



### Ejercicio 1 (5 puntos)

Basándose en los resultados del apartado b) del proyecto (solución, coste de la solución y número de nodos expandidos) responda a las siguientes cuestiones:

A. ¿Es la búsqueda en anchura óptima? (Responda SI/NO y por qué)

**Si, porque todos los costes son iguales.**

B. ¿Es la heurística h1 admisible? En caso afirmativo, diga por qué y en caso negativo indique un nodo que no cumpla la propiedad.

**No, por ejemplo el nodo P tiene como valor  $h1(P)=3$  y el coste para alcanzar el nodo objetivo G desde P es 1 pues solo se necesita un movimiento del caballo.**

C. ¿Qué heurística es mejor h1 o h2? ¿Por qué?

**h2, porque h2 expande un menor número de nodos.**

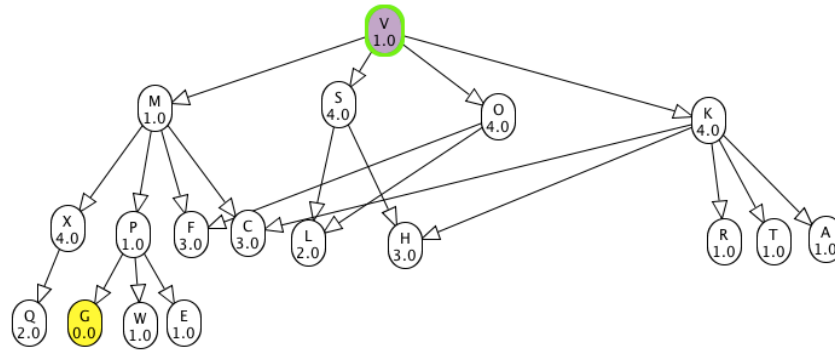
D. Diseñe una heurística h3 que mejore la búsqueda en profundidad. Para ello, defina directamente los valores de la heurística h3 sobre cada nodo usando la tabla siguiente.

Nodo	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Valor h3			4		1	4	0				4		1		4	1			4			1	1	4
Coste			2		3	2	3				1		1		1	2			1			0	3	2
h3+coste			6		4	6	3				5		2		5	3			5			1	4	6

Con esos valores de h3 (los valores que faltan no afectan) aseguramos que sólo se expandan 4 nodos en lugar de 6 nodos que expande la búsqueda en profundidad. Especificamos aquí la frontera:

	Frontera											
iter = 0	V											
iter = 1		M	S	O	K							
iter = 2			P	S	O	K	X	F	C			
iter = 3				G	W	E	S	O	K	X	F	C

A continuación, dibuje el árbol de búsqueda en la herramienta Search y use el algoritmo A\* con la heurística h3 para verificar que mejora la búsqueda en profundidad. Guarde el fichero con el nombre Ajedrez.xml.




## Ejercicio 2 (5 puntos)

Realizar la implementación que muestre los movimientos para capturar al rey negro con un alfil de paso uno, es decir, que no puede avanzar más de una casilla cada vez. Los posibles movimientos se harán en diagonal siguiendo el orden de las agujas de un reloj, comenzando por la casilla de arriba a la izquierda.

En este caso la casilla de salida es la W, y se reduce el número de filas y columnas a 4, desapareciendo la primera fila y la última columna. El rey permanecerá en la casilla G.

Implementar la solución utilizando las clases java del paquete AIMA.

A	B	C	D	
E	F	G 	H	I
J	K	L	M	N
O	P	Q	R	S
T	U	V 	W	X

```
package aima.core.environment.alfil;
import aima.core.agent.Action;
import aima.core.agent.impl.DynamicAction;

public class AlfilBoard {
    public static Action R1U1 = new DynamicAction("1 derecha+1 arriba");
    public static Action R1D1 = new DynamicAction("1 derecha+1 abajo");
    public static Action L1D1 = new DynamicAction("1 izquierda+1 abajo");
    public static Action L1U1 = new DynamicAction("1 izquierda+1 arriba");
    private int row;
    private int column;

    public AlfilBoard() {
        this.row = 3;
        this.column = 3;
    }
    public AlfilBoard(int r, int c) {
        this.row = r;
        this.column = c;
    }
    public AlfilBoard(AlfilBoard copyBoard) {
        this(copyBoard.getRow(), copyBoard.getColumn());
    }
    public int getColumn() {
        return column;
    }
    public int getRow() {
        return row;
    }
    public void moveR1U1() {
        this.column++;
        this.row--;
    }
}
```

