# POUR WATER DFS

#include <malloc.h>

#include <stdio.h>

#define tank\_CapacityX 9

#define tank\_CapacityY 4

#define empty 0

#define goal 6

#define maxLength 100

const char \*action[] = {"First State", "pour Water Full X", "pour Water Full Y", "pour Water Empty X", "pour Water empty Y", "pour Water X to Y", "pour Water Y to X"};

typedef struct State {

    int X, Y;

} State;

void makeNULLState(State \*S) {

    S->X = 0;

    S->Y = 0;

}

void printState(State S) {

    printf("\n X: %d\t---\tY: %d\n", S.X, S.Y);

}

int goalCheck(State S) {

    return (S.X == goal || S.Y == goal);

}

int max(int a, int b) {

    return a > b ? a : b;

}

int min(int a, int b) {

    return a < b ? a : b;

}

int pourWaterFullX(State cur\_S, State \*result) {

    if (cur\_S.X < tank\_CapacityX) {

        result->X = tank\_CapacityX;

        result->Y = cur\_S.Y;

        return 1;

    }

    return 0;

}

int pourWaterFullY(State cur\_S, State \*result) {

    if (cur\_S.Y < tank\_CapacityY) {

        result->Y = tank\_CapacityY;

        result->X = cur\_S.X;

        return 1;

    }

    return 0;

}

int pourWaterEmptyX(State cur\_S, State \*result) {

    if (cur\_S.X == empty)

        return 0;

    result->X = empty;

    result->Y = cur\_S.Y;

    return 1;

}

int pourWaterEmptyY(State cur\_S, State \*result) {

    if (cur\_S.Y == empty)

        return 0;

    result->Y = empty;

    result->X = cur\_S.X;

    return 1;

}

int pourWaterXY(State cur\_S, State \*result) {

    if (cur\_S.X <= 0 || cur\_S.Y >= tank\_CapacityY)

        return 0;

    result->X = max(cur\_S.X - (tank\_CapacityY - cur\_S.Y), empty);

    result->Y = min(cur\_S.Y + cur\_S.X, tank\_CapacityY);

    return 1;

}

int pourWaterYX(State cur\_S, State \*result) {

    if (cur\_S.Y <= 0 || cur\_S.X >= tank\_CapacityX)

        return 0;

    result->Y = max(cur\_S.Y - (tank\_CapacityX - cur\_S.X), empty);

    result->X = min(cur\_S.Y + cur\_S.X, tank\_CapacityX);

    return 1;

}

int call\_operator(State cur\_S, State \*result, int option) {

    switch (option) {

        case 1:

            return pourWaterFullX(cur\_S, result);

        case 2:

            return pourWaterFullY(cur\_S, result);

        case 3:

            return pourWaterEmptyX(cur\_S, result);

        case 4:

            return pourWaterEmptyY(cur\_S, result);

        case 5:

            return pourWaterXY(cur\_S, result);

        case 6:

            return pourWaterYX(cur\_S, result);

        default:

            printf("Error calls operators\n");

            return 0;

    }

}

typedef struct Node {

    State state;

    struct Node \*parent;

    int no\_Function;

} Node;

typedef struct Stack {

    Node \*Elements[maxLength];

    int top\_Idx;

} Stack;

void makeNULLStack(Stack \*stack) {

    stack->top\_Idx = maxLength;

}

int emptyStack(Stack stack) {

    return stack.top\_Idx == maxLength;

}

int fullStack(Stack stack) {

    return stack.top\_Idx == 0;

}

Node \*top(Stack stack) {

    if (!emptyStack(stack))

        return stack.Elements[stack.top\_Idx];

    return NULL;

}

void pop(Stack \*stack) {

    if (!emptyStack(\*stack))

        stack->top\_Idx += 1;

    else

        printf("Error! Stack is empty");

}

void push(Node \*X, Stack \*stack) {

    if (fullStack(\*stack))

        printf("Error! Stack is full");

    else {

        stack->top\_Idx -= 1;

        stack->Elements[stack->top\_Idx] = X;

    }

}

int compareStates(State S1, State S2) {

    return (S1.X == S2.X && S1.Y == S2.Y);

}

int findState(State state, Stack openStack) {

    while (!emptyStack(openStack)) {

        if (compareStates(top(openStack)->state, state))

            return 1;

        pop(&openStack);

    }

    return 0;

}

Node \*DFS\_Algorithm(State state) {

    Stack OpenDFS, CloseDFS;

    makeNULLStack(&OpenDFS);

    makeNULLStack(&CloseDFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    push(root, &OpenDFS);

    while (!emptyStack(OpenDFS)) {

        Node \*node = top(OpenDFS);

        pop(&OpenDFS);

        push(node, &CloseDFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 1; opt <= 6; opt++) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findState(newState, CloseDFS) || findState(newState, OpenDFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                push(newNode, &OpenDFS);

            }

        }

    }

    return NULL;

}

void printWaysToGetGoal(Node \*node) {

    Stack stackPrint;

    makeNULLStack(&stackPrint);

    while (node->parent != NULL) {

        push(node, &stackPrint);

        node = node->parent;

    }

    push(node, &stackPrint);

    int noAction = 0;

    while (!emptyStack(stackPrint)) {

        printf("\nAction: %d: %s", noAction, action[top(stackPrint)->no\_Function]);

        printState(top(stackPrint)->state);

        pop(&stackPrint);

        noAction++;

    }

}

typedef struct Queue {

    Node \*Element[maxLength];

    int front, rear;

} Queue;

void makeNULLQueue(Queue \*q) {

    q->front = -1;

    q->rear = -1;

}

int emptyQueue(Queue q) {

    return q.front == -1;

}

int fullQueue(Queue q) {

    return ((q.rear - q.front + 1) % maxLength) == 0;

}

Node \*getFront(Queue q) {

    if (emptyQueue(q))

        printf("Queue is empty");

    else

        return q.Element[q.front];

}

void delQueue(Queue \*q) {

    if (!emptyQueue(\*q)) {

        if (q->front == q->rear)

            makeNULLQueue(q);

        else

            q->front = (q->front + 1) % maxLength;

    } else

        printf("Eror, Delete");

}

void pushQueue(Node \*x, Queue \*q) {

    if (!fullQueue(\*q)) {

        if (emptyQueue(\*q))

            q->front = 0;

        q->rear = (q->rear + 1) % maxLength;

        q->Element[q->rear] = x;

    } else

        printf("Error, Push");

}

int compareStates(State S1, State S2) {

    return (S1.X == S2.X && S1.Y == S2.Y);

}

int findState(State state, Queue openQueue) {

    while (!emptyQueue(openQueue)) {

        if (compareStates(getFront(openQueue)->state, state))

            return 1;

        delQueue(&openQueue);

