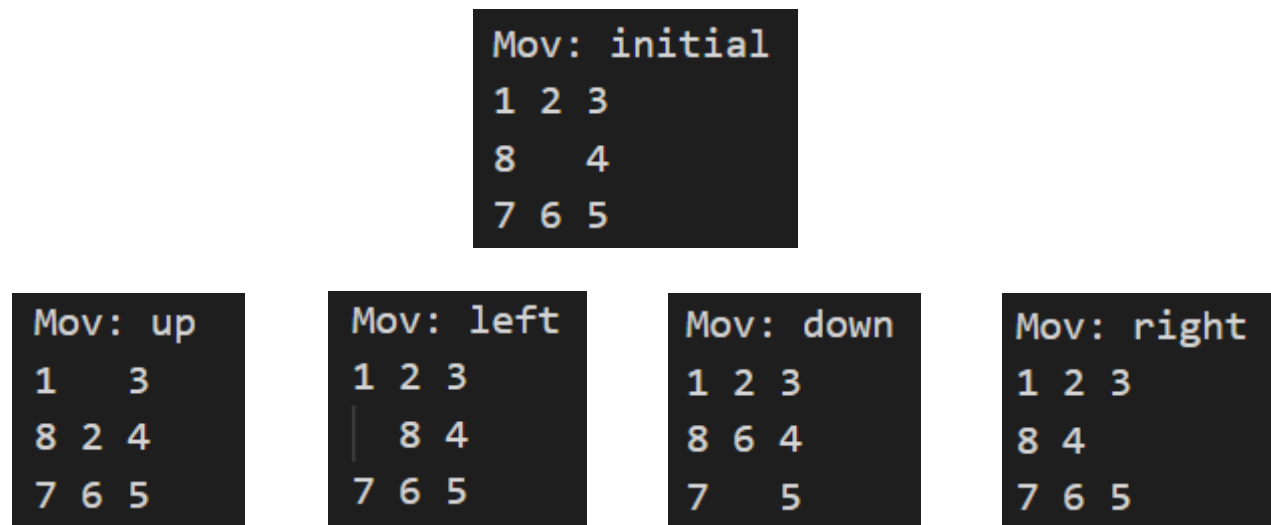
El caso consiste en llevar el puzzle de un estado inicial a un estado final moviendo el cuadro vacío (arriba, izquierda, abajo, derecha) y luego comparando si este ya ha llegado a la configuración final:



La expansión de los nodos haciendo uso de la búsqueda en amplitud en la implementación de nuestro código es la siguiente:

Nodos expandidos: 21

Una vez tomado nuestro estado inicial existen 4 posibles movimientos (arriba, izquierda, abajo, derecha) que serían los nodos expandidos del segundo nivel:



Una vez expandidos todos los nodos del segundo nivel pasamos al tercer nivel donde se expanden (arriba, izquierda, abajo, derecha) los siguientes nodos:

Mov: left	Mov: right	Mov: up	Mov: down	Mov: left	Mov: right	Mov: up	Mov: down
1 3 8 2 4 7 6 5	1 3 8 2 4 7 6 5	2 3 1 8 4 7 6 5	1 2 3 7 8 4 6 5	1 2 3 8 6 4 7 5	1 2 3 8 6 4 7 5	1 2 8 4 3 7 6 5	1 2 3 8 4 5 7 6

Una vez expandidos todos los nodos del tercer nivel pasamos al cuarto y último nivel donde se expanden los siguientes nodos:

Mov: down	Mov: down	Mov: right	Mov: right	Mov: up	Mov: up	Mov: left	Mov: left
8 1 3 7 2 4 6 5	1 3 4 8 2 5 7 6	2 3 1 8 4 7 6 5	1 2 3 7 8 4 6 5	1 2 3 6 4 8 7 5	1 2 3 8 6 7 5 4	1 2 8 4 3 7 6 5	1 2 3 8 4 5 7 6

Por último, los pasos para llegar del estado inicial al estado objetivo son los siguientes:

Movimientos: 3



De esta forma se concluye este caso donde desde un estado inicial a un estado final se hace uso de la búsqueda en amplitud para buscar el camino y poder realizar los movimientos necesarios para llegar de la configuración inicial a la objetivo.

Se llegó al estado objetivo:

1	2	3
8	4	5
7		6

Enlace al código en GitHub

<https://github.com/Miguel-Arturo-Madrigal-Escoto/8-puzzle-problem---BFS>

Conclusión

Con el ejercicio de esta práctica correspondiente a la resolución del famoso problema 8-puzzle, mediante distintos algoritmos de búsqueda con la intención de medir su desempeño de acuerdo a ciertas métricas, como su complejidad en tiempo y espacio; se analizó en este caso, la aplicación del algoritmo de búsqueda primero en anchura con la intención de proponer una ruta desde un estado inicial hacia un estado objetivo([1, 2, 3], [4, 5, 6], [7, 8, 9]). Es importante recalcar las ventajas como que si existen múltiples soluciones siempre encontrará la más corta, además de nunca quedar atrapado en un callejón sin salida; así como sus desventajas que son la complejidad en tiempo y las restricciones de memoria(ya que almacena todos los nodos del nivel actual antes de pasar al siguiente) que posee la búsqueda primero en anchura(BFS), ya que sabiendo de antemano esto, se puede determinar a priori si el algoritmo será óptimo para el problema dado. No obstante, considerando el poder de cómputo actual, y sabiendo que el espacio de estados del 8-puzzle es de 181,440, para estándares actuales resulta computable. Se eligió trabajar con el lenguaje de programación C++, ya que posee bastantes utilidades y prestaciones para implementar algoritmos con este, no por nada, es de los lenguajes más utilizados en este ámbito, además de que permite optimizar aún más el rendimiento en comparación a otros lenguajes.