Report of Homework Assignment #2

1.

Code for function **is\_goal\_state:**

//Check if the given state is a goal state

//Returns: true if is goal state, false otherwise

function is\_goal\_state(state) {

++helper\_eval\_state\_count; //Keep track of how many states are evaluated (DO NOT REMOVE!)

var temp=state.grid;

var is\_goal\_state=(temp[0][0]==1 && temp[0][1]==2 && temp[0][2]==3 && temp[1][0]==8 && temp[1][1]==0

&& temp[1][2]==4 && temp[2][0]==7 && temp[2][1]==6 && temp[2][2]==5);

return is\_goal\_state/\*\*\*Your code to check for goal state here!\*\*\*/;

}

Code for function **find\_successors:**

//Find the list of actions that can be performed from the given state and the new

//states that result from each of those actions

//Returns: Array of successor objects (where each object has a valid actionID member and corresponding resultState member)

function create\_newstate(state){

let newState={

grid : state.grid.map(x => x.slice(0)) //Deep copy of grid

};

return newState;

}

function find\_successors(state) {

++helper\_expand\_state\_count; //Keep track of how many states are expanded (DO NOT REMOVE!)

let successors=[];

/\*\*\*Your code to generate successors here!\*\*\*/

//Hint: Javascript objects are passed by reference, so don't modify "state" directy.

//Make copies instead:

for(let j=0;j<3;++j)

for(let i=0;i<3;++i)

{

if (state.grid[j][i]==0)

{

k=j-1;

if (k>=0 && k<=2){

let newState=create\_newstate(state);

newState.grid[j][i]=newState.grid[k][i];

newState.grid[k][i]=0;

successors.push({actionID : 1/\*ID\*/,resultState : newState});

}

k=i-1;

if (k>=0 && k<=2){

let newState=create\_newstate(state);

newState.grid[j][i]=newState.grid[j][k];

newState.grid[j][k]=0;

successors.push({actionID : 3/\*ID\*/,resultState : newState});

}

k=j+1;

if (k>=0 && k<=2){

let newState=create\_newstate(state);

newState.grid[j][i]=newState.grid[k][i];

newState.grid[k][i]=0;

successors.push({actionID : 2/\*ID\*/,resultState : newState});

}

k=i+1;

if (k>=0 && k<=2){

let newState=create\_newstate(state);

newState.grid[j][i]=newState.grid[j][k];

newState.grid[j][k]=0;

successors.push({actionID : 4/\*ID\*/,resultState : newState});

}

}

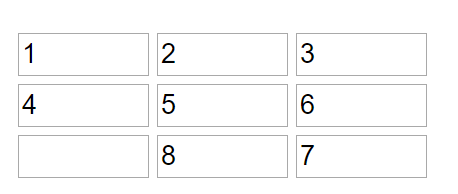
}

return successors;

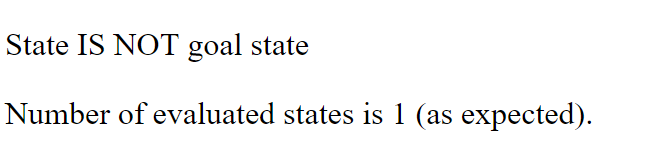
}

2. For the three boar configurations I tried, all of the results match my expectation, including whether the state is goal state and the possible successor state. For board configuration (1) and (3) as we can see, they are not the goal state, while configuration (2) is. And the successor states are also correct.

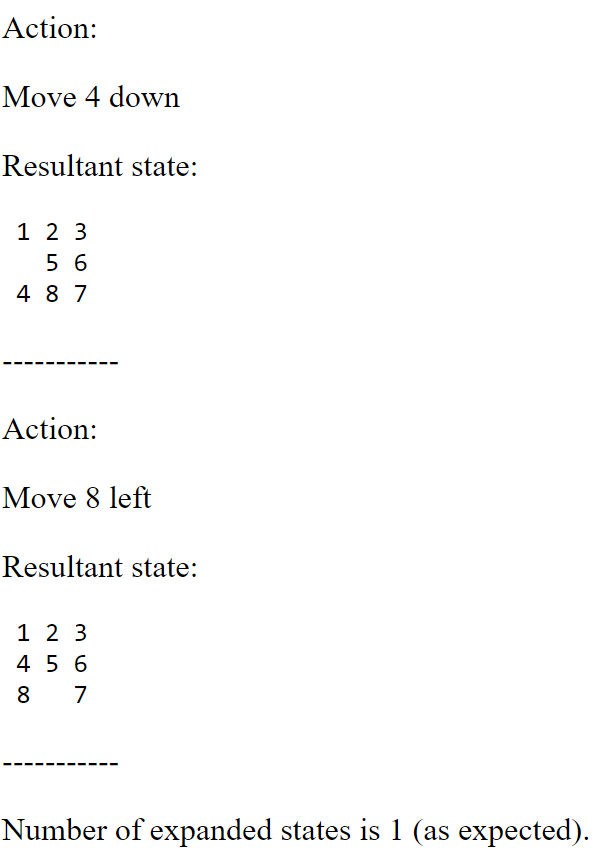
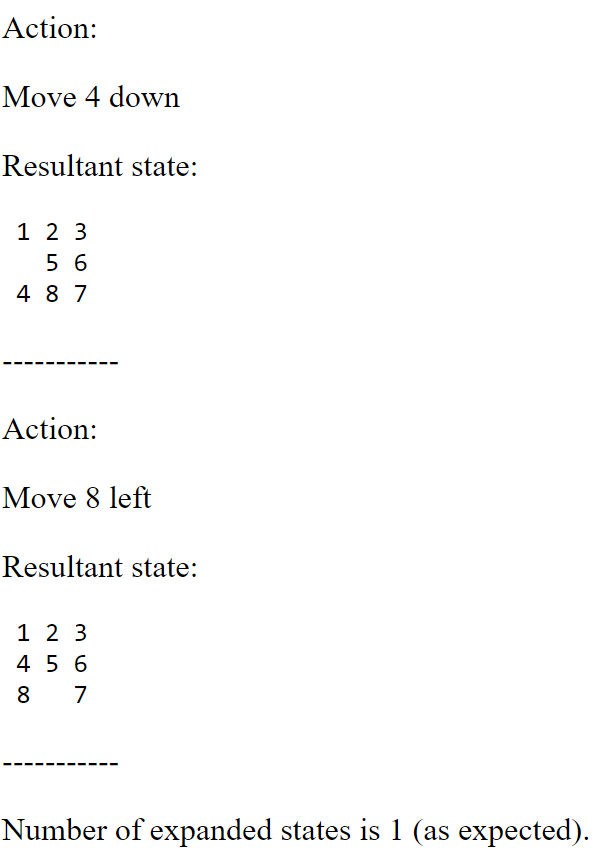
(1) board configuration:



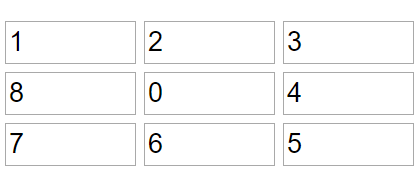
Output of “Is Goal State?”:



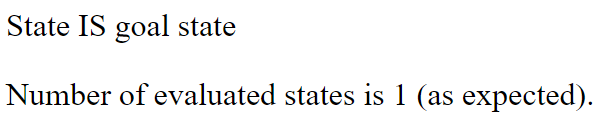
Output of “Find Successors”:



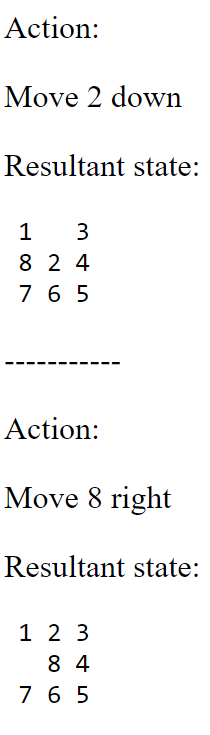
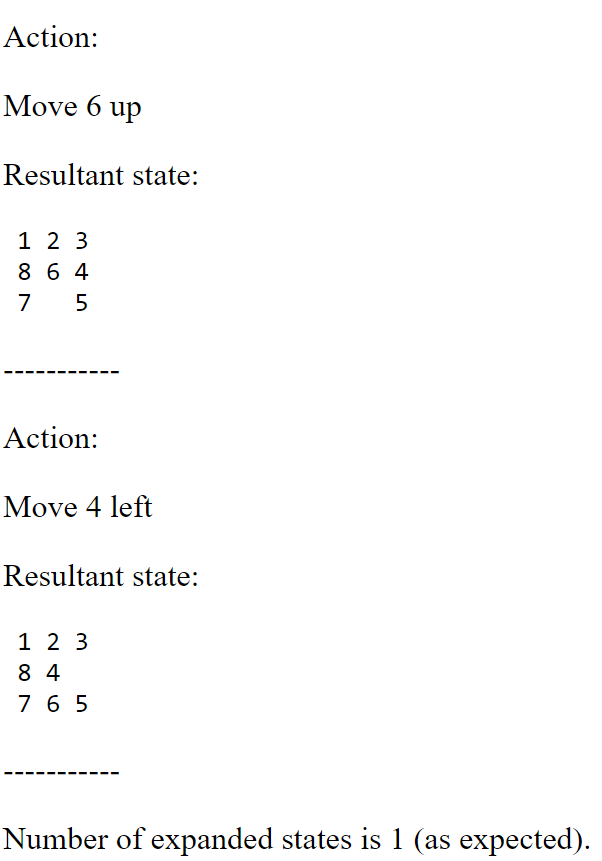
(2) board configuration:



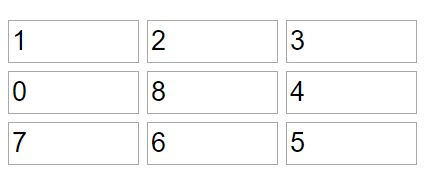
Output of “Is Goal State?”:



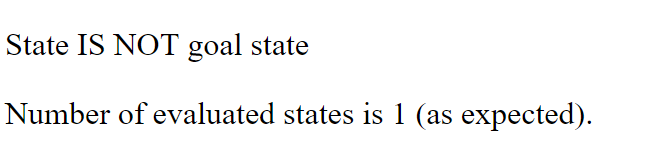
Output of “Find Successors”:

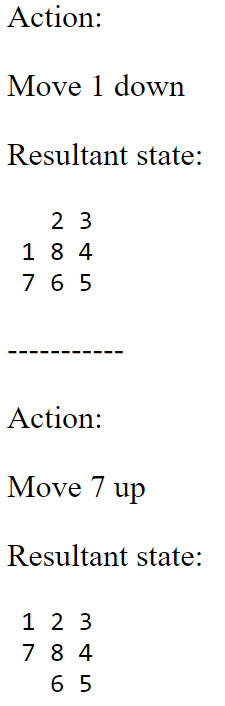
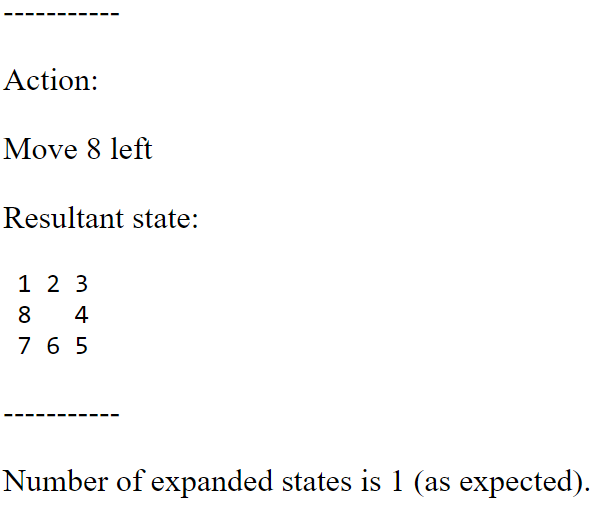
(3) board configuration:



Output of “Is Goal State?”:



Output of “Find Successors”:

3.

Code for breadth-first search**:**

function breadth\_first\_search(initial\_state) {

let open = [];

let closed = new Set();

var state\_temp,action\_temp;

let augmented\_state\_list=[];

var augmented\_state;

var state\_arr=[];

var action\_arr=[];

/\*\*\*Your code for breadth-first search here\*\*\*/

// The difference of bfs and dfs is the way to add and extract elements in open and closed

augmented\_init\_state={

state:initial\_state,

predecessor:null,

action:null

};

open.push(augmented\_init\_state);

while(open.length!=0){

aug\_state\_temp=open.shift();

if (closed.has(aug\_state\_temp)){

continue;

}

if (!is\_goal\_state(aug\_state\_temp.state)){

successors=find\_successors(aug\_state\_temp.state);

for (var i=0;i<successors.length;i++){

suc=successors[i];

augmented\_state={

state:suc.resultState,

predecessor:aug\_state\_temp, //need a function to get the predecessor

action:suc.actionID};

open.push(augmented\_state);

// augmented\_state\_list.push(augmented\_state);

}

closed.add(aug\_state\_temp);// all of the passed states are in closed set

}else{

state\_arr.unshift(aug\_state\_temp.state);

augmented\_state\_temp=aug\_state\_temp.predecessor;

while (augmented\_state\_temp!=null){

state\_arr.unshift(augmented\_state\_temp.state);

action\_temp=augmented\_state\_temp.action;

action\_arr.unshift(action\_temp);

augmented\_state\_temp=augmented\_state\_temp.predecessor;

}

state\_arr.shift();

action\_arr.shift();

return {

actions : action\_arr/\*array of action ids\*/,

states : state\_arr/\*array of states\*/

};

}

}

console.log("No solution found!")

return null;

}

4.

Code for depth-limited search**:**

function depth\_limited\_search(initial\_state,depth\_limit) {

/\*\*\*Your code for depth-limited search here!\*\*\*/

let open = [];

let closed = new Set();

var state\_temp,action\_temp;

let augmented\_state\_list=[];

var augmented\_state;

var state\_arr=[];

var action\_arr=[];

aug\_state\_temp={

state:initial\_state,

predecessor:null,

action:null,

depth:0

};

open.push(aug\_state\_temp);

while(open.length!=0 ){

aug\_state\_temp=open.pop(); // pop up the last element in open

if (aug\_state\_temp.depth>depth\_limit){

continue;

}

if (!is\_goal\_state(aug\_state\_temp.state)){

successors=find\_successors(aug\_state\_temp.state);

for (var i=0;i<successors.length;i++){

suc=successors[i];

augmented\_state={

state:suc.resultState,

predecessor:aug\_state\_temp,

action:suc.actionID,

depth:aug\_state\_temp.depth+1};

open.push(augmented\_state);

// augmented\_state\_list.push(augmented\_state);

}

closed.add(aug\_state\_temp);

}else{

state\_arr.unshift(aug\_state\_temp.state);

augmented\_state\_temp=aug\_state\_temp.predecessor;

while (augmented\_state\_temp!=null){

state\_arr.unshift(augmented\_state\_temp.state);

action\_temp=augmented\_state\_temp.action;

action\_arr.unshift(action\_temp);

augmented\_state\_temp=augmented\_state\_temp.predecessor;

}

state\_arr.shift();

action\_arr.shift();

return {

actions : action\_arr/\*array of action ids\*/,

states : state\_arr/\*array of states\*/

};

}

}

console.log("No solution found!")

return null;

}

5.