    }

    return 0;

}

Node \*BFS\_Algorithm(State state) {

    Queue OpenBFS, CloseBFS;

    makeNULLQueue(&OpenBFS);

    makeNULLQueue(&CloseBFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    pushQueue(root, &OpenBFS);

    while (!emptyQueue(OpenBFS)) {

        Node \*node = getFront(OpenBFS);

        delQueue(&OpenBFS);

        pushQueue(node, &CloseBFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 1; opt <= 6; opt++) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findState(newState, CloseBFS) || findState(newState, OpenBFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                pushQueue(newNode, &OpenBFS);

            }

        }

    }

    return NULL;

}

int main() {

    State cur\_S = {0, 0};

    Node \*p = DFS\_Algorithm(cur\_S);

    printWaysToGetGoal(p);

    return 0;

}

# TU SI

#include <malloc.h>

#include <stdio.h>

#define EMPTY 0

#define MAXLENGTH 100

#define MAXPERSON 3

const char \*action[] = {"--First State--", "Move 1 Monk:", "Move 1 Demon:", "Move 2 Monk:", "Move 1 Monk and 1 Demon:", "Move 2 Demon:"};

typedef struct State {

    int Monk;

    int Demon;

    char Boat\_Loc;

} State;

void makeNULLState(State \*S) {

    S->Monk = 0;

    S->Demon = 0;

    S->Boat\_Loc = 'A';

}

void changeState(State \*s, int monk, int demon, char boat) {

    s->Monk = monk;

    s->Demon = demon;

    s->Boat\_Loc = boat;

}

void printState(State S) {

    // printf("\nA:%d:%d\tB:%d:%d\tBoat location: %c\n", S.Monk, S.Demon, MAXPERSON - S.Monk, MAXPERSON - S.Demon, S.Boat\_Loc);

    if (S.Boat\_Loc == 'A')

        printf("\nA:%d:%d-Boat-------B:%d:%d\n", S.Monk, S.Demon, MAXPERSON - S.Monk, MAXPERSON - S.Demon);

    if (S.Boat\_Loc == 'B')

        printf("\nA:%d:%d-------Boat-B:%d:%d\n", S.Monk, S.Demon, MAXPERSON - S.Monk, MAXPERSON - S.Demon);

}

int goalCheck(State S) {

    return (S.Monk == 0 && S.Demon == 0 && S.Boat\_Loc == 'B');

}

int max(int a, int b) { return a > b ? a : b; }

int min(int a, int b) { return a < b ? a : b; }

int checkState(State s) {

    if (s.Demon > MAXPERSON || s.Demon < 0 || s.Monk > MAXPERSON || s.Monk < 0)

        return 0;

    if (s.Monk > 0 && s.Monk < s.Demon)

        return 0;

    if ((MAXPERSON - s.Monk) > 0 && (MAXPERSON - s.Monk) < (MAXPERSON - s.Demon))

        return 0;

    return 1;

}

char moveBoat(char c) { return c == 'A' ? 'B' : 'A'; }

int moveOneMonk(State cur\_S, State \*result) {

    if (cur\_S.Boat\_Loc == 'A') {

        cur\_S.Monk--;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    } else {

        cur\_S.Monk++;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    }

}

int moveOneDemon(State cur\_S, State \*result) {

    if (cur\_S.Boat\_Loc == 'A') {

        cur\_S.Demon--;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    } else {

        cur\_S.Demon++;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    }

}

int moveTwoMonks(State cur\_S, State \*result) {

    if (cur\_S.Boat\_Loc == 'A') {

        cur\_S.Monk -= 2;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    } else {

        cur\_S.Monk += 2;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    }

}

int moveMonkAndDemon(State cur\_S, State \*result) {

    if (cur\_S.Boat\_Loc == 'A') {

        cur\_S.Monk--;

        cur\_S.Demon--;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    } else {

        cur\_S.Monk++;

        cur\_S.Demon++;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    }

}

int moveTwoDemon(State cur\_S, State \*result) {

    if (cur\_S.Boat\_Loc == 'A') {

        cur\_S.Demon -= 2;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    } else {

        cur\_S.Demon += 2;

        if (checkState(cur\_S)) {

            result->Demon = cur\_S.Demon;

            result->Monk = cur\_S.Monk;

            result->Boat\_Loc = moveBoat(cur\_S.Boat\_Loc);

            return 1;

        }

        return 0;

    }

}

int call\_operator(State cur\_S, State \*result, int option) {

    switch (option) {

        case 1:

            return moveOneMonk(cur\_S, result);

        case 2:

            return moveOneDemon(cur\_S, result);

        case 3:

            return moveTwoMonks(cur\_S, result);

        case 4:

            return moveMonkAndDemon(cur\_S, result);

        case 5:

            return moveTwoDemon(cur\_S, result);

        default:

            printf("Error calls operators\n");

            return 0;

    }

}

typedef struct Node {

    State state;

    struct Node \*parent;

    int no\_Function;

} Node;

// BFS SECTION//

typedef struct Queue {

    Node \*Element[MAXLENGTH];

    int front, rear;

} Queue;

void makeNULLQueue(Queue \*q) {

    q->front = -1;

    q->rear = -1;

}

int emptyQueue(Queue q) {

    return q.front == -1;

}

int fullQueue(Queue q) {

    return ((q.rear - q.front + 1) % MAXLENGTH) == 0;

}

Node \*getFront(Queue q) {

    if (emptyQueue(q))

        printf("Queue is empty");

    else

        return q.Element[q.front];

}

void delQueue(Queue \*q) {

    if (!emptyQueue(\*q)) {

        if (q->front == q->rear)

            makeNULLQueue(q);

        else

            q->front = (q->front + 1) % MAXLENGTH;

    } else

        printf("Eror, Delete");

}

void pushQueue(Node \*x, Queue \*q) {

    if (!fullQueue(\*q)) {

        if (emptyQueue(\*q))

            q->front = 0;

        q->rear = (q->rear + 1) % MAXLENGTH;

        q->Element[q->rear] = x;

    } else

        printf("Error, Push");

}

int compareStates(State S1, State S2) {

    return (S1.Demon == S2.Demon && S1.Monk == S2.Monk && S1.Boat\_Loc == S2.Boat\_Loc);

}

int findState(State state, Queue openQueue) {

    while (!emptyQueue(openQueue)) {

        if (compareStates(getFront(openQueue)->state, state))

            return 1;

        delQueue(&openQueue);

    }

    return 0;

}

Node \*BFS\_Algorithm(State state) {

    Queue OpenBFS, CloseBFS;

    makeNULLQueue(&OpenBFS);

    makeNULLQueue(&CloseBFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    pushQueue(root, &OpenBFS);

    while (!emptyQueue(OpenBFS)) {

        Node \*node = getFront(OpenBFS);

        delQueue(&OpenBFS);

        pushQueue(node, &CloseBFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 1; opt <= 5; opt++) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findState(newState, CloseBFS) || findState(newState, OpenBFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                pushQueue(newNode, &OpenBFS);

