Demetrius Johnson

CIS-479-RETAKE

With Dr. Wang

Program 1

IDS and A\* Search

9-18-22

I was only able to do one program: A\* search because I wrote and adapted the Priority Queue as a binary search tree and the hash table myself. Also, I have the correct solution output, but some slight difference because my agent stops searching after hitting goal.

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated with medium confidence

Here are my code files. I have them well commented, so I will not write anything about how I implemented them. Just check comments please. I am trying to get the points I can. I could have done both programs and 100% correct but Dr. Wang in my opinion did not give us enough time. Thank you. I absolutely hate to turn in half-done work and fast reports, but he is taking 1%/hr from my grade.

# MAIN.CPP

// CIS-479\_RETAKE-P1-MEECH.cpp : This file contains the 'main' function. Program execution begins and ends there.

//

//Author: Demetrius E Johnson

//Purpose: Implement the A\* Search and Iterative Deepening Search (IDS) algorithms

//Date Created: 7/15/22

//Date Modified:

//CIS-479: Windy Maze Problem (North wind)

/\* [] = blank tile, #= wall obstacle, 0 = initial state, G = goal state

[] [] [] [] []

[] # # [] []

[] # G [] []

0 # # [] []

[] # [] [] []

[] [] [] [] []

Move North (against wind) cost = 3.

Move South (with wind) cost = 1.

Move East or West (side wind) cost = 2.

//special notes: each valid location is a state.

\*/

#include <iostream>

#include "FIFO.h"

#include "StateNode.h"

#include "minHeapPQ.h"

#include "minHeapPQ.cpp"

#include "QuadraticProbing.h"

#include <cstdlib>

#include <iomanip> //so we can set number of digits output for integer in the output table

using namespace std;

////////////////////////FUNCTION DECLARATIONS//////////////////////

int getValueLocation\_row(StateNode& state, int value);

int getValueLocation\_col(StateNode& state, int value);

int calcHeuristic(const StateNode& goal\_state, const StateNode& curr\_state);

void output\_maze\_table(vector<vector<int>>& table, const int& numRows, const int& numCols);

int expandNode\_WNES\_order(vector<vector<int>>& table, StateNode& parentNode);

void output\_expansionOrder(vector<vector<int>> table\_copy);

void output\_traced\_solution(vector<vector<int>> table\_copy, StateNode& terminal);

bool A\_Star\_search(vector<vector<int>>& table, StateNode& solutionNode);

void welcome\_menu(void);

//////////////GLOBAL VARIABLES//////////////

int initialize\_hashTableSize = 300; //use this to set hash table size in the event that solution is found but cannot be traced due to rehash() function needing to be called and making parent pointers bad

minHeapPQ<StateNode> frontierSet\_PQ; //Frontier Set: all leaf nodes available for expansion at any given point (in time / during traversal).

HashTable<StateNode> exploreSet\_HashTable(initialize\_hashTableSize); //Use explored set to remember every expanded node to avoid redundant paths.

HashTable<FIFO> FIFO\_tie\_breaker\_HashTable; //use this hash table to maintain FIFO order among nodes with same EvalFunction magnitude

StateNode stateGoal;

StateNode stateStart;

int searchLoop\_Limitation;

int maze\_Rows = 6;

int maze\_Cols = 5;

int init\_row\_location = 3;

int init\_col\_location = 0;

int goal\_row\_location = 2;

int goal\_col\_location = 2;

vector<StateNode> expansionOrder\_vector; //track expansion order

//////////////MAIN FUNCTION///////////////////

int main()

{

welcome\_menu();

/////////////A\* Search//////////////////////////

//initialize windy maze (North Wind)

/\* [] = blank tile, #= wall obstacle, i = initial state, G = goal state

[] [] [] [] []

[] # # [] []

[] # G [] []

i # # [] []

[] # [] [] []

[] [] [] [] []

\*/

//initiate 2D array table using vectors

vector<vector<int>>windy\_maze\_array;

windy\_maze\_array.resize(maze\_Rows);

for (int i = 0; i < windy\_maze\_array.size(); i++)

windy\_maze\_array[i].resize(maze\_Cols);

//initialize all locations to blank tile locations, since majority are blank = 0:

for (int i = 0; i < maze\_Rows; i++)

for (int j = 0; j < maze\_Cols; j++)

windy\_maze\_array[i][j] = 0;