## FORMULACION:

- ESTADO: Coordenadas X e Y de la matriz
- ESTADO INICIAL:  $x=3, y=3$
- FUNCION OBJETIVO: estamos en  $x=2, y=0$ ?
- ACCIONES:
  - A1: L1U1  $\rightarrow x-=1, y-=1$
  - A2: R1U1  $\rightarrow x+=1, y-=1$
  - A3: R1D1  $\rightarrow x+=1, y+=1$
  - A4: L1D1  $\rightarrow x-=1, y+=1$
- MODELO DE TRANSICION: El estado resultante será el resultado de sumar o resta 1 a x e y, según se aplique en cada accion
- RESTRICCIONES: Valores ilegales:  $x < 0, x > 3, y < 0, y > 3$
- COSTE: Cada accion tiene un coste de 1. La solucion será la secuencia de acciones empleada

BFS: L1U1, L1U1, R1U1

PATH COST = 3

NODES EXP = 3

MAXQUEUE SIZE = 4

DFS: L1U1, L1D1, L1U1, R1U1, R1U1

PATHCOST=5

NODESEXP=5

MAXQUEUE SIZE=4

```

public void moveR1D1() {
    this.column++;
    this.row++;
}
public void moveL1D1() {
    this.column--;
    this.row++;
}
public void moveL1U1() {
    this.column--;
    this.row--;
}
public boolean canMoveAlfil(Action where) {
    boolean retVal = true;
    int nextRow=-1, nextColumn=-1;

    if (where.equals(R1U1)) {
        nextColumn = getColumn() + 1;
        nextRow = getRow() - 1;
    } else if (where.equals(R1D1)) {
        nextColumn = getColumn() + 1;
        nextRow = getRow() + 1;
    } else if (where.equals(L1D1)) {
        nextColumn = getColumn() - 1;
        nextRow = getRow() + 1;
    } else if (where.equals(L1U1)) {
        nextColumn = getColumn() - 1;
        nextRow = getRow() - 1;
    }
    retVal = ((nextRow >= 0 && nextRow <=3)
        && (nextColumn >= 0 && nextColumn <=3));
    return retVal;
}
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final AlfilBoard other = (AlfilBoard) obj;
    if (this.row != other.row) {
        return false;
    }
    if (this.column != other.column) {
        return false;
    }
    return true;
}
public int hashCode() {
    int hash = 3;
    hash = 31 * hash + this.row;
    hash = 31 * hash + this.column;
    return hash;
}
public String toString() {
    return "AlfilBoard{" + "row=" + row + ", column=" + column + '}';
}
}