            }

        }

    }

    return NULL;

}

typedef struct Stack {

    Node \*Elements[MAXLENGTH];

    int top\_Idx;

} Stack;

void makeNULLStack(Stack \*stack) {

    stack->top\_Idx = MAXLENGTH;

}

int emptyStack(Stack stack) {

    return stack.top\_Idx == MAXLENGTH;

}

int fullStack(Stack stack) {

    return stack.top\_Idx == 0;

}

Node \*top(Stack stack) {

    if (!emptyStack(stack))

        return stack.Elements[stack.top\_Idx];

    return NULL;

}

void pop(Stack \*stack) {

    if (!emptyStack(\*stack))

        stack->top\_Idx += 1;

    else

        printf("Error! Stack is EMPTY");

}

void push(Node \*X, Stack \*stack) {

    if (fullStack(\*stack))

        printf("Error! Stack is full");

    else {

        stack->top\_Idx -= 1;

        stack->Elements[stack->top\_Idx] = X;

    }

}

// DFS SECTION//

int compareStatesDFS(State S1, State S2) {

    return (S1.Demon == S2.Demon && S1.Monk == S2.Monk && S1.Boat\_Loc == S2.Boat\_Loc);

}

int findStateDFS(State state, Stack openStack) {

    while (!emptyStack(openStack)) {

        if (compareStatesDFS(top(openStack)->state, state))

            return 1;

        pop(&openStack);

    }

    return 0;

}

Node \*DFS\_Algorithm(State state) {

    Stack OpenDFS, CloseDFS;

    makeNULLStack(&OpenDFS);

    makeNULLStack(&CloseDFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    push(root, &OpenDFS);

    while (!emptyStack(OpenDFS)) {

        Node \*node = top(OpenDFS);

        pop(&OpenDFS);

        push(node, &CloseDFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 1; opt <= 5; opt++) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findStateDFS(newState, CloseDFS) || findStateDFS(newState, OpenDFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                push(newNode, &OpenDFS);

            }

        }

    }

    return NULL;

}

void printWaysToGetGoal(Node \*node) {

    Stack stackPrint;

    makeNULLStack(&stackPrint);

    while (node->parent != NULL) {

        push(node, &stackPrint);

        node = node->parent;

    }

    push(node, &stackPrint);

    int noAction = 0;

    while (!emptyStack(stackPrint)) {

        printf("Action: %d:\n%s", noAction, action[top(stackPrint)->no\_Function]);

        printState(top(stackPrint)->state);

        printf("\n");

        pop(&stackPrint);

        noAction++;

    }

}

int main() {

    State cur\_S = {3, 3, 'A'};

    Node \*p = DFS\_Algorithm(cur\_S);

    printf("LOCATION: (MONK : DEMON)\n");

    printWaysToGetGoal(p);

    return 0;

}

# UP LY

#include <malloc.h>

#include <stdio.h>

#define EMPTY 0

#define MAXLENGTH 100

#define MAXGLASSES 6

#define GLASSESPERMOVE 3

#define UP 1

#define DOWN 0

// const char \*action[] = {"--First State--", "Move 1 Monk:", "Move 1 Demon:", "Move 2 Monk:", "Move 1 Monk and 1 Demon:", "Move 2 Demon:"};

typedef struct State {

    int arrGlasses[6];

} State;

void makeNULLState(State \*s) {

    int i;

    for (i = 0; i < 6; i++)

        s->arrGlasses[i] = !(i % 2);  // 1 = UP // 0 = DOWN

}

void changeState(State \*s, int a, int b, int c, int x, int y, int z) {

    s->arrGlasses[0] = a;

    s->arrGlasses[1] = b;

    s->arrGlasses[2] = c;

    s->arrGlasses[3] = x;

    s->arrGlasses[4] = y;

    s->arrGlasses[5] = z;

}

void printState(State s) {

    int i;

    for (i = 0; i < 6; i++) {

        if (s.arrGlasses[i] == DOWN)

            printf("DOWN\t");

        else

            printf("UP\t");

    }

    printf("\n");

}

int goalCheck(State s) {

    int i;

    for (i = 0; i < 6; i++)

        if (s.arrGlasses[i] == DOWN)

            return 0;

    return 1;

}

int max(int a, int b) { return a > b ? a : b; }

int min(int a, int b) { return a < b ? a : b; }

int flipGlass(State cur\_S, State \*result, int start\_Pos) {

    if (start\_Pos < 1 || start\_Pos > (MAXGLASSES - GLASSESPERMOVE + 1))

        return 0;

    int i;

    \*result = cur\_S;

    for (i = start\_Pos - 1; i < start\_Pos - 1 + GLASSESPERMOVE; i++) {

        result->arrGlasses[i] = !cur\_S.arrGlasses[i];

    }

    return 1;

}

int call\_operator(State cur\_S, State \*result, int option) {

    return flipGlass(cur\_S, result, option);

}

typedef struct Node {

    State state;

    struct Node \*parent;

    int no\_Function;

} Node;

typedef struct Stack {

    Node \*Elements[MAXLENGTH];

    int top\_Idx;

} Stack;

void makeNULLStack(Stack \*stack) {

    stack->top\_Idx = MAXLENGTH;

}

int emptyStack(Stack stack) {

    return stack.top\_Idx == MAXLENGTH;

}

int fullStack(Stack stack) {

    return stack.top\_Idx == 0;

}

Node \*top(Stack stack) {

    if (!emptyStack(stack))

        return stack.Elements[stack.top\_Idx];

    return NULL;

}

void pop(Stack \*stack) {

    if (!emptyStack(\*stack))

        stack->top\_Idx += 1;

    else

        printf("Error! Stack is EMPTY");

}

void push(Node \*X, Stack \*stack) {

    if (fullStack(\*stack))

        printf("Error! Stack is full");

    else {

        stack->top\_Idx -= 1;

        stack->Elements[stack->top\_Idx] = X;

    }

}

typedef struct Queue {

    Node \*Element[MAXLENGTH];

    int front, rear;

} Queue;

void makeNULLQueue(Queue \*q) {

    q->front = -1;

    q->rear = -1;

}

int emptyQueue(Queue q) {

    return q.front == -1;

}

int fullQueue(Queue q) {

    return ((q.rear - q.front + 1) % MAXLENGTH) == 0;

}

Node \*getFront(Queue q) {

    if (emptyQueue(q))

        printf("Queue is empty");

    else

        return q.Element[q.front];