//initialize wall obstacle locations = INT\_MIN

windy\_maze\_array[1][1] = INT\_MIN;

windy\_maze\_array[2][1] = INT\_MIN;

windy\_maze\_array[3][1] = INT\_MIN;

windy\_maze\_array[4][1] = INT\_MIN;

windy\_maze\_array[1][2] = INT\_MIN;

windy\_maze\_array[3][2] = INT\_MIN;

//initialize goal location = INT\_MAX

windy\_maze\_array[2][2] = INT\_MAX;

//initialize start location = -1

windy\_maze\_array[init\_row\_location][init\_col\_location] = -1;

//output maze table to ensure it was built properly

cout << "\nNorth Wind. Movement costs: W and E side wind = 2; N against wind = 3, S with wind = 1.\n\n";

output\_maze\_table(windy\_maze\_array, maze\_Rows, maze\_Cols);

//Initialize the start and goal states; first their location; then their heuristic values, pathcost, and eval fucntion

stateStart.row\_location = init\_row\_location;

stateStart.col\_location = init\_col\_location;

stateGoal.row\_location = goal\_row\_location;

stateGoal.col\_location = goal\_row\_location;

stateStart.heuristicVal = calcHeuristic(stateGoal, stateStart);

stateGoal.heuristicVal = calcHeuristic(stateGoal, stateGoal);

//initialize starting path cost for initial state (the root) and set evaluation function value.

stateStart.pathCost = 0;

stateStart.setEval\_value();

//insert initial state as first node in the frontier set (initialize frontier); also add to FIFO\_tie\_breaker tracking array

FIFO\_tie\_breaker\_HashTable.insert(stateStart.tie\_breaker);

frontierSet\_PQ.insert(stateStart);

//set loop search limitation: allows for controlling and being notifiedof how large of a solution you want to find or if there is some loop that occurs

searchLoop\_Limitation = 2500;

//ready to search:

StateNode solutionNode;

bool solution\_found = false;

solution\_found = A\_Star\_search(windy\_maze\_array, solutionNode);

//print expansion order and solution trace:

if (solution\_found) {

output\_expansionOrder(windy\_maze\_array);

output\_traced\_solution(windy\_maze\_array, solutionNode);

}

else

cout << "\n\n~NO SOLUTION FOUND. POSSIBLE ERROR, POSSIBLE Search Depth Limitation or no more nodes to search.~\n\n";

cout << "\nProgram Completed. Thank you. Exiting 0.\n";

system("pause");

return 0;

}

//FUNCTION DEFINITIONS//

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

enum PathCostDirection { WEST = 2, NORTH = 3, EAST = 2, SOUTH = 1 };

int calcHeuristic(const StateNode& goal\_state, const StateNode& curr\_state) {

int heuristic = 0;

//cout << "\ninitial state: (" << curr\_state.row\_location << ", " << curr\_state.col\_location << ")\n";

//cout << "\goal state: (" << goal\_state.row\_location << ", " << goal\_state.col\_location << ")\n";

//calculate heuristic (Manhattan distance) for the state = difference in row location + difference in col location

//if goal row is LESS than current row, then we need to move N (into the N wind) = cost of 3 per northward step:

if ((goal\_state.row\_location - curr\_state.row\_location) < 0)

heuristic += NORTH \* abs(goal\_state.row\_location - curr\_state.row\_location);

//else if goal row is GREATER than current row, then we need to move S (with the N wind) = cost of 1 per southward step:

if ((goal\_state.row\_location - curr\_state.row\_location) > 0)

heuristic += SOUTH \* abs(goal\_state.row\_location - curr\_state.row\_location);

//finally, add the column component of the heuristic value; for W and E, we have a side wind with cost = 2:

heuristic += 2 \* abs(goal\_state.col\_location - curr\_state.col\_location);

return heuristic;

}

void output\_maze\_table(vector<vector<int>>& table, const int& numRows, const int& numCols) {

for (int i = 0; i < numRows; i++) {

cout << "\t";

for (int j = 0; j < numCols; j++) {

if (table[i][j] == INT\_MIN)

cout << "#\t";

else if (table[i][j] == INT\_MAX)

cout << "G\t";

else if (i == init\_row\_location && j == init\_col\_location)

cout << "i\t";

else

cout << setw(2) << setfill('0') << table[i][j] << '\t';