```

```

package aima.core.environment.alfil;
import java.util.LinkedHashSet;
import java.util.Set;
import aima.core.agent.Action;
import aima.core.search.framework.ActionsFunction;
import aima.core.search.framework.ResultFunction;

public class AlfilFunctionFactory {
    private static ActionsFunction _actionsFunction = null;
    private static ResultFunction _resultFunction = null;

    public static ActionsFunction getActionsFunction() {
        if (null == _actionsFunction) {
            _actionsFunction = new AlfActionsFunction();
        }
        return _actionsFunction;
    }
    public static ResultFunction getResultFunction() {
        if (null == _resultFunction) {
            _resultFunction = new AlfResultFunction();
        }
        return _resultFunction;
    }
    private static class AlfActionsFunction implements ActionsFunction {
        public Set<Action> actions(Object state) {
            AlfilBoard board = (AlfilBoard) state;
            Set<Action> actions = new LinkedHashSet<Action>();

            if (board.canMoveAlfil(AlfilBoard.L1U1)) {
                actions.add(AlfilBoard.L1U1);
            }
            if (board.canMoveAlfil(AlfilBoard.R1U1)) {
                actions.add(AlfilBoard.R1U1);
            }
            if (board.canMoveAlfil(AlfilBoard.R1D1)) {
                actions.add(AlfilBoard.R1D1);
            }
            if (board.canMoveAlfil(AlfilBoard.L1D1)) {
                actions.add(AlfilBoard.L1D1);
            }
            return actions;
        }
    }
    private static class AlfResultFunction implements ResultFunction {
        public Object result(Object s, Action a) {
            AlfilBoard board = (AlfilBoard) s;

            if (AlfilBoard.R1U1.equals(a)
                && board.canMoveAlfil(AlfilBoard.R1U1)) {
                AlfilBoard newBoard = new AlfilBoard(board);
                newBoard.moveR1U1();
                return newBoard;
            } else if (AlfilBoard.R1D1.equals(a)
                && board.canMoveAlfil(AlfilBoard.R1D1)) {
                AlfilBoard newBoard = new AlfilBoard(board);
                newBoard.moveR1D1();
                return newBoard;
            } else if (AlfilBoard.L1D1.equals(a)
                && board.canMoveAlfil(AlfilBoard.L1D1)) {
                AlfilBoard newBoard = new AlfilBoard(board);
                newBoard.moveL1D1();
                return newBoard;
            } else if (AlfilBoard.L1U1.equals(a)
                && board.canMoveAlfil(AlfilBoard.L1U1)) {
                AlfilBoard newBoard = new AlfilBoard(board);
                newBoard.moveL1U1();
                return newBoard;
            }
        }
    }
}

```

```

        return s;
    }
}

```

```

package aima.core.environment.alfil;
import aima.core.search.framework.GoalTest;

public class AlfilGoalTest implements GoalTest {
    AlfilBoard goal = new AlfilBoard(0,2);

    public boolean isGoalState(Object state) {
        AlfilBoard board = (AlfilBoard) state;
        return board.equals(goal);
    }
}

```

```

package aima.gui.demo.search;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;
import aima.core.agent.Action;
import aima.core.environment.alfil.*;
import aima.core.search.framework.*;
import aima.core.search.uninformed.BreadthFirstSearch;
import aima.core.search.uninformed.DepthFirstSearch;

public class AlfilAjedrezDemo {
    static AlfilBoard boardInicial = new AlfilBoard();

    public static void main(String[] args) {
        breadthDemo();
        depthGraphDemo();
    }

    private static void breadthDemo() {
        System.out.println("\nAlfilDemo breadth -->");
        try {
            Problem problem = new Problem(boardInicial,
                AlfilFunctionFactory.getActionsFunction(),
                AlfilFunctionFactory.getResultFunction(),
                new AlfilGoalTest());
            Search search = new BreadthFirstSearch();
            SearchAgent agent = new SearchAgent(problem, search);
            printActions(agent.getActions());
            printInstrumentation(agent.getInstrumentation());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void depthGraphDemo() {
        System.out.println("\nAlfilDemo depth graph -->");
        try {
            Problem problem = new Problem(boardInicial,
                AlfilFunctionFactory.getActionsFunction(),
                AlfilFunctionFactory.getResultFunction(),
                new AlfilGoalTest());
            Search search = new DepthFirstSearch(new GraphSearch());
            SearchAgent agent = new SearchAgent(problem, search);
            printActions(agent.getActions());
            printInstrumentation(agent.getInstrumentation());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
private static void printInstrumentation(Properties properties) {
    Iterator<Object> keys = properties.keySet().iterator();
    while (keys.hasNext()) {
        String key = (String) keys.next();
        String property = properties.getProperty(key);
        System.out.println(key + " : " + property);
    }
}

private static void printActions(List<Action> actions) {
    for (int i = 0; i < actions.size(); i++) {
        String action = actions.get(i).toString();
        System.out.println(action);
    }
}
}
```