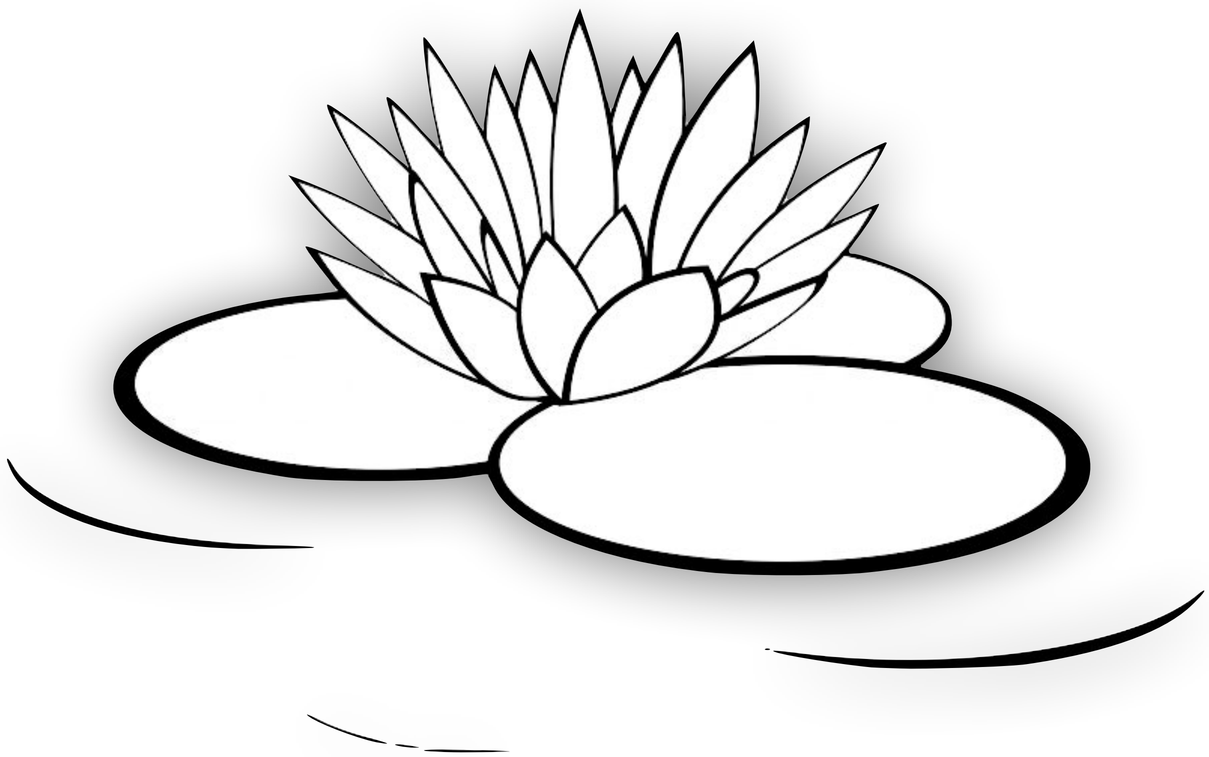
The Jumping Frogs Puzzle

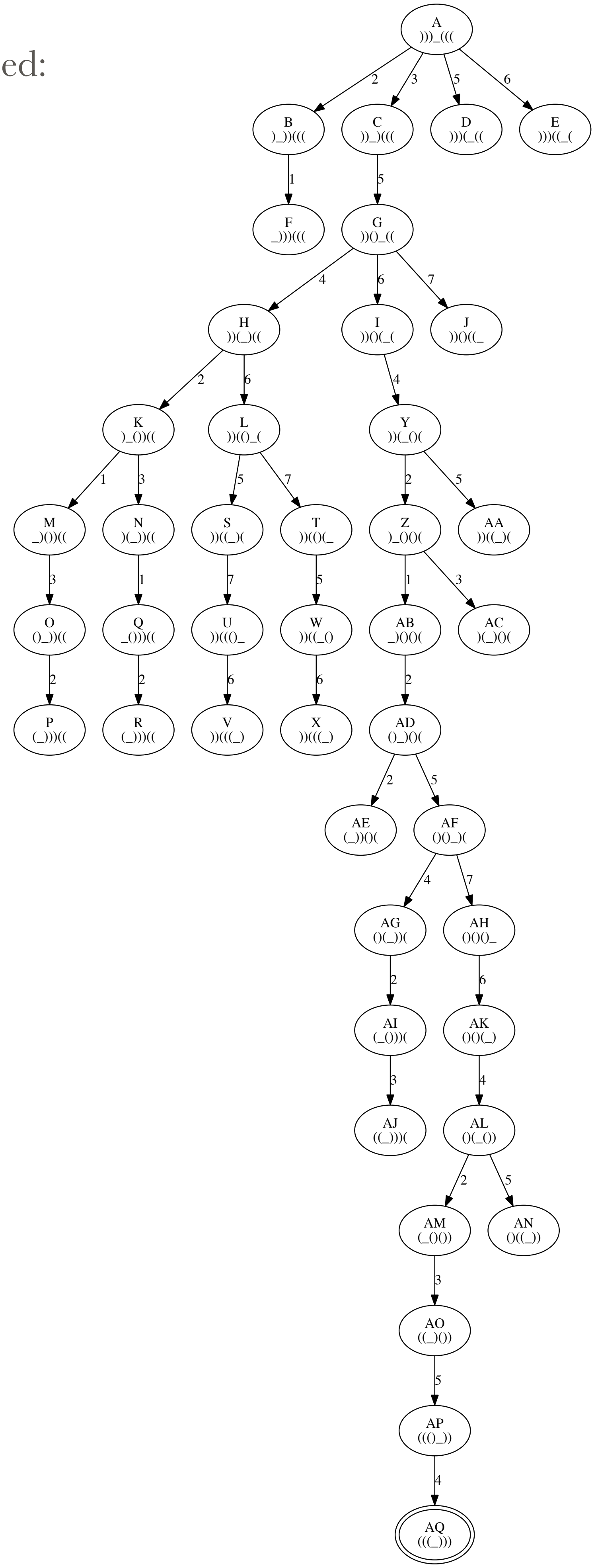
Artificial Intelligence



Nicolas Tsagarides - U134N0275

Fall 2014

Tree searched:



OPEN and CLOSE lists

Iteration	OPEN	CLOSE
1	A	
2	B, C, D, E	A
3	F, C, D, E	A, B
4	C, D, E	A, B, F
5	G, D, E	A, B, F, C
6	H, I, J, D, E	A, B, F, C, G
7	K, L, I, J, D, E	A, B, F, C, G, H
8	M, N, L, I, J, D, E	A, B, F, C, G, H, K
9	O, N, L, I, J, D, E	A, B, F, C, G, H, K, M
10	P, N, L, I, J, D, E	A, B, F, C, G, H, K, M, O
11	N, L, I, J, D, E	A, B, F, C, G, H, K, M, O, P
12	Q, L, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N
13	R, L, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q
14	L, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R
15	S, T, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, L
16	U, T, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S
17	V, T, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U
18	T, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V
19	W, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T
20	X, I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W
21	I, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X
22	Y, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I
23	Z, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y
24	AB, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z

Iteration	OPEN	CLOSE
25	AD, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB
26	AE, AF, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD
27	AF, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE
28	AG, AH, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF
29	AI, AH, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG
30	AJ, AH, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI
31	AH, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ
32	AK, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH
33	AL, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH, AK
34	AM, AN, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH, AK, AL
35	AO, AN, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH, AK, AL, AM
36	AP, AN, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH, AK, AL, AM, AO
37	AQ, AN, AC, AA, J, D, E	A, B, F, C, G, H, K, M, O, P, N, Q, R, S, U, V, T, W, X, I, Y, Z, AB, AD, AE, AF, AG, AI, AJ, AH, AK, AL, AM, AO, AP

Nodes Expanded: 29

Data Structures:

```
enum lily {EMPTY, GREEN, BROWN};

class State
{
public:
    lily lake[7];
    State* parent;
    int derivedFromOperation = -1;
    lily colorOfParent;

     State(lily[], State*);
     State(lily[], State*, int);
     State(lily[], State*, int, lily);
     State(State*);
     static void childOf(State*, State*);
 }
```

I created an enumeration `lily` with values of `EMPTY`, `GREEN`, and `BROWN` to represent the state of a lily-pad.

I created a class `State` to have it used as a node while building the search tree.

The class has a `lily` array of 7 elements called `lake` which represents the state of the 7 lily-pads that are on the lake.

I am using a `State` pointer named `parent` used to point to the parent of each node.

There is an integer `derivedFromOperation` that is used to store the action that produced this node. It has the default value of -1 which means no operation was performed yet.

The `colorOfParent` is used to store the color of the frog that jumped to produce this node.

I used several constructors to set-up the data-members mentioned above.