}

cout << endl << endl;

}

}

int expandNode\_WNES\_order(vector<vector<int>>& table, StateNode& parentNode) {

StateNode childNode;

int numChildren\_generated = 0;

/\* below is instruction of how to expand in each direction: W, N, E, S

\*expand to WEST --> j - 1 (j >= 1) && j-1 != internal obstacle

\*expand to NORTH --> i - 1 (i >= 1) && i-1 != internal obstacle

\*expand to EAST --> j + 1 (j < numCols - 1 ) && j+1 != internal obstacle

\*expand to SOUTH --> i + 1 (i < numRows - 1) && i+1 != internal obstacle

\*/

//case: check WEST of parent (expansion node) location and ensure there is no wall or obstacle

if ( (parentNode.col\_location >= 1) && (table[parentNode.row\_location][parentNode.col\_location - 1] != INT\_MIN) ) {

childNode = parentNode;

childNode.col\_location = parentNode.col\_location - 1; //child node is WEST of parent node

//\*\*next 7 lines\*\*:

//calculate and set child heuristic value

//set g(n) == pathcost == parent cost + W move cost

//g(n) and h(n) set; now set f(n) = g(n) + f(n).

//hash and track parent --> hash parent in explore set hash table so we know that it has been explored, and set child node to point to that hashed parent.

//set hash\_string for newly created child node so that it can later be hashed when it is explored

//check to see if a node has already been added with the childNode's EvalFunction value; increment insert order value if so to maintain FIFO (see < and > functions for StateNode)

//add child to frontier set (priority queue)

childNode.heuristicVal = calcHeuristic(stateGoal, childNode);

childNode.pathCost = parentNode.pathCost + WEST;

childNode.setEval\_value();

childNode.parent = exploreSet\_HashTable.getOBJ\_reference(parentNode);

childNode.setHash\_string();

if (FIFO\_tie\_breaker\_HashTable.contains(childNode.tie\_breaker)) {

FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val++; //increase order

childNode.tie\_breaker.insert\_order\_val = FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val; //set child node to that new increased order

}

else FIFO\_tie\_breaker\_HashTable.insert(childNode.tie\_breaker);

frontierSet\_PQ.insert(childNode);

numChildren\_generated++;

}

//case: check NORTH of parent (expansion node) location and ensure there is no wall or obstacle

if ((parentNode.row\_location >= 1) && (table[parentNode.row\_location - 1][parentNode.col\_location] != INT\_MIN)) {

childNode = parentNode;

childNode.row\_location = parentNode.row\_location - 1; //child node is NORTH of parent node

//\*\*next 7 lines\*\*:

//calculate and set child heuristic value

//set g(n) == pathcost == parent cost + N move cost

//g(n) and h(n) set; now set f(n) = g(n) + f(n).

//hash and track parent --> hash parent in explore set hash table so we know that it has been explored, and set child node to point to that hashed parent.

//set hash\_string for newly created child node so that it can later be hashed when it is explored

//check to see if a node has already been added with the childNode's EvalFunction value; increment insert order value if so to maintain FIFO (see < and > functions for StateNode)

//add child to frontier set (priority queue)

childNode.heuristicVal = calcHeuristic(stateGoal, childNode);

childNode.pathCost = parentNode.pathCost + NORTH;

childNode.setEval\_value();

childNode.parent = exploreSet\_HashTable.getOBJ\_reference(parentNode);

childNode.setHash\_string();

if (FIFO\_tie\_breaker\_HashTable.contains(childNode.tie\_breaker)) {

FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val++; //increase order

childNode.tie\_breaker.insert\_order\_val = FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val; //set child node to that new increased order

}

else FIFO\_tie\_breaker\_HashTable.insert(childNode.tie\_breaker);

frontierSet\_PQ.insert(childNode);

numChildren\_generated++;

}

//case: check EAST of parent (expansion node) location and ensure there is no wall or obstacle

if ((parentNode.col\_location < (maze\_Cols - 1)) && (table[parentNode.row\_location][parentNode.col\_location + 1] != INT\_MIN)) {

childNode = parentNode;

childNode.col\_location = parentNode.col\_location + 1; //child node is EAST of parent node

//\*\*next 7 lines\*\*:

//calculate and set child heuristic value

//set g(n) == pathcost == parent cost + E move cost

//g(n) and h(n) set; now set f(n) = g(n) + f(n).

