THE BRONCHO TOWER: A MODIFICATION OF THE TOWER OF HANOI PUZZLE

STEPHEN B. GREGG AND JUAN OROZCO

Appendix

The Iterative C++ Program

```
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;


//Stores the starting peg and destination peg for a move
class MovePair{
public:
     int start, dest;
     MovePair(int x, int y);
     MovePair();
};


MovePair::MovePair(int x, int y){
     start = x;
     dest = y;
}


MovePair::MovePair(){}


//Class to store a state of the puzzle
class State{
public:
     vector<int> pegs[3];
     int min; //Min number of moves to reach the state
     int parent; // Parent state
     bool explored; //Have the states following the state been explored
     MovePair prev_move;  //The move taken to reach the parent state
     void new_state(State& state,const int& move_num);
     State();
};
```

```
//Makes the State a copy of another state
void State::new_state(State& state, const int& move_num){
    min = move_num;

    for (int i = 0; i < 3; i++){
        for (int j = 0; j < state.pegs[i].size(); j++){
            pegs[i].push_back(state.pegs[i][j]);
        }
    }
    return;
}


State::State(){
     min = 0;
     parent = 0;
     explored = 0;
}


//Class to hold all the possible states for a certain move number
class PossibleStates{
public:
     vector<State> states;
     PossibleStates();
};


PossibleStates::PossibleStates(){}


class Peg{
public:
     vector<int> disks;
     Peg();
     Peg(int res);
};


Peg::Peg(){}


Peg::Peg(int res){
     disks.reserve(res);}


class MoveList{
public:
     vector<MovePair> moves;
     void push_back(int x, int y);
```

```
    void push_back(MovePair move_pair);
    void copy(MoveList& master);
    MoveList();
};


void MoveList::push_back(int x, int y){
    moves.push_back(MovePair(x, y));}


void MoveList::push_back(MovePair move_pair){
    moves.push_back(MovePair(move_pair.start, move_pair.dest));}


void MoveList::copy(MoveList& master){
    for (int i = 0; i < master.moves.size(); i++){
        moves.push_back(master.moves[i]);}
}


MoveList::MoveList(){}

//Function declarations
void next_move(vector<PossibleStates>& state_tree, int& move_num);
void iterate_state(State& seed_state, const int& parent, const int& move_num,
    vector<PossibleStates>& state_tree);
bool validate(int& start, int& destination, State& seed_state);
bool compare_states(State& state1, State& state2);
bool check_history(vector<PossibleStates>& state_tree, State& testing_state,
    const int& move_num);
void print_pegs(State& pegs);
bool test_solution(vector<int>& destination);
void store_moves(State& state, MoveList& move_list, vector<PossibleStates>&
    state_tree);
void print_moves(MoveList& move_list);

//Global Variables
int n;  //# of disks
int k;  //# of disks at bottom of stack which must be largest
int solution_min;


//Global Class member
vector<MoveList> solutions_list;


int main(){
    int move_num = 0;

    cout << "Enter the number of disks (n) \n";
    cin >> n;
```

```
    cout << "Enter the value of k: \n";
    cin >> k;

    getchar();  //Clear input stream

    solution_min = int(pow(2, n) - 1);

    vector<PossibleStates> state_tree;        //Each element of this vector
        represents all the states at that move number
    state_tree.push_back(PossibleStates());
    state_tree[0].states.push_back(State());

    //Initial state
    for (int i = 0; i < n; i++){
        state_tree[0].states[0].pegs[0].push_back(i);}

    //Call the main recursive function
    next_move(state_tree, move_num);

    //End of Program
    cout << endl;

    //Output Solutions
    for (int i = 0; i < solutions_list.size(); i++){
        print_moves(solutions_list[i]);
        cout << endl << endl;}

    cout << "Minimum number of moves is:" << solution_min << endl;

   // Wait at the end of the program until the user presses enter
    char c = '\0';
    cout << "Press Enter to exit: ";
    while(c != '\n'){
        c = getchar();
        if(c == '\n'){
            cout << "Exiting";
        }else{
            getchar();
            cout << "Not Enter" << endl;
        }
    }
    return 0;
}

void next_move(vector<PossibleStates>& state_tree, int& move_num){
     move_num++;
     state_tree.push_back(PossibleStates());

     //Take all the possible moves for each state
     for (int i = 0; i < state_tree[move_num - 1].states.size(); i++){
```

```
        iterate_state(state_tree[move_num - 1].states[i], i, move_num,
            state_tree);
    }

    if (move_num < solution_min){
        next_move(state_tree, move_num);}

    return;
}


void iterate_state(State& seed_state, const int& parent, const int& move_num,
    vector<PossibleStates>& state_tree){
    //Loop over the 3 pegs
    for (int start = 0; start < 3; start++){

        //Loop over the two possible moves for each disk
        for (int step = 1; step < 3; step++){

            //Set the destination(peg we are moving to) equal to (starting peg (
                start) + step) mod 3
            int destination = (start + step) % 3; // Current peg + step % 3
                gives destination peg