I also created a `static void childOf()` function that takes 2 `State` pointers as arguments. It sets the second `State` as a parent of the first `State`

The code:

```
// Nicolas Tsagarides
// 29/11/2014
// The jumping frogs puzzle
//
// Operations:
// Green jumps one spot
// Green jumps two spots
// Brown jumps one spot
// Brown jumps two spots
// Simplified into one operation:
// Frog on place x jumps to the empty spot
// The priority is from frog x = 0 to frog x = 6

#include <iostream>
#include <stack>
using namespace std;

enum lily {EMPTY, GREEN, BROWN};
enum validation {INVALID, TWOLEFT, ONELEFT, TWORIGHT, ONERIGHT};

class State
{
public:
    lily lake[7];
    State* parent;
    int derivedFromOperation = -1; // -1 means no operation was performed
    lily colorOfParent;

    State(lily[], State*);
    State(lily[], State*, int);
    State(lily[], State*, int, lily);
    State(State*);
    static void childOf(State*, State*);
};

validation valid(lily[], int);
bool lakesEqual(lily[], lily[]);
bool nodeAlreadyExpanded(lily[], stack <State>);

int main (int argc, char const *argv[])
{
    lily startingState[] = {GREEN, GREEN, GREEN, EMPTY, BROWN, BROWN,
BROWN}; // This is the starting state
    lily acceptedState[] = {BROWN, BROWN, BROWN, EMPTY, GREEN, GREEN,
GREEN}; // This is the goal

    stack <State> open; // OPEN stack
    stack <State> close; // CLOSE stack
    stack <State> expandedStates; // temporary stack to put the expanded
states before pushing them in the open stack

    open.push(State(startingState, NULL, -1, EMPTY)); // push the starting
state to the OPEN stack

    State* ParentNode;
    State* ThisNode;

    lily currentState[7];
    lily currentColorOfParent = EMPTY;
```

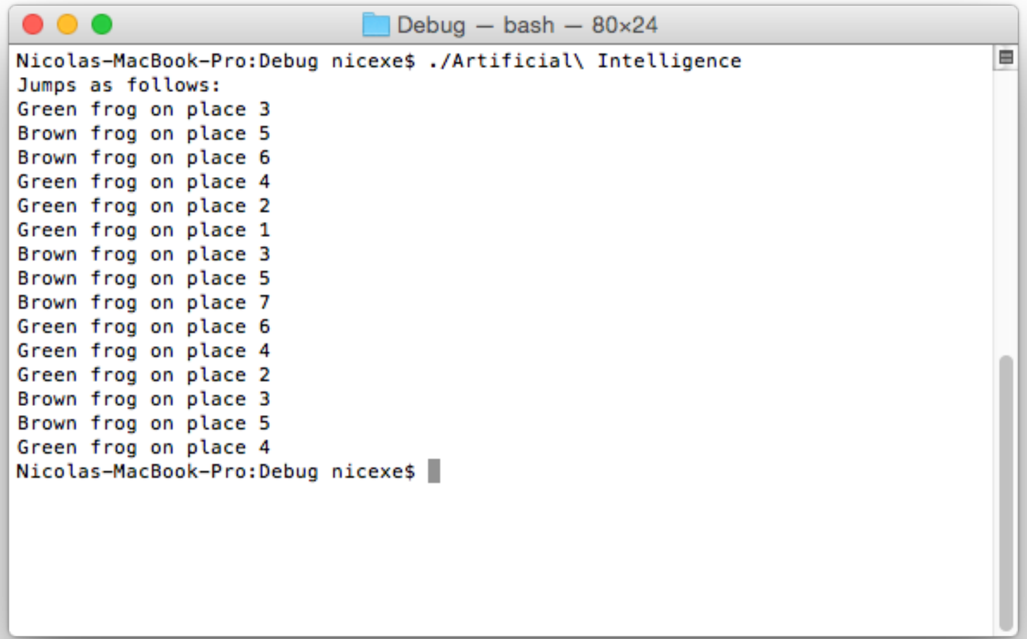
```
    int currentOperation;

    while (!open.empty())
    {
        copy(begin(open.top().lake), end(open.top().lake),
begin(currentState)); // get the current state
        ParentNode = open.top().parent; // saving the parent of the state
to be pushed into the CLOSE stack later
        ThisNode = new State(&open.top()); // saving this node to be used
as a parent on the expanded nodes
        currentOperation = open.top().derivedFromOperation;
        currentColorOfParent = open.top().colorOfParent;

        open.pop(); // pop the current state from the OPEN stack

        if (lakesEqual(currentState, acceptedState))
        {
            stack <State*> solutionPath;
            solutionPath.push(ThisNode);

            while (solutionPath.top()->parent != NULL) // filling the
solutionPath stack with the nodes
            {
                solutionPath.push(ThisNode->parent);
                ThisNode=ThisNode->parent;
            }

            cout << "Jumps as follows:\n";

            solutionPath.pop(); // discard the first node from the stack
since its the root node

            while ( !solutionPath.empty() )
            {
                string color;
                if (solutionPath.top()->colorOfParent == GREEN)
                {
                    color = "Green";
                }
                else if (solutionPath.top()->colorOfParent == BROWN)
                {
                    color = "Brown";
                }
                cout << color << " frog on place " << solutionPath.top()-
>derivedFromOperation + 1 << endl;
                solutionPath.pop();
            }

            break;
        }
        else
        {
            close.push(State(currentState, ParentNode, currentOperation,
currentColorOfParent));
            for (int i=0; i<7; i++)
            {
                validation action = valid(currentState, i); // checking
which action is valid if any

                lily newState[7];

                for (int j=0; j<7; j++)
```

```
                {
                    newState[j]=currentState[j];
                }

                switch(action) // performs the valid action if any
                {
                    case TWOLEFT:
                        newState[i]=EMPTY; // frog jumps
                        newState[i-2]=BROWN; // frog lands 2 spots to the
left
                        if ( !nodeAlreadyExpanded(newState, open) && !
nodeAlreadyExpanded(newState, close) ) // checking if current state is
already in the open or close stack
                        {
                            expandedStates.push(State(newState, ThisNode,
i, BROWN)); // add the current state to the OPEN stack
                        }
                        break;
                    case ONELEFT:
                        newState[i]=EMPTY;
                        newState[i-1]=BROWN;
                        if ( !nodeAlreadyExpanded(newState, open) && !
nodeAlreadyExpanded(newState, close) )
                        {
                            expandedStates.push(State(newState, ThisNode,
i, BROWN));
                        }
                        break;
                    case TWORIGHT:
                        newState[i]=EMPTY;
                        newState[i+2]=GREEN;
                        if ( !nodeAlreadyExpanded(newState, open) && !
nodeAlreadyExpanded(newState, close) )
                        {
                            expandedStates.push(State(newState, ThisNode,
i, GREEN));
                        }
                        break;
                    case ONERIGHT:
                        newState[i]=EMPTY;
                        newState[i+1]=GREEN;
                        if ( !nodeAlreadyExpanded(newState, open) && !
nodeAlreadyExpanded(newState, close) )
                        {
                            expandedStates.push(State(newState, ThisNode,
i, GREEN));
                        }
                        break;
                    default:
                        break;
                }
            }
            while (!expandedStates.empty()) // emptying the expanded
states in the open stack
            {
                open.push(expandedStates.top());
                expandedStates.pop();
            }
        }
    }

    return 0;
```

```
}

State::State (lily la[], State* p) // creating a state with a specified
lake layout and a specified parent
{
    State::childOf(this, p);
    for (int i=0; i<7; i++)
    {
        lake[i]=la[i];
    }
}

State::State (lily la[], State* p, int op) : State(la, p) // also setting
the operation that this node was generated from
{
    derivedFromOperation = op;
}

State::State (lily la[], State* p, int op, lily c) : State(la, p, op) //
also setting the color of the frog from the parent node
{
    colorOfParent = c;
}

State::State (State* s)
{
    for (int i=0; i<7; i++)
    {
        lake[i]=s->lake[i];
    }

    derivedFromOperation = s->derivedFromOperation;

    parent = s->parent;

    colorOfParent = s->colorOfParent;
}

void State::childOf(State* c, State* p) // setting the parent of the state
{
    c->parent=p;
}

validation valid (lily lake[], int i)
{
    if (lake[i]==GREEN) // check in the frog on i is green
    {
        if (i<6) // check if the frog goes out of bounds
        {
            if ( (lake[i+1]) == EMPTY) // check if there is an empty spot
            {
                return (ONERIGHT);
            }
        }
        if (i<5)
        {
            if ( (lake[i+2]) == EMPTY)
            {
                return (TWORIGHT);
            }
        }
    }
```

```
    else if (lake[i]==BROWN)
    {
        if (i>0)
        {
            if (lake[i-1]==EMPTY)
            {
                return (ONELEFT);
            }
        }
        if (i>1)
        {
            if (lake[i-2]==EMPTY)
            {
                return (TWOLEFT);
            }
        }
    }

    return (INVALID);
}

bool lakesEqual(lily a[], lily b[]) // check if lakes are equal
{
    bool equality = true;
    for (int i=0; i<6; i++)
    {
        if (a[i]!=b[i])
        {
            equality=false;
        }
    }

    return (equality);
}

bool nodeAlreadyExpanded(lily la[], stack <State> st) // check if the
state already exist in the stack
{
    while (!st.empty())
    {
        if (lakesEqual(la, st.top().lake))
        {
            return true;
        }
        st.pop();
    }
    return false;
}
```

Screenshot:



Debug — bash — 80×24

```
Nicolas-MacBook-Pro:Debug nicexe$ ./Artificial\ Intelligence
Jumps as follows:
Green frog on place 3
Brown frog on place 5
Brown frog on place 6
Green frog on place 4
Green frog on place 2
Green frog on place 1
Brown frog on place 3
Brown frog on place 5
Brown frog on place 7
Green frog on place 6
Green frog on place 4
Green frog on place 2
Brown frog on place 3
Brown frog on place 5
Green frog on place 4
Nicolas-MacBook-Pro:Debug nicexe$
```