Code for iterative-deepening search**:**

function iterative\_deepening\_search(initial\_state) {

/\*\*\*Your code for iterative deepening search here!\*\*\*/

let depth\_limit\_upperbound=Number.POSITIVE\_INFINITY;

let open = [];

let closed = new Set();

var augmented\_state;

var state\_arr=[];

var action\_arr=[];

for (var depth\_limit=1;depth\_limit<depth\_limit\_upperbound;depth\_limit++){

open = [];

closed = new Set();

aug\_state\_temp={

state:initial\_state,

predecessor:null,

action:null,

depth:0

};

open.push(aug\_state\_temp);

while(open.length!=0 ){

aug\_state\_temp=open.pop(); // pop up the last element in open

if (aug\_state\_temp.depth>depth\_limit){

continue;

}

if (!is\_goal\_state(aug\_state\_temp.state)){

successors=find\_successors(aug\_state\_temp.state);

for (var i=0;i<successors.length;i++){

suc=successors[i];

augmented\_state={

state:suc.resultState,

predecessor:aug\_state\_temp, //need a function to get the predecessor

action:suc.actionID,

depth:aug\_state\_temp.depth+1};

open.push(augmented\_state);

}

closed.add(aug\_state\_temp);// all of the passed states are in closed set

}else{

state\_arr.unshift(aug\_state\_temp.state);

augmented\_state\_temp=aug\_state\_temp.predecessor;

while (augmented\_state\_temp!=null){

state\_arr.unshift(augmented\_state\_temp.state);

action\_temp=augmented\_state\_temp.action;

action\_arr.unshift(action\_temp);

augmented\_state\_temp=augmented\_state\_temp.predecessor;

}

state\_arr.shift();

action\_arr.shift();

return {

actions : action\_arr/\*array of action ids\*/,

states : state\_arr/\*array of states\*/

};

}

}

}

console.log("No solution found!")

return null;

}

6. Code for A \* search algorithm:

function astar\_search(initial\_state) {

let open = new FastPriorityQueue(function(a,b) { return a.estimated\_total\_cost < b.estimated\_total\_cost; });

let closed = new Set();

let fixed\_step\_cost = 1; //Assume action cost is constant

var action\_temp;

var augmented\_state;

var state\_arr=[];

var action\_arr=[];

est\_total\_cost=calculate\_heuristic(initial\_state)+0;

augmented\_init\_state={

state:initial\_state,

predecessor:null,

action:null,

estimated\_total\_cost:est\_total\_cost,

depth:0

};

open.add(augmented\_init\_state);

while(open.length!=0){

aug\_state\_temp=open.poll();

if (closed.has(aug\_state\_temp)){

continue;

}

if (!is\_goal\_state(aug\_state\_temp.state)){

successors=find\_successors(aug\_state\_temp.state);

for (var i=0;i<successors.length;i++){

suc=successors[i];

est\_cost\_temp=calculate\_heuristic(suc.resultState)+aug\_state\_temp.depth+1;

augmented\_state={

state:suc.resultState,

predecessor:aug\_state\_temp,

action:suc.actionID,

estimated\_total\_cost:est\_cost\_temp,

depth:aug\_state\_temp.depth+1};

open.add(augmented\_state);

// augmented\_state\_list.push(augmented\_state);

}

closed.add(aug\_state\_temp);// all of the passed states are in closed set

}else{

state\_arr.unshift(aug\_state\_temp.state);

augmented\_state\_temp=aug\_state\_temp.predecessor;

while (augmented\_state\_temp!=null){

state\_arr.unshift(augmented\_state\_temp.state);

action\_temp=augmented\_state\_temp.action;

action\_arr.unshift(action\_temp);

augmented\_state\_temp=augmented\_state\_temp.predecessor;

}

state\_arr.shift();

action\_arr.shift();

return {

actions : action\_arr/\*array of action ids\*/,

states : state\_arr/\*array of states\*/

};

}

}

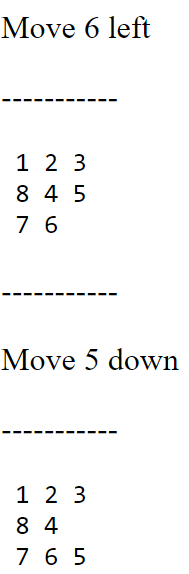
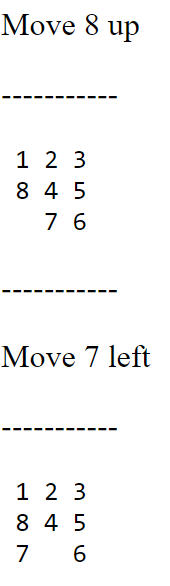
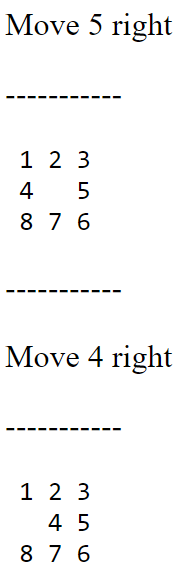
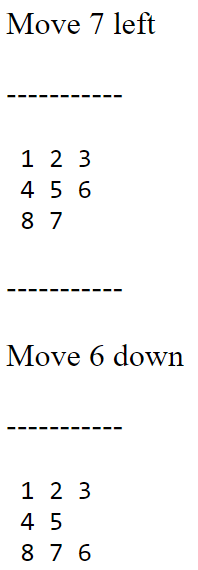
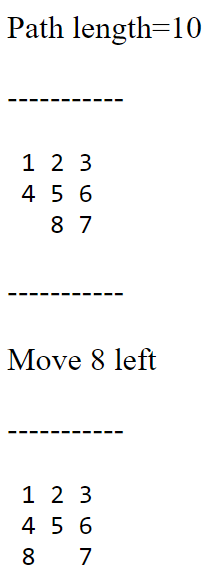
console.log("No solution found!")

return null;

}

7. I used the same board configuration as in problem 2.

For board configuration (1), the solutions are all the same for the search methods. It is:



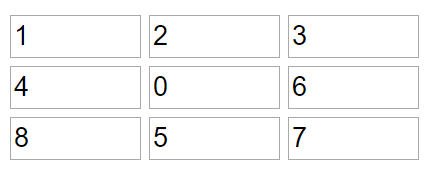
The solution above is the optimal solution for the initial configuration, and the path length is 10.

For board configuration (2), it is already the goal state. No need to search.

For board configuration (3), the solutions are all the same for the search methods. Just move “8” to the left, then I got the goal state.

8.

The board configuration is as follows:



The numbers of states evaluated and expanded for each search function are shown below. DLS (1) is for the depth limit, which is 10, from BFS and DLS (2) is for the twice of first depth limit. A \* (Ma) is the case using Manhattan Distance heuristics, A \* (Mi) is the case using Misplaced Tile Count, and A \* (St) is for “stupid” heuristic.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | BFS | DLS (1) | DLS (2) | IDS | A \* (Ma) | A \* (Mi) | A \* (St) |
| # States evaluated | 60205 | 22394 | 76823 | 62503 | 66 | 497 | 40264 |
| # States expanded | 60204 | 22393 | 76822 | 62502 | 65 | 496 | 40263 |

The numbers match my expectations. The number of states evaluated includes the root node (initial state in our case), while the number of states expanded doesn’t include the root node. So, the latter number is always one less than the former. And comparing different search methods, we can see, the number of evaluated states for BFS is very large, as we discussed in class, the time and space complexity of BFS is exponential. For DLS (1), because it uses the depth limit just for the solution, its performance is much better, while DLS (2) evaluates much more states than DLS (1). For IDS, the numbers are also large, because it searches from depth limit = 1 to the solution’ depth limit 10. As for A \* search, the numbers for 3 cases are significantly different, which indicates that complexity of A \* strongly depends on the choice of heuristic function. If we use Manhattan Distance heuristics and Misplaced Tile Count, the performance is very good, and Manhattan Distance heuristics is a little bit better than Misplaced Tile Count, while for stupid heuristics, number of evaluated states is also large because the complexity is also exponential.