}

void delQueue(Queue \*q) {

    if (!emptyQueue(\*q)) {

        if (q->front == q->rear)

            makeNULLQueue(q);

        else

            q->front = (q->front + 1) % MAXLENGTH;

    } else

        printf("Eror, Delete");

}

void pushQueue(Node \*x, Queue \*q) {

    if (!fullQueue(\*q)) {

        if (emptyQueue(\*q))

            q->front = 0;

        q->rear = (q->rear + 1) % MAXLENGTH;

        q->Element[q->rear] = x;

    } else

        printf("Error, Push");

}

int compareStates(State S1, State S2) {

    int i;

    for (i = 0; i < MAXGLASSES; i++) {

        if (S1.arrGlasses[i] != S2.arrGlasses[i])

            return 0;

    }

    return 1;

}

int findState(State state, Queue openQueue) {

    while (!emptyQueue(openQueue)) {

        if (compareStates(getFront(openQueue)->state, state))

            return 1;

        delQueue(&openQueue);

    }

    return 0;

}

Node \*BFS\_Algorithm(State state) {

    Queue OpenBFS, CloseBFS;

    makeNULLQueue(&OpenBFS);

    makeNULLQueue(&CloseBFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    pushQueue(root, &OpenBFS);

    while (!emptyQueue(OpenBFS)) {

        Node \*node = getFront(OpenBFS);

        delQueue(&OpenBFS);

        pushQueue(node, &CloseBFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 4; opt >= 1; opt--) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findState(newState, CloseBFS) || findState(newState, OpenBFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                pushQueue(newNode, &OpenBFS);

            }

        }

    }

    return NULL;

}

int findStateDFS(State state, Stack openStack) {

    while (!emptyStack(openStack)) {

        if (compareStates(top(openStack)->state, state))

            return 1;

        pop(&openStack);

    }

    return 0;

}

Node \*DFS\_Algorithm(State state) {

    Stack OpenDFS, CloseDFS;

    makeNULLStack(&OpenDFS);

    makeNULLStack(&CloseDFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_Function = 0;

    push(root, &OpenDFS);

    while (!emptyStack(OpenDFS)) {

        Node \*node = top(OpenDFS);

        pop(&OpenDFS);

        push(node, &CloseDFS);

        if (goalCheck(node->state))

            return node;

        int opt;

        for (opt = 4; opt >= 1; opt--) {

            State newState;

            makeNULLState(&newState);

            if (call\_operator(node->state, &newState, opt)) {

                if (findStateDFS(newState, CloseDFS) || findStateDFS(newState, OpenDFS))

                    continue;

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_Function = opt;

                push(newNode, &OpenDFS);

            }

        }

    }

    return NULL;

}

void printWaysToGetGoal(Node \*node) {

    Stack stackPrint;

    makeNULLStack(&stackPrint);

    while (node->parent != NULL) {

        push(node, &stackPrint);

        node = node->parent;

    }

    push(node, &stackPrint);

    int noAction = 0;

    while (!emptyStack(stackPrint)) {

        printf("Action: %d: \n", noAction);

        printState(top(stackPrint)->state);

        printf("\n");

        pop(&stackPrint);

        noAction++;

    }

}

int main() {

    State cur\_S;

    makeNULLState(&cur\_S);

    Node \*p = DFS\_Algorithm(cur\_S);

    printWaysToGetGoal(p);

    return 0;

}

# BFS Heuristic and A-Star

#include <stdio.h>

#include <stdlib.h>

#define ROWS 3

#define COLS 3

#define EMPTY 0

#define MAX\_OPERATOR 4

#define MAXLENGTH 500

const char \*action[] = {"First State", "Move cell EMPTY to UP", "Move cell EMPTY to DOWN", "Move cell EMPTY to LEFT", "Move cell EMPTY to RIGHT"};

typedef struct State {

    int eightPuzzel[ROWS][COLS];

    int emptyRow;

    int emptyCol;

} State;

void printState(State state) {

    int row, col;

    printf("\n--------\n");

    for (row = 0; row < ROWS; row++) {

        for (col = 0; col < COLS; col++)

            printf("|%d ", state.eightPuzzel[row][col]);

        printf("|\n");

    }

    printf("--------\n");

}

int compareStates(State state1, State state2) {

    if (state1.emptyRow != state2.emptyRow || state1.emptyCol != state2.emptyCol)

        return 0;

    int row, col;

    for (row = 0; row < ROWS; row++)

        for (col = 0; col < COLS; col++)

            if (state1.eightPuzzel[row][col] != state2.eightPuzzel[row][col])

                return 0;

    return 1;

}

int goalCheck(State state, State goal) {

    return compareStates(state, goal);

}

int upOperator(State state, State \*result) {

    \*result = state;

    int empRowCurrent = state.emptyRow, empColCurrent = state.emptyCol;

    if (empRowCurrent > 0) {

        result->emptyRow = empRowCurrent - 1;

        result->emptyCol = empColCurrent;

        result->eightPuzzel[empRowCurrent][empColCurrent] = state.eightPuzzel[empRowCurrent - 1][empColCurrent];

        result->eightPuzzel[empRowCurrent - 1][empColCurrent] = EMPTY;

        return 1;

    }

    return 0;

}

int downOperator(State state, State \*result) {

    \*result = state;

    int empRowCurrent = state.emptyRow, empColCurrent = state.emptyCol;

    if (empRowCurrent < ROWS - 1) {

        result->emptyRow = empRowCurrent + 1;

        result->emptyCol = empColCurrent;

        result->eightPuzzel[empRowCurrent][empColCurrent] = state.eightPuzzel[empRowCurrent + 1][empColCurrent];

        result->eightPuzzel[empRowCurrent + 1][empColCurrent] = EMPTY;

        return 1;

    }

    return 0;

}

int leftOperator(State state, State \*result) {

    \*result = state;

    int empRowCurrent = state.emptyRow, empColCurrent = state.emptyCol;

    if (empColCurrent > 0) {

        result->emptyRow = empRowCurrent;

        result->emptyCol = empColCurrent - 1;

        result->eightPuzzel[empRowCurrent][empColCurrent] = state.eightPuzzel[empRowCurrent][empColCurrent - 1];

        result->eightPuzzel[empRowCurrent][empColCurrent - 1] = EMPTY;

        return 1;

    }

    return 0;

}

int rightOperator(State state, State \*result) {

    \*result = state;

    int empRowCurrent = state.emptyRow, empColCurrent = state.emptyCol;

    if (empColCurrent < COLS - 1) {

        result->emptyRow = empRowCurrent;

        result->emptyCol = empColCurrent + 1;

        result->eightPuzzel[empRowCurrent][empColCurrent] = state.eightPuzzel[empRowCurrent][empColCurrent + 1];

        result->eightPuzzel[empRowCurrent][empColCurrent + 1] = EMPTY;

        return 1;