//hash and track parent --> hash parent in explore set hash table so we know that it has been explored, and set child node to point to that hashed parent.

//set hash\_string for newly created child node so that it can later be hashed when it is explored

//check to see if a node has already been added with the childNode's EvalFunction value; increment insert order value if so to maintain FIFO (see < and > functions for StateNode)

//add child to frontier set (priority queue)

childNode.heuristicVal = calcHeuristic(stateGoal, childNode);

childNode.pathCost = parentNode.pathCost + EAST;

childNode.setEval\_value();

childNode.parent = exploreSet\_HashTable.getOBJ\_reference(parentNode);

childNode.setHash\_string();

if (FIFO\_tie\_breaker\_HashTable.contains(childNode.tie\_breaker)) {

FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val++; //increase order

childNode.tie\_breaker.insert\_order\_val = FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val; //set child node to that new increased order

}

else FIFO\_tie\_breaker\_HashTable.insert(childNode.tie\_breaker);

frontierSet\_PQ.insert(childNode);

numChildren\_generated++;

}

//case: check SOUTH of parent (expansion node) location and ensure there is no wall or obstacle

if ((parentNode.row\_location < (maze\_Rows - 1)) && (table[parentNode.row\_location + 1][parentNode.col\_location] != INT\_MIN)) {

childNode = parentNode;

childNode.row\_location = parentNode.row\_location + 1; //child node is SOUTH of parent node

//\*\*next 7 lines\*\*:

//calculate and set child heuristic value

//set g(n) == pathcost == parent cost + S move cost

//g(n) and h(n) set; now set f(n) = g(n) + f(n).

//hash and track parent --> hash parent in explore set hash table so we know that it has been explored, and set child node to point to that hashed parent.

//set hash\_string for newly created child node so that it can later be hashed when it is explored

//check to see if a node has already been added with the childNode's EvalFunction value; increment insert order value if so to maintain FIFO (see < and > functions for StateNode)

//add child to frontier set (priority queue)

childNode.heuristicVal = calcHeuristic(stateGoal, childNode);

childNode.pathCost = parentNode.pathCost + SOUTH;

childNode.setEval\_value();

childNode.parent = exploreSet\_HashTable.getOBJ\_reference(parentNode);

childNode.setHash\_string();

if (FIFO\_tie\_breaker\_HashTable.contains(childNode.tie\_breaker)) {

FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val++; //increase order

childNode.tie\_breaker.insert\_order\_val = FIFO\_tie\_breaker\_HashTable.getOBJ\_reference(childNode.tie\_breaker)->insert\_order\_val; //set child node to that new increased order

}

else FIFO\_tie\_breaker\_HashTable.insert(childNode.tie\_breaker);

frontierSet\_PQ.insert(childNode);

numChildren\_generated++;

}

return numChildren\_generated;

}

void output\_expansionOrder(vector<vector<int>> table\_copy){

int expansion = 0;

cout << "NODE EXPANSION ORDER:\n\n";

for (int i = 0; i < expansionOrder\_vector.size(); i++) {

if (i == expansionOrder\_vector.size() - 1)

cout << "EXPANSION #" << expansion << ": \*\*GOAL STATE!\*\*" << endl;

else

cout << "EXPANSION #" << expansion << endl;

cout << "location: (" << expansionOrder\_vector[i].row\_location << ", " << expansionOrder\_vector[i].col\_location << ")\n";

cout << "[path cost-->g(n): " << expansionOrder\_vector[i].pathCost << "]\n[heuristic-->h(n): " << expansionOrder\_vector[i].heuristicVal << "]\n";

cout << "[EvalFunc-->g(n) + h(n): " << expansionOrder\_vector[i].EvalFunction << "]";

//build a table so that we can output the solution visually when this for for the traversal loop ends

table\_copy[expansionOrder\_vector[i].row\_location][expansionOrder\_vector[i].col\_location] = expansion;

expansion++;

cout << endl << endl << endl;

}

output\_maze\_table(table\_copy, maze\_Rows, maze\_Cols); //finally, output the table for visual representation of the expansion order

}

void output\_traced\_solution(vector<vector<int>> table\_copy, StateNode& terminal) {