            //Call validate function to see if this move (start to destination)
                is a valid one
            if (validate(start, destination, seed_state)){

                state_tree[move_num].states.push_back(State());
                state_tree[move_num].states.back().new_state(seed_state,
                    move_num);  //Initialize state to previous state
                state_tree[move_num].states.back().pegs[destination].push_back(
                    state_tree[move_num].states.back().pegs[start].back()); //
                    Put disk being moved on top of destination peg
                state_tree[move_num].states.back().pegs[start].pop_back();  //
                    Remove disk from the peg

                if (check_history(state_tree, state_tree[move_num].states.back()
                    , move_num)){
                    state_tree[move_num].states.back().min = move_num;
                    state_tree[move_num].states.back().parent = parent;
                    state_tree[move_num].states.back().prev_move.start = start;
                    state_tree[move_num].states.back().prev_move.dest =
                        destination;

                    //Test to see if the disks are in formation of the solution
                    if (test_solution(state_tree[move_num].states.back().pegs
                        [2])){
                        solution_min = move_num;
                        solutions_list.push_back(MoveList());
```

```
                        store_moves(state_tree[move_num].states.back(),
                            solutions_list.back(), state_tree);
                    }
                } else{
                    state_tree[move_num].states.pop_back();
                }
            }
        }
    }
    return;
}


bool validate(int& start, int& destination, State& seed_state){
    //Check that the peg is not empty
    if (seed_state.pegs[start].size() == 0){return 0;}

    //Check that you're not moving the same disk that was moved previously
    if (start == seed_state.prev_move.dest){return 0;}

    //Check that top disk is larger than the disk being moved if stack size is
       <= k
    if (seed_state.pegs[destination].size() != 0){
        if (seed_state.pegs[destination].size() <= k){
            if (seed_state.pegs[destination].back() > seed_state.pegs[start].
                back()){return 0;}
        }
        else if (seed_state.pegs[destination].size() > k){   //Check that the
            kth disk is larger than the disk being moved if stack size is > k
            if (seed_state.pegs[destination][k - 1] > seed_state.pegs[start].
                back()){return 0;}
        }
    }

    //Check to see if peg 3 is in order if so don't move from it
    bool ordered = 1;
    if (start == 2){
        for (int j = 0; j < seed_state.pegs[2].size(); j++){
            if (seed_state.pegs[2][j] != j){
                ordered = 0;
                j = seed_state.pegs[2].size();
            }
        }
        if (ordered){return 0;}
    }
    return 1;
}


bool compare_states(State& state1, State& state2){
    for (int i = 0; i < 3; i++){
```

```
        if (state1.pegs[i].size() != state2.pegs[i].size()){ return 0;}

        for (int j = 0; j < state1.pegs[i].size(); j++){
            if(state1.pegs[i][j] != state2.pegs[i][j]){
                return 0;
            }
        }
    }
    return 1;
}


//Check whether the state has been achieved in fewer moves
bool check_history(vector<PossibleStates>& state_tree, State& testing_state,
    const int& move_num){
    for (int j = state_tree.size() - 2; j >= 0; j--){
        for (int i = 0; i < state_tree[j].states.size(); i++){
            if (compare_states(state_tree[j].states[i], testing_state)){
                if (state_tree[j].states[i].min < move_num){
                    return 0;}
            }
        }
    }
    return 1;
}


//Print the current pegs state
void print_pegs(State& state){
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < state.pegs[i].size(); j++){
            cout << state.pegs[i][j] << " ";
        }
        cout << "Size:" << state.pegs[i].size() << endl;
    }
    cout << endl;
    return;
}


bool test_solution(vector<int>& destination){
    if (destination.size() != n){ return 0;}

    for (int i = 0; i < n; i++){
        if (destination[i] != i){
            return 0;
        }
    }
    return 1;
}
```

```
void store_moves(State& state, MoveList& move_list, vector<PossibleStates>&
    state_tree){
    move_list.push_back(state.prev_move);
    int temp_parent = state.parent;
    //Place the moves for a given state on the move_list in reverse order
    for (int i = state.min; i > 1; i--){
        move_list.push_back(state_tree[i - 1].states[temp_parent].prev_move);
        temp_parent = state_tree[i - 1].states[temp_parent].parent;
    }
    return;
}


void print_moves(MoveList& move_list){
    for (int i = (move_list.moves.size() - 1); i >= 0; i--){
        cout << "(" << move_list.moves[i].start << ", " << move_list.moves[i].
            dest << "), " ;}
    return;
}
```