    }

    return 0;

}

int callOperators(State state, State \*result, int opt) {

    switch (opt) {

        case 1:

            return upOperator(state, result);

        case 2:

            return downOperator(state, result);

        case 3:

            return leftOperator(state, result);

        case 4:

            return rightOperator(state, result);

        default:

            printf("Cannot call operators");

            return 0;

    }

}

int heuristicOne(State state, State goal) {

    int row, col, count = 0;

    for (row = 0; row < ROWS; row++)

        for (col = 0; col < COLS; col++)

            if (state.eightPuzzel[row][col] != goal.eightPuzzel[row][col])

                count++;

    return count;

}

typedef struct Node {

    State state;

    struct Node \*parent;

    int no\_function;

    int heuristic;

    // A\_STAR

    //  int f;

    //  int g;

    //  int h;

} Node;

typedef struct List {

    Node \*Elements[MAXLENGTH];

    int size;

} List;

void makeNULLList(List \*list) {

    list->size = 0;

}

int emptyList(List list) {

    return list.size == 0;

}

int fullList(List list) {

    return list.size == MAXLENGTH;

}

Node \*elementAt(int p, List list) {

    return list.Elements[p - 1];

}

void pushList(Node \*x, int position, List \*list) {

    if (!fullList(\*list)) {

        int q;

        for (q = list->size; q >= position; q--)

            list->Elements[q] = list->Elements[q - 1];

        list->Elements[position - 1] = x;

        list->size++;

    } else

        printf("List is full\n");

}

void deleteList(int position, List \*list) {

    if (emptyList(\*list))

        printf("List is empty\n");

    else if (position < 1 || position > list->size)

        printf("Position is not possible to delete\n");

    else {

        int i;

        for (i = position - 1; i < list->size; i++)

            list->Elements[i] = list->Elements[i + 1];

        list->size--;

    }

}

Node \*findState(State state, List list, int \*position) {

    int i;

    for (i = 1; i <= list.size; i++)

        if (compareStates(elementAt(i, list)->state, state)) {

            \*position = i;

            return elementAt(i, list);

        }

    return NULL;

}

void sortList(List \*list) {

    int i, j;

    for (i = 0; i < list->size - 1; i++)

        for (j = i + 1; j < list->size; j++)

            if (list->Elements[i]->heuristic > list->Elements[j]->heuristic) {

                Node \*node = list->Elements[i];

                list->Elements[i] = list->Elements[j];

                list->Elements[j] = node;

            }

}

Node \*bestFirstSearch(State state, State goal) {

    List OpenBFS, CloseBFS;

    makeNULLList(&OpenBFS);

    makeNULLList(&CloseBFS);

    Node \*root = (Node \*)malloc(sizeof(Node));

    root->state = state;

    root->parent = NULL;

    root->no\_function = 0;

    root->heuristic = heuristicOne(root->state, goal);

    pushList(root, OpenBFS.size + 1, &OpenBFS);

    while (!emptyList(OpenBFS)) {

        Node \*node = elementAt(1, OpenBFS);

        deleteList(1, &OpenBFS);

        pushList(node, CloseBFS.size + 1, &CloseBFS);

        if (goalCheck(node->state, goal))

            return node;

        int opt;

        for (opt = 1; opt <= MAX\_OPERATOR; opt++) {

            State newState;

            newState = node->state;

            if (callOperators(node->state, &newState, opt)) {

                Node \*newNode = (Node \*)malloc(sizeof(Node));

                newNode->state = newState;

                newNode->parent = node;

                newNode->no\_function = opt;

                newNode->heuristic = heuristicOne(newState, goal);

                int posOpen, posClose;

                Node \*nodeFoundOpen = findState(newState, OpenBFS, &posOpen);

                Node \*nodeFoundClose = findState(newState, CloseBFS, &posClose);

                if (nodeFoundOpen == NULL && nodeFoundClose == NULL) {

                    pushList(newNode, OpenBFS.size + 1, &OpenBFS);

                } else if (nodeFoundOpen != NULL && nodeFoundOpen->heuristic > newNode->heuristic) {

                    deleteList(posOpen, &OpenBFS);

                    pushList(newNode, posOpen, &OpenBFS);

                } else if (nodeFoundClose != NULL && nodeFoundClose->heuristic > newNode->heuristic) {

                    deleteList(posClose, &CloseBFS);

                    pushList(newNode, OpenBFS.size + 1, &OpenBFS);

                }

                sortList(&OpenBFS);

            }

        }

    }

    return NULL;

}

// Node \*A\_Star(State state, State goal) {

//     List Open\_A\_Star, Close\_A\_Star;

//     makeNULLList(&Open\_A\_Star);

//     makeNULLList(&Close\_A\_Star);

//     Node \*root = (Node \*)malloc(sizeof(Node));

//     root->state = state;

//     root->parent = NULL;

//     root->no\_function = 0;

//     root->g = 0;

//     root->h = heuristicOne(root->state, goal);

//     root->f = root->g + root->h;

//     pushList(root, Open\_A\_Star.size + 1, &Open\_A\_Star);

//     while (!emptyList(Open\_A\_Star)) {

//         Node \*node = elementAt(1, Open\_A\_Star);

//         deleteList(1, &Open\_A\_Star);

//         pushList(node, Close\_A\_Star.size + 1, &Close\_A\_Star);

//         if (goalCheck(node->state, goal))

//             return node;

//         int opt;

//         for (opt = 1; opt <= MAX\_OPERATOR; opt++) {

//             State newstate;

//             if (callOperators(node->state, &newstate, opt)) {

//                 Node \*newNode = (Node \*)malloc(sizeof(Node));

//                 newNode->state = newstate;

//                 newNode->parent = node;

//                 newNode->no\_function = opt;

//                 newNode->g = node->g + 1;

//                 newNode->h = heuristicOne(newstate, goal);

//                 newNode->f = newNode->g + newNode->h;

//                 int posOpen, posClose;

//                 Node \*nodeFoundOpen = findState(newstate, Open\_A\_Star, &posOpen);

//                 Node \*nodeFoundClose = findState(newstate, Close\_A\_Star, &posClose);

//                 if (nodeFoundOpen == NULL && nodeFoundClose == NULL) {

//                     pushList(newNode, Open\_A\_Star.size + 1, &Open\_A\_Star);

//                 } else if (nodeFoundOpen != NULL && nodeFoundOpen->g > newNode->g) {

//                     deleteList(posOpen, &Open\_A\_Star);

//                     pushList(newNode, posOpen, &Open\_A\_Star);

//                 } else if (nodeFoundClose != NULL && nodeFoundClose->g > newNode->g) {

//                     deleteList(posClose, &Close\_A\_Star);

//                     pushList(newNode, Open\_A\_Star.size + 1, &Open\_A\_Star);

//                 }

//                 sortList(&Open\_A\_Star);

//             }

//         }

//     }

//     return NULL;

// }

void printWaysToGetGoal(Node \*node) {

    List listPrint;

    makeNULLList(&listPrint);

    while (node->parent != NULL) {

        pushList(node, listPrint.size + 1, &listPrint);

        node = node->parent;

    }

    pushList(node, listPrint.size + 1, &listPrint);

    int no\_action = 0, i;

    for (i = listPrint.size; i > 0; i--) {

        printf("\nAction %d: %s", no\_action, action[elementAt(i, listPrint)->no\_function]);

        printState(elementAt(i, listPrint)->state);

        no\_action++;

    }

}

int main() {

    State state, goal;

    // State state, goal, result;

    state.emptyCol = 1;

    state.emptyRow = 1;

    state.eightPuzzel[0][0] = 3;

    state.eightPuzzel[0][1] = 4;

    state.eightPuzzel[0][2] = 5;

    state.eightPuzzel[1][0] = 1;

    state.eightPuzzel[1][1] = 0;

    state.eightPuzzel[1][2] = 2;

    state.eightPuzzel[2][0] = 6;

    state.eightPuzzel[2][1] = 7;

    state.eightPuzzel[2][2] = 8;

    goal.emptyCol = 0;

    goal.emptyRow = 0;

    goal.eightPuzzel[0][0] = 0;

    goal.eightPuzzel[0][1] = 1;

    goal.eightPuzzel[0][2] = 2;

    goal.eightPuzzel[1][0] = 3;

    goal.eightPuzzel[1][1] = 4;

    goal.eightPuzzel[1][2] = 5;

    goal.eightPuzzel[2][0] = 6;

    goal.eightPuzzel[2][1] = 7;

    goal.eightPuzzel[2][2] = 8;

    Node \*p = bestFirstSearch(state, goal);

    printWaysToGetGoal(p);

    // Node \*p = A\_Star(state, goal);

    // if (p == NULL)

    //     printf("Can't find the answer!!\n");

    // else

    //     printWaysToGetGoal(p);

    return 0;

}

# KEN KEN

#include <stdio.h>

#include <stdlib.h>

#define MAX\_LENGTH 100

#define NB\_ROWS 4

#define NB\_COLUMNS 4

#define MAX\_VALUE 5

#define EMPTY 0

#define INF 999999

int NB\_BLOCK = 8;

typedef struct List {

    int data[MAX\_LENGTH];

    int size;

} List;

void makeNULLList(List \*list) {

    list->size = 0;

}

int emptyList(List list) {

    return list.size == EMPTY;

}

int fullList(List list) {

    return list.size == MAX\_LENGTH;

}

int elementAt(List list, int p) {

    return list.data[p - 1];

}

void pushList(List \*list, int n) {

    list->data[list->size] = n;

    list->size++;

}

typedef struct Coord {

    int x, y;

} Coord;

typedef struct ListCoord {

    Coord data[MAX\_LENGTH];

    int size;

} ListCoord;

void initListCoord(ListCoord \*list) {

    list->size = 0;

}

void appendListCoord(ListCoord \*list, Coord coord) {

    list->data[list->size++] = coord;

}

typedef struct Constrains {

    int data[NB\_ROWS \* NB\_COLUMNS][NB\_ROWS \* NB\_COLUMNS];

    int n;

} Constrains;

void initConstrains(Constrains \*constrains) {

    int i, j;

    for (i = 0; i < NB\_ROWS \* NB\_COLUMNS; i++)

        for (j = 0; j < NB\_ROWS \* NB\_COLUMNS; j++)

            constrains->data[i][j] = 0;

    constrains->n = NB\_ROWS \* NB\_COLUMNS;

}

int indexOf(Coord coord) {

    return NB\_ROWS \* coord.x + coord.y;

}

Coord positionOfVertex(int vertex) {

    Coord coord;

    coord.x = vertex / NB\_ROWS;

    coord.y = vertex % NB\_COLUMNS;

    return coord;

}

int addConstrain(Constrains \*constrains, Coord source, Coord target) {

    int u = indexOf(source);

    int v = indexOf(target);

    if (constrains->data[u][v] == 0) {

        constrains->data[u][v] = 1;

        constrains->data[v][u] = 1;

        return 1;

    }

    return 0;

}

ListCoord getConstrains(Constrains constrains, Coord coord) {

    int i;

    int v = indexOf(coord);

    ListCoord result;

    initListCoord(&result);

    for (i = 0; i < constrains.n; i++) {

        if (constrains.data[v][i] == 1) {

            appendListCoord(&result, positionOfVertex(i));

        }

    }

    return result;

}

typedef struct Block {

    int value;

    int currentValue;

    char operator;

    ListCoord list;

    int filledCells;

} Block;

void initBlock(Block \*block, int sum, char c, ListCoord list) {

    block->value = sum;

    block->currentValue = 0;

    block->filledCells = 0;

    block->operator= c;

    int i;

    initListCoord(&block->list);

    for (i = 0; i < list.size; i++)

        appendListCoord(&block->list, list.data[i]);

}

int whichBlockAreYouIn(Coord position, Block block[]) {

    int i, j;

    for (i = 0; i < NB\_BLOCK; i++) {

        for (j = 0; j < block[i].list.size; j++) {

            if (position.x == block[i].list.data[j].x && position.y == block[i].list.data[j].y)

                return i;

        }

    }

}

int blockIsFilled(Block block) {

    return block.filledCells == block.list.size;

}

typedef struct KenKen {

    int cells[NB\_ROWS][NB\_COLUMNS];

    Constrains constrains;

    Block block[10];

} KenKen;

void initKenKen(KenKen \*kenken) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            kenken->cells[i][j] = EMPTY;

    initConstrains(&kenken->constrains);

}

void printKenKen(KenKen kenken) {

    int i, j;

    printf("KenKen:\n");

    for (i = 0; i < NB\_ROWS; i++) {

        printf("-------------------------\n");

        for (j = 0; j < NB\_COLUMNS; j++) {

            printf("|%3d  ", kenken.cells[i][j]);

        }

        printf("|\n");

    }

    printf("------------------------\n");

    printf("KenKen Operator:\n");

    for (i = 0; i < NB\_ROWS; i++) {

        printf("-----------------\n");

        for (j = 0; j < NB\_COLUMNS; j++) {

            int k = whichBlockAreYouIn((Coord){i, j}, kenken.block);

            if (kenken.block[k].operator== '.')

                printf("|%3d  ", kenken.block[k].value);

            else

                printf("|%3d%c ", kenken.block[k].value, kenken.block[k].operator);

        }

        printf("|\n");

    }

    printf("-------------------------\n");

}

int isFilledKenKen(KenKen kenken) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            if (kenken.cells[i][j] == EMPTY)

                return 0;

    return 1;

}

void spreadConstrainsForm(Coord position, Constrains \*constrains) {

    int row = position.x, column = position.y;

    int i, j;

    for (i = 0; i < NB\_ROWS; i++) {

        if (i != row) {

            Coord pos = {i, column};

            addConstrain(constrains, position, pos);

        }

    }

    for (i = 0; i < NB\_COLUMNS; i++) {

        if (i != column) {

            Coord pos = {row, i};

            addConstrain(constrains, position, pos);

        }

    }

}

void resetConstrains(Constrains \*constrains, int cells[NB\_ROWS][NB\_COLUMNS]) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            constrains->data[i][j] = 0;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            if (cells[i][j] != 0)

                spreadConstrainsForm((Coord){i, j}, constrains);

}

List getAvailableValues(Coord position, KenKen kenken) {

    ListCoord posList = getConstrains(kenken.constrains, position);

    int availables[MAX\_VALUE];

    int i;

    for (i = 1; i < MAX\_VALUE; i++)

        availables[i] = 1;

    for (i = 0; i < posList.size; i++) {

        Coord pos = posList.data[i];

        if (kenken.cells[pos.x][pos.y] != EMPTY) {

            availables[kenken.cells[pos.x][pos.y]] = 0;

        }

    }

    List result;

    makeNULLList(&result);

    for (i = 1; i < MAX\_VALUE; i++) {

        if (availables[i])

            pushList(&result, i);

    }

    return result;

}

Coord getNextEmptyCell(KenKen kenken) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++) {

        for (j = 0; j < NB\_COLUMNS; j++) {

            Coord pos = {i, j};

            if (kenken.cells[i][j] == EMPTY)

                return pos;

        }

    }

}

int resultFromOperatorAndTwoValues(int a, int b, char c) {

    if (c == '+')

        return a + b;

    if (c == '-')

        if (a > b)

            return a - b;

        else

            return b - a;

    if (c == '/')

        if (a > b)

            return a / b;

        else

            return b / a;

    if (c == '\*' || c == 'x')

        return a \* b;

    return b;

}

Coord getNextMinDomainCell(KenKen kenken) {

    int minLength = INF;

    int i, j;

    Coord result;

    for (i = 0; i < NB\_ROWS; i++) {

        for (j = 0; j < NB\_COLUMNS; j++) {

            if (kenken.cells[i][j] == EMPTY) {

                Coord pos = {i, j};

                int availablesLength = getAvailableValues(pos, kenken).size;

                if (availablesLength < minLength) {

                    minLength = availablesLength;

                    result = pos;

                }

            }

        }

    }

    return result;

}

int exploredCounter = 0;

int kenkenBackTracking(KenKen \*kenken) {

    if (isFilledKenKen(\*kenken))

        return 1;

    Coord position = getNextEmptyCell(\*kenken);

    // Coord position = getNextMinDomainCell(\*kenken);

    List availables = getAvailableValues(position, \*kenken);

    if (availables.size == 0) {

        return 0;

    }

    int j;

    for (j = 0; j < availables.size; j++) {

        int value = availables.data[j];

        int blockNB = whichBlockAreYouIn(position, kenken->block);

        kenken->cells[position.x][position.y] = value;

        int currentValue = kenken->block[blockNB].currentValue;

        if (kenken->block[blockNB].filledCells == 0) {

            kenken->block[blockNB].filledCells++;

            kenken->block[blockNB].currentValue += value;

            spreadConstrainsForm(position, &kenken->constrains);

        } else {

            kenken->block[blockNB].filledCells++;

            kenken->block[blockNB].currentValue = resultFromOperatorAndTwoValues(kenken->block[blockNB].currentValue, value, kenken->block[blockNB].operator);

            spreadConstrainsForm(position, &kenken->constrains);

        }

        if (blockIsFilled(kenken->block[blockNB]) && (kenken->block[blockNB].currentValue != kenken->block[blockNB].value)) {

            kenken->cells[position.x][position.y] = EMPTY;

            kenken->block[blockNB].filledCells--;

            kenken->block[blockNB].currentValue = currentValue;

            resetConstrains(&kenken->constrains, kenken->cells);

            continue;

            // return 0;

        }

        exploredCounter++;

        if (kenkenBackTracking(kenken))

            return 1;

        kenken->cells[position.x][position.y] = EMPTY;

        kenken->block[blockNB].filledCells--;

        kenken->block[blockNB].currentValue = currentValue;

        resetConstrains(&kenken->constrains, kenken->cells);

    }

    return 0;

}

int solveKenKen(KenKen \*kenken) {

    initConstrains(&kenken->constrains);

    exploredCounter = 0;

    return kenkenBackTracking(kenken);

}

void readKenKen(KenKen \*kenken) {

    int i;

    printf("Number of Cage: ");

    scanf("%d", &NB\_BLOCK);

    for (i = 0; i < NB\_BLOCK; i++) {

        ListCoord cage;

        initListCoord(&cage);

        int k, n, value;

        // printf("Enter value of cage: ");

        scanf("%d", &value);

        char c;

        // printf("Enter operator: ");

        scanf(" %c", &c);

        getchar();

        // printf("Number of cells of cage%d: ", i + 1);

        scanf("%d", &n);

        for (k = 0; k < n; k++) {

            // printf("Enter position(x,y): ");

            int u, v;

            scanf("%d%d", &u, &v);

            appendListCoord(&cage, (Coord){u, v});

        }

        initBlock(&kenken->block[i], value, c, cage);

    }

}

int main() {

    KenKen kenken;

    initKenKen(&kenken);

    int i;

    freopen("test4.txt", "r", stdin);

    readKenKen(&kenken);

    if (solveKenKen(&kenken)) {

        printf("Solved\n");

        printKenKen(kenken);

    } else

        printf("Can not solve\n");

    printf("Explored %d states\n", exploredCounter);

}

# 8 QUEENS

#include <stdio.h>

#include <stdlib.h>

#define NB\_ROWS 8

#define NB\_COLUMNS 8

#define QUEEN 1

#define EMPTY 0

#define INF 999999

typedef struct Coord {

    int x, y;

} Coord;

typedef struct Constrains {

    int data[NB\_ROWS][NB\_COLUMNS];

} Constrains;

void initConstrains(Constrains \*constrains) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_ROWS; j++)

            constrains->data[i][j] = 0;

}

void addConstrain(Constrains \*constrains, int u, int v) {

    constrains->data[u][v] = 1;

}

typedef struct Chess {

    int cells[NB\_ROWS][NB\_COLUMNS];

    int queens;

    Constrains constrains;

} Chess;

void initChess(Chess \*chess) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            chess->cells[i][j] = EMPTY;

    chess->queens = 0;

    initConstrains(&chess->constrains);

}

void printChess(Chess chess) {

    int i, j;

    printf("Chess board:\n");

    for (i = 0; i < NB\_ROWS; i++) {

        for (j = 0; j < NB\_COLUMNS; j++) {

            printf("| ");

            if (chess.cells[i][j] == 0) {

                printf("  ");

            } else

                printf("Q ");

        }

        printf("|\n");

    }

    printf("Queens: %d \n", chess.queens);

    printf("--------------------------------\n");

}

int isFilledChess(Chess chess) {

    return chess.queens == NB\_COLUMNS;

}

void spreadConstrainsForm(Coord position, Constrains \*constrains) {

    int row = position.x, column = position.y;

    int i, j;

    for (i = 0; i < NB\_ROWS; i++) {

        if (i != row)

            addConstrain(constrains, i, column);

    }

    for (i = 0; i < NB\_COLUMNS; i++) {

        if (i != column)

            addConstrain(constrains, row, i);

    }

    for (i = row, j = column; i < NB\_ROWS && j < NB\_COLUMNS; i++, j++) {

        addConstrain(constrains, i, j);

    }

    for (i = row, j = column; i >= 0 && j >= 0; i--, j--) {

        addConstrain(constrains, i, j);

    }

    for (i = row, j = column; i < NB\_ROWS && j >= 0; i++, j--) {

        addConstrain(constrains, i, j);

    }

    for (i = row, j = column; i >= 0 && j < NB\_COLUMNS; i--, j++) {

        addConstrain(constrains, i, j);

    }

}

void resetConstrains(Constrains \*constrains, int cells[NB\_ROWS][NB\_COLUMNS]) {

    int i, j;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            constrains->data[i][j] = 0;

    for (i = 0; i < NB\_ROWS; i++)

        for (j = 0; j < NB\_COLUMNS; j++)

            if (cells[i][j] != 0)

                spreadConstrainsForm((Coord){i, j}, constrains);

}

void printConstrain(Constrains constrains) {

    int i, j;

    printf("Constrain: \n");

    for (i = 0; i < NB\_ROWS; i++) {

        for (j = 0; j < NB\_COLUMNS; j++)

            printf("%d ", constrains.data[i][j]);

        printf("\n");

    }

}

int exploredCounter = 0;

int solveChess(Chess \*chess, int col) {

    if (isFilledChess(\*chess))

        return 1;

    int i;

    for (i = 0; i < NB\_COLUMNS; i++) {

        if (chess->constrains.data[i][col] != 1) {

            chess->cells[i][col] = QUEEN;

            chess->queens++;

            spreadConstrainsForm((Coord){i, col}, &chess->constrains);

            if (solveChess(chess, col + 1))

                return 1;

            chess->cells[i][col] = EMPTY;

            chess->queens--;

            resetConstrains(&chess->constrains, chess->cells);

        }

    }

    return 0;

}

int main() {

    Chess chess;

    initChess(&chess);

    printf("------%d QUEENS PROBLEMS------\n", NB\_ROWS);

    if (solveChess(&chess, 0)) {

        printf("Solved\n");

        printChess(chess);

    } else {

        printf("Can't find solution!!!");

    }

    return 0;

}

# PROLOG

power(\_, 0, 1).

power(X, Y, Z) :- Y > 0,Y1 is Y - 1,power(X, Y1, Z1),Z is X \* Z1.

factorial(0, 1).

factorial(N, Result) :- N > 0,N1 is N - 1,factorial(N1, Result1),Result is N \* Result1.

combination(N, 0, 1).

combination(N, K, Result) :- N >= K,factorial(N, FN),factorial(K, FK),NK is N - K,factorial(NK, FNK),Result is FN / (FK \* FNK).

% combination(5, 2, X).

% combination(15, 4, X).

thempt(L1, X, L2) :- append(L1, [X], L2).

noids([],L,L).

noids([X1|L1],L2,[X1|L3]) :- noids(L1,L2,L3).

ptvtle([], []).

ptvtle([X], [X]).

ptvtle([H, \_ | T], [H | Result]) :- ptvtle(T, Result).

replace(I, [Old|T], [New|T], Old, New) :- I == 0, !.

replace(I, [H|T], [H|T1], Old, New) :- I > 0, I1 is I - 1, replace(I1, T, T1, Old, New).

swap(I, J, L1, L2) :-   replace(I, L1, L3, X, Y),replace(J, L3, L2, Y, X).

list\_sum([],0).

list\_sum([Head|Tail], Sum) :-   list\_sum(Tail,SumTemp),Sum is Head + SumTemp.

list\_sum\_odd(L, X):- ptvtle(L, L1), list\_sum(L1,X).

less\_than\_X([], \_, []).

less\_than\_X([H|T], N, X) :- H < N,less\_than\_X(T, N, Z),X = [H|Z].

less\_than\_X([H|T], N, X) :- H >= N,less\_than\_X(T, N, X).

gcd(0, X, X):-  X > 0, !.

gcd(X, Y, Z):-  X >= Y, X1 is X-Y,gcd(X1,Y,Z).

gcd(X, Y, Z):-  X < Y, X1 is Y-X, gcd(X1,X,Z).

coprime(X,Y):- gcd(X,Y,Z), Z=:=1.

divide(N,P):- P mod N =:= 0.

divide(N,P):- P mod N \= 0,N\*N < P,N1 is N+1, divide(N1,P).

prime(P):- \+divide(2,P).

count(\_, [], 0) :- !.

count(X, [X|T], N) :- count(X, T, N1), N is N1 + 1, !.

count(X, [\_|T], N) :- count(X, T, N).

list\_length([],0).

list\_length([\_|TAIL],N) :- list\_length(TAIL,N1), N is N1 + 1.

list\_concat([],L,L).

list\_concat([X1|L1],L2,[X1|L3]) :- list\_concat(L1,L2,L3).

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

list\_rev([],[]).

list\_rev([Head|Tail],Reversed) :-

list\_rev(Tail, RevTail),list\_concat(RevTail, [Head],Reversed).

set([], []) :- !.

set([H|T], [H|T1]) :- subtract(T, [H], T2), set(T2, T1).

prime\_factor(N, D) :- find\_prime\_factor(N, 2, D).

find\_prime\_factor(N, D, D) :- 0 is N mod D.

find\_prime\_factor(N, D, R) :- D < N,

    (0 is N mod D -> (N1 is N/D, find\_prime\_factor(N1, D, R));

    (D1 is D + 1, find\_prime\_factor(N, D1, R))).

prime\_factors(N, L) :- findall(D, prime\_factor(N, D), L).