StateNode\* tree\_traversal\_ptr = exploreSet\_HashTable.getOBJ\_reference(terminal);

vector<StateNode\*> traversal;

int trace = 0;

cout << "SOLUTION TRACE:\n\n";

if (exploreSet\_HashTable.get\_num\_rehash\_fx\_called()) {

cout << "\*\*Solution found, but cannot print out traced solution back to root.\*\*"

<< "\n~Rehash Fx called : shifted memory and corrupted solution trace.~"

<< "\n~Please change (increase) initialize\_hashTableSize global varibale and re - run the program.~\n";

return;

}

//use this while loop to traverse the solution trace, and then output it in the order from start state to goal

//(essentially, this loop reverses the trace since the initial data we have is from the goal state).

while (tree\_traversal\_ptr != nullptr) {

traversal.push\_back(tree\_traversal\_ptr);

tree\_traversal\_ptr = tree\_traversal\_ptr->parent;

}

for (int i = traversal.size() - 1; i >= 0; i--) {

if (i == 0)

cout << "TRACE #" << trace << ": \*\*GOAL STATE!\*\*" << endl;

else

cout << "TRACE #" << trace << endl;

cout << "location: (" << traversal[i]->row\_location << ", " << traversal[i]->col\_location << ")\n";

cout << "[path cost-->g(n): " << traversal[i]->pathCost << "]\n[heuristic-->h(n): " << traversal[i]->heuristicVal << "]\n";

cout << "[EvalFunc-->g(n) + h(n): " << traversal[i]->EvalFunction << "]";

//build a table so that we can output the solution visually when this for for the traversal loop ends

table\_copy[traversal[i]->row\_location][traversal[i]->col\_location] = trace;

trace++;

cout << endl << endl << endl;

}

output\_maze\_table(table\_copy, maze\_Rows, maze\_Cols); //finally, output the table for visual representation of the trace

}

bool A\_Star\_search(vector<vector<int>>& table, StateNode& solutionNode) {

StateNode parentNode;

do {

if (frontierSet\_PQ.isEmpty()) { return false; } //fail case: no more nodes to expand; solution not found

//else pop a node from the FIFO min heap Priority Queue for possible expansion

parentNode = frontierSet\_PQ.deleteMin(); //FIFO min heap priority queue will pop element with lowest EvalFunction value and do FIFO for tie breakers

parentNode.setHash\_string(); //set hash string

//check to see if popped node == goal state

if (parentNode == stateGoal) {

solutionNode = parentNode; //this allows the reference passed into have the terminal/solution node so that it can be traced back to the root == get solution path.

exploreSet\_HashTable.insert(solutionNode);//add to explore set (even though solution found, this is necessary for the trace\_solution function)

expansionOrder\_vector.push\_back(parentNode); //add to expansion order so final/solution node can be printed out in another function

return true;

}

//IF the popped node is not in the frontier OR explored set; FIFO priority queue and quadratic probing hash table controls which node is selected for expansion:

//THEN expand all possible children of the popped node

//in West North East South order; (check west tile, then north, then east, then south)

if (!exploreSet\_HashTable.contains(parentNode)) {

//add node to explored set: insert into hash table (if it is not already; hash function handles this)

exploreSet\_HashTable.insert(parentNode);

//expand selected node

expandNode\_WNES\_order(table, parentNode);

//add to expand order vector so that it can be printed out

expansionOrder\_vector.push\_back(parentNode);

}

searchLoop\_Limitation--; //limit the numbder of times this loop can run in case of a very large solution search, or in case of a loop occurence

} while (searchLoop\_Limitation != 0);

cout << "\n\n~LOOP OCCURENCE, OR SEARCH TREE TOO LARGE TO FIND A SOLUTION.~\n~CHANGE searchLoop\_Limitation global variable and re-run program for deeper search.~\n\n";

return false;

}

void welcome\_menu(void) {

cout << "Welcome to the A\* Search and Iterative Deepening Search (IDS) Program. Written By Demetrius Johnson.\nFor CIS-479 at UM-Dearborn with Prof Dr. Shenquan Wang. Fall 2022.\n";

}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu

// Debug program: F5 or Debug > Start Debugging menu

// Tips for Getting Started:

// 1. Use the Solution Explorer window to add/manage files

// 2. Use the Team Explorer window to connect to source control

// 3. Use the Output window to see build output and other messages

// 4. Use the Error List window to view errors

// 5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files to the project

// 6. In the future, to open this project again, go to File > Open > Project and select the .sln file

# MIN HEAP PQ.H

# MIN HEAP PQ.CPP

//Author: Demetrius E Johnson

//Purpose: creat a min heap priority queue template class so I can use it for my MST program and for future uses

//Date Created: 7/25/21

//Date Modified: 9/18/22

//New adaptation: added functions to support my CIS-479 P1 windy maze puzzle A\* search program

//note: minHeapPQ is a template class. There are two solutions to avoid FATAL LINK ERROR:

//1) Place do not include cpp file in the project, only include the header, and add #include .cpp at end of header file,

//then just #include the .h file in the main program source file.

//2)just #include the .h and the cpp files of the header in the main program source file.

//\*\*for this situation, I tested both cases (they both work!), and decided to go with option 1 for this test project.

#include "minHeapPQ.h"

//pq using an array

//min will always be at root

//chidlren will be greater than their parent

//i has children 2i and 2i+1 (left and right child, respectively)

//i has parent i/2

//start of a priority queue using a heap structure is at index i = 1 (not index 0; index 0 is place holder)

//note: since start of heap is nullObject placeholder (index 0 = null element), then size of heap == will always return 1; thus true heap size is heapSize - 1

template<typename T>

minHeapPQ<T>::minHeapPQ() {

vectorPtr = &heapArray;

parent = -1;

leftChild = -1;

rightChild = -1;

currentPos = -1;

heapArray.push\_back(nullObject); //element 0 is a placeholder holding a T object that will be ignored /treated as null

}

template<typename T>

void minHeapPQ<T>::insert(T insertElement) {

//remember, element 1 is a placeholder; so size == 1 means element 0 == nullObject

if (heapArray.size() == 1) { //case: first element to be added to queue; simply add it and exit function

heapArray.push\_back(insertElement);

return;

}

else {//case: size of heap array > 1:

//insert the element at the end and initialize the parent and children values relative to the size of the heap upon calling the function:

heapArray.push\_back(insertElement); //insert

currentPos = heapArray.size() - 1; //track current position of inserted element

//acquire and keep track of the parent of the inserted element:

if ((heapArray.size() - 1) % 2 == 0)//if last element location is even, then do reverse left child function to find parent since (any odd or even number) \* 2 == even number

parent = currentPos / 2;

else

parent = (currentPos - 1) / 2; //else, last element location is odd, then do the reverse right child function to find parent since (any even or odd number) \* 2 + 1 == odd number

while (heapArray.at(currentPos) < heapArray.at(parent)) { //check: is the parent greater than the child? if so: swap the parent and child so that the smaller child is moved up in the queue / PQ tree

//swap operation (bubble min value up the array/heap):

T parentHolder = heapArray[parent]; //store the parent

heapArray[parent] = heapArray[currentPos]; //overwrite parent with the child

heapArray[currentPos] = parentHolder; //overwrite child with the parent

currentPos = parent; //new location of inserted element is now at the location that the parent was at (swap operation completed.)

//now update parents in the event that loop needs to run again and bubble up the inserted element another level:

if (currentPos % 2 == 0) //currentPos == even, then do reverse left child function to find parent since (any odd or even number) \* 2 == even number

parent = currentPos / 2;

else

parent = (currentPos - 1) / 2; //else, currentPos == odd, then do the reverse right child function to find parent since (any even or odd number) \* 2 + 1 == odd number

if (currentPos == 1) { return; } //element has been moved to root; has no parent; exit function as there is no more swaps possible

}

//loop will exit when inserted element is in its correct position in the min heap

}

return; //element inserted...exit function

}

template<typename T>

T minHeapPQ<T>::deleteMin(void) {

if (heapArray.size() == 1) { return nullObject; } //case: exit function if queue is empty; no minimum to pop and delete

T minVal = heapArray[1]; //set minVal to the root (element 1), which stores the min value in the heap, so that we can return it

heapArray[1] = heapArray[heapArray.size() - 1]; //overwrite front of array (root of PQ tree) with the last element in the array; technique to avoid having to resize array (which is an O(N) operation)

heapArray.erase(heapArray.end() - 1); //remove last element: vector.erase() function will not resize capacity of array as long as elements from the end are deleted; array size decreased by 1

bool noRightChild = false; //need the no right or left child check for the swap while loop

bool noLeftChild = false;

//now we must keep the tree a complete binary tree; thus we have to adjust the element moved to the root down the tree as necessary (walk down the tree by doing swaps -> O(logN)):

//root = element 1; left child = 2\*1 = 2 and right child = 2\*1 +1 = 3

currentPos = 1; //current position of element (1 = root) that will be walked down the heap to keep tree as a complete binary tree

leftChild = 2 \* currentPos; //starting left child position

rightChild = (2 \* currentPos) + 1; //starting right child position

//switch statement only allows variable initialization outside of itself or in the default section only

T minSwapElement;

int swapLocation;

switch (int heapSize = heapArray.size()) {

case 1: //element already popped from queue; thus if size is 1 then that means only the place holder nullobject is in the queue; simply return minVal

case 2: //there were only 2 elements in the queue (array size == 3); after the min element was deleted, then there is only 1 element in the queue; no swaps necessary; simply return minVal

return minVal;

case 3: //after deletion, size of array == 3; thus there are only 2 elements in the queue, we simply need to compare these two elements and determine if a swap is necessary

rightChild = 1;//set right child == root so that it is not used in the comparison for default case

default:

while //if there exists a child larger than parent...

(heapArray[currentPos] > heapArray[leftChild] || heapArray[currentPos] > heapArray[rightChild])

{

//first, check if no right or left child; default swap element and

//location to compare will be whichever is the only child that exists.

//else do: if left < right, assign swap element that left value, otherwise,

//assign it the right value (the smallest between the two which are both smaller than their parent):

if (noLeftChild) {

minSwapElement = heapArray[rightChild];

swapLocation = rightChild;

}

else if (noRightChild) {

minSwapElement = heapArray[leftChild];

swapLocation = leftChild;

}

else {

minSwapElement = (heapArray[leftChild] < heapArray[rightChild]) ? heapArray[leftChild] : heapArray[rightChild];

swapLocation = (heapArray[leftChild] < heapArray[rightChild]) ? leftChild : rightChild;

}

//perform swap:

heapArray[swapLocation] = heapArray[currentPos]; //smallest child assigned to parent value

heapArray[currentPos] = minSwapElement; //parent value assigned to smallest child

//next lines: set new currentPos and the respective children for the next while loop (if necessary):

currentPos = swapLocation;

//only change child location if it exists (don't go out of bounds in the array used for the heap/queue);

//otherwise set their value to 1 (the root == smallest value) so that in next iteration of loop they will essentially be ignored

//remember: we do size - 1 since first element in array/heap is simply a placeholder:

if (heapArray.size() - 1 >= 2 \* currentPos) {

leftChild = 2 \* currentPos;

noLeftChild = false;

}

else { noLeftChild = true; }

if (heapArray.size() - 1 >= (2 \* currentPos) + 1) {

rightChild = (2 \* currentPos) + 1;

noRightChild = false;

}

else { noRightChild = true; }

//case: no more children for the currentPos to compare/swap with; tree balanced; exit loop

if (noLeftChild && noRightChild) { break; }

}

}

return minVal; //tree is now balanced; we can return the deleted root / minValue in the queue (front of queue)

}

template<typename T>

bool minHeapPQ<T>::isEmpty(void) {

//element 0 is a placeholder; thus if size == 1 then the heap is considered empty

if (heapArray.size() == 1) { return true; }

if (heapArray.size() > 1) { return false; }

//I did both if statements to ensure function/heap class is functioning properly; size should never go below 1 (place holder always occupies element 0)

}

//return a copy the current minimum of the heap without popping it from the heap. Returns default constructor copy if heap is empty.

template<typename T>

T minHeapPQ<T>::currentMin(void) {

T copy;

if (heapArray.size() > 1)

copy = heapArray[1]; //remember, for this minHeap library, element 0 has a place holder object; first element == min == element 1

return copy;

}

//note: minHeapPQ is a template class. There are two solutions to avoid FATAL LINK ERROR:

//1) Place do not include cpp file in the project, only include the header, and add #include .cpp at end of header file,

//then just #include the .h file in the main program source file.

//2)just #include the .h and the cpp files of the header in the main program source file.

//\*\*for this situation, I tested both cases (they both work!), and decided to go with option 1 for this test project.

# FIFO.H

#ifndef FIFO\_H

#define FIFO\_H

struct FIFO {

int evalFx\_val;

int insert\_order\_val;

bool operator== (const FIFO& RHoperand) const {

if (evalFx\_val == RHoperand.evalFx\_val)

return true;

else

return false;

}

bool operator!= (const FIFO& RHoperand) const {

if (evalFx\_val != RHoperand.evalFx\_val)

return true;

else

return false;

}

};

#endif

# QUADRATIC PROBING.H

//Hash Table function provided by Professor Dr. Junhua Guo -- UM-DEARBORN --

//Modfied by Demetrius Johnson --for CIS-350 Summer 2021 and CIS-479 Summer 2022

//this version of the class was adapted for the Windy Puzzle Program 1 for CIS-479 with Doctor Shenquan Wang

#ifndef QUADRATIC\_PROBING\_H

#define QUADRATIC\_PROBING\_H

#include <vector>

#include <string>

#include "FIFO.h"

#include "StateNode.h"

using namespace std;

int nextPrime( int n );

int hash1( const string & key );

int hash1( int key );

int hash1(const StateNode& key);

int hash1(const FIFO& key);

// QuadraticProbing Hash table class

//

// CONSTRUCTION: an approximate initial size or default of 101

//

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PUBLIC OPERATIONS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// bool insert( x ) --> Insert x

// HashedObj\* getOBJ\_reference( x ) --> return reference to hashed OBJ if in table

// bool remove( x ) --> Remove x

// bool contains( x ) --> Return true if x is present

// void makeEmpty( ) --> Remove all items

// int hash( string str ) --> Global method to hash strings

// int get\_num\_rehash\_fx\_called( void ) --> return number of times rehash fx called

template <typename HashedObj>

class HashTable

{

public:

explicit HashTable( int size = 101 ) : array( nextPrime( size ) )

{ makeEmpty( ); }

bool contains( const HashedObj & x ) const

{

//to check, run hash function for object to get hash value, then check if matching hash value AND object is in table.

//quadratic probing: may require you to check several locations in hash table; you need both hash value and object stored at a location to match and location is ACTIVE

//example 1): hash value matches, ACTIVE, but wrong object at location - apply quadratic probe function to check next location.

//example 2): hash value matches, DELETED, but wrong object at location - apply quadratic probe function to check next location.

//example 3): hash value matches, ACTIVE, and object matches; return true: object is in the table.

//examnple 4): hash value matches, DELETED, and object matches; return false: object was in table, but not anymore.

return isActive( findPos( x ) );

}

void makeEmpty( )

{

currentSize = 0;

for( int i = 0; i < array.size( ); i++ )

array[ i ].info = EMPTY;

}

bool insert( const HashedObj & x )

{

// Insert x as active

int currentPos = findPos( x );

if( isActive( currentPos ) )

return false;

array[ currentPos ] = HashEntry( x, ACTIVE );

// Rehash; see Section 5.5

if( ++currentSize > array.size( ) / 2 )

rehash( );

return true;

}

int get\_num\_rehash\_fx\_called(void) {

return num\_rehash\_calls;

}

bool remove( const HashedObj & x )

{

int currentPos = findPos( x );

if( !isActive( currentPos ) )

return false;

array[ currentPos ].info = DELETED;

return true;

}

HashedObj\* getOBJ\_reference(const HashedObj & x)

{

if ( contains( x ) )

{

int tableLocation = findPos( x );

return & array[ tableLocation ].element;

}

else

return nullptr;

}

enum EntryType { ACTIVE, EMPTY, DELETED };

private:

struct HashEntry

{

HashedObj element;

EntryType info;

HashEntry( const HashedObj & e = HashedObj( ), EntryType i = EMPTY )

: element( e ), info( i ) { }

};

vector<HashEntry> array;

int currentSize;

//track number of times rehash function was called. Useful for many other functions

// -- esp adapting or correcting pointer race conditions for when

//rehash causes vector to resize casuing ptrs to point to bad memory

int num\_rehash\_calls = 0;

bool isActive( int currentPos ) const

{ return array[ currentPos ].info == ACTIVE; }

//notice const after function definition means that all elements from the this-> object which calls findPos function is const qualified;

//therefore, array is a member of the calling object (this->object), and is const qualified.

int findPