Jeremy Grifski

CSE 5521

September 12, 2019

Homework 3

1. Implement a successor function and goal test function for the 8-puzzle problem, as described in the slides. Refer to the provided template file eight\_puzzle\_student.js for more detailed instructions. (Also see the two\_jugs.js and vaccuum.js files for examples of these functions for some other problems.) (4 pts)

**See Attached**

1. Come up with several board configurations and use eight\_puzzle.htm to test your functions from (1) against them. (Be sure to include the goal state!) Do the results from your functions match your expectations? Explain. (1 pt)

**After playing with a few configurations, I can confirm that the goal test works well. Since I check the current state against a hardcoded goal state, I wouldn’t expect it to fail. As for the successor state function, it’s a bit harder to prove. To implement it, I used a mapping of action IDs to row, column vectors which represent the move in terms of integers. After some tweaking, I’m confident this method works as well. See tests below.**

|  |  |  |
| --- | --- | --- |
| 8 | 1 | 3 |
| 2 | 4 | 5 |
| 7 | 6 |  |

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 8 |  | 4 |
| 7 | 6 | 5 |

|  |  |  |
| --- | --- | --- |
| 8 | 1 | 3 |
| 2 | 4 | 5 |
| 7 | 6 |  |

1. Implement the breadth-first search algorithm. Refer to the provided template file bfs.js for more detailed instructions. (6 pts)

Your search functions must be generic (i.e., they don’t depend on the problem you are solving). You should be able to use the provided example problems in two\_jugs.htm and vaccuum.htm as additional tests for your code.

**See Attached.**

1. Implement the depth-limited search algorithm. Refer to the provided template file dls.js for more detailed instructions. (6 pts)

**See Attached.**

1. Implement the iterative-deepening search algorithm. Refer to the provided template file ids.js for more detailed instructions. (2 pts)

**See Attached.**

1. Implement the A\* search algorithm. Refer to the provided template file astar.js for more detailed instructions. (4 pts)

**See Attached.**

1. Come up with several board configurations and test your 4 search functions on them (you may re-use the boards from (2) ). Run your depth-limited search twice, first using as the depth limit the length of the path returned by either your BFS or IDS. Second, use twice that value. Do the returned solutions (or lack thereof) match your expectations? Explain. (1 pt)

**First, this problem helped me isolate a bug in my DLS solution (off by one) and another bug in my BFS solution (using shift/unshift improperly). Then, I ended up with a bizarre behavior where the proper depth works but doubling the depth causes the path length to be larger. Of course, that makes sense because DLS is going to continually deepen until it hits the limit or the goal. That goal path is not necessarily the shortest path. When DLS is restricted to the proper length, it’s going to get it right.**

1. Choose a non-trivial board configuration and report the number of states evaluated and expanded for each search function. Run depth-limited search with two different depth values as in (7). Test A\* search using both the Misplaced Tile Count and Manhattan Distance heuristics (both have been provided for you in eight\_puzzle\_student.js). Also, test A\* using a “stupid” heuristic that returns only 0. Do these values match your expectations? Discuss. (2 pts)

|  |  |  |
| --- | --- | --- |
| 1 | 5 | 2 |
|  | 4 | 3 |
| 8 | 7 | 6 |

**BFS: 1018 states evaluated, 1017 states expanded, and path length of 12.**

**IDS: 139206 states evaluated, 49022 states expanded, and a path length of 12.**

**A\* (Manhattan): 23 states evaluated, 22 states expanded, and a path length of 12.**

**A\* (Misplaced): 78 states evaluated, 77 states expanded, and a path length of 12.**

**A\* (Zero): 1233 states evaluated, 1232 states expanded, and a path length of 12.**

**DLS (depth=10): 18724 states evaluated, 6436 states expanded, and no solution found.**

**DLS (depth=12): 58965 states evaluated, 20273 states expanded, and a path length of 12.**

**DLS (depth=24): 58977 states evaluated, 20285 states expanded, and a path length of 22.**

**For the most part, I’m not that surprised by the results. However, I do think the A\* (Zero) result is interesting. That said, I figured the algorithm would behave like Djikstra without a heuristic. Other than that, I’m not sure if the discrepancy between the other two heuristics is significant. I’d need to test it a bit more.**

1. (Optional) Include an estimate of the time you (total if working in a pair) spent working on this assignment. (This will used to help evaluate how to adjust assignments in future iterations of the course.)

**On Thursday, I started working on this assignment right when I got home at 6PM and worked on it until about midnight—so ~6 hours. Of course, 4 of these hours was spent coding. The remaining time was spent completing the report.**

eight\_puzzle\_student.js

//////////////////////////////////////////////////////////////////////////////

// Complete the following two functions

//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!)

const goal\_state = [

[1, 2, 3],

[8, 0, 4],

[7, 6, 5]

];

for (var i = 0; i < goal\_state.length; i++) {

for (var j = 0; j < goal\_state[i].length; j++) {

if (state.grid[i][j] != goal\_state[i][j]) {

return false;

}

}

}

return true;

}

/\*\*

\* Computes the location of the blank.

\*

\* @param {Array} state

\* @returns {Array} location of blank in row by column

\*/

function find\_blank(state) {

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

for (var j = 0; j < state.grid[0].length; j++) {

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

return [i, j];

}

}

}

}

/\*\*

\* Determines if a position is valid.

\*

\* @param {Array} position a pair of values i, j

\* @returns {boolean} true if position is valid

\*/

function is\_valid\_position(position) {

return position[0] >= 0 && position[0] <= 2 && position[1] >= 0 && position[1] <=2;

}

//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 find\_successors(state) {

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

let successors=[];

const actions = {

1: [-1, 0],

2: [1, 0],

3: [0, -1],

4: [0, 1]

}

const blankPosition = find\_blank(state);

for (var i = 1; i < 5; i++) {

// clone state

let newState = {

grid : state.grid.map(x => x.slice(0))

};

// Generate new blank position using available actions

newBlankPosition = [

blankPosition[0] + actions[i][0],

blankPosition[1] + actions[i][1]

]

// If new move is valid, generate new state and add it to successors

if (is\_valid\_position(newBlankPosition)) {

valAtNewPosition = newState.grid[newBlankPosition[0]][newBlankPosition[1]]

newState.grid[newBlankPosition[0]][newBlankPosition[1]] = 0;

newState.grid[blankPosition[0]][blankPosition[1]] = valAtNewPosition;

successors.push({

actionID : i,

resultState : newState

});

}

}

return successors;

}

dls.js

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

var stack = [];

stack.push(wrap\_dfs\_state(initial\_state, null, null, 1));

while (stack.length != 0 && !is\_goal\_state(stack[stack.length-1].state)) {

let state = stack.pop();

if (state.depth < depth\_limit) {

let successors = find\_successors(state.state);

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

let successor = successors[i];

stack.push(wrap\_dfs\_state(successor.resultState, state, successor.actionID, state.depth + 1))

}

}

}

return stack.length == 0 ? null: compute\_path(stack[stack.length-1]);

}

/\*\*

\* Wraps the state to include predecessor and action

\*

\* @param {Array} state the state as a 3x3 grid

\* @param {Object} predecessor the return value of this function for a previous state

\* @param {number} action the action to achieve this state

\* @param {number} depth the current stack depth

\*/

function wrap\_dfs\_state(state, predecessor, action, depth) {

return {

state: state,

predecessor: predecessor,

action: action,

depth: depth

}

}

bfs.js

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

let open = []; //See push()/pop() and unshift()/shift() to operate like stack or queue

//https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array

let closed = new Set(); //https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Set

// Perform breadth-first search

open.push(wrap\_bfs\_state(initial\_state, null, null));

while (open.length != 0 && !is\_goal\_state(open[0].state)) {

let state = open.shift();

closed.add(state\_to\_uniqueid(state.state));

let successors = find\_successors(state.state);

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

let successor = successors[i];

if (!closed.has(state\_to\_uniqueid(successor.resultState))) {

open.push(wrap\_bfs\_state(successor.resultState, state, successor.actionID));

}

}

}

// If we've exhausted the open list, we've failed.

return open.length == 0 ? null: compute\_path(open[0]);

}

/\*\*

\* Computes the path using the predecessor path.

\*

\* @param {Object} node the goal state node

\*/

function compute\_path(node) {

let path = {

actions: [],

states: []

};

while (node.predecessor != null) {

path.states.unshift(node.state);

path.actions.unshift(node.action);

node = node.predecessor;

}

return path;

}

/\*\*

\* Wraps the state to include predecessor and action

\*

\* @param {Array} state the current state

\* @param {Object} predecessor the previous state

\* @param {number} action the current action

\*/

function wrap\_bfs\_state(state, predecessor, action) {

return {

state: state,

predecessor: predecessor,

action: action

}

}

astar.js

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

// Perform a-star

open.add(wrap\_astar\_state(initial\_state, null, null, 0, 0));

while (!open.isEmpty() && !is\_goal\_state(open.peek().state)) {

let state = open.poll();

closed.add(state\_to\_uniqueid(state.state));

let successors = find\_successors(state.state);

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

let successor = successors[i];

if (!closed.has(state\_to\_uniqueid(successor.resultState))) {

open.add(

wrap\_astar\_state(

successor.resultState,

state,

successor.actionID,

state.path\_cost + fixed\_step\_cost,

state.path\_cost + calculate\_heuristic(successor.resultState)

)

);

}

}

}

// If we've exhausted the open list, we've failed.

return open.isEmpty() ? null: compute\_path(open.peek());

}

/\*\*

\* Wraps the state to include predecessor and action

\*

\* @param {Array} state the current state

\* @param {Object} predecessor the previous state

\* @param {number} action the current action

\* @param {number} path\_cost the current path cost

\* @param {number} estimated\_total\_cost the estimated total path cost

\*/

function wrap\_astar\_state(state, predecessor, action, path\_cost, estimated\_total\_cost) {

return {

state: state,

predecessor: predecessor,

action: action,

path\_cost: path\_cost,

estimated\_total\_cost: estimated\_total\_cost

}

}

ids.js

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

let i = 1;

let path = depth\_limited\_search(initial\_state, i);

while (path == null) {

i++;

path = depth\_limited\_search(initial\_state, i);

}

return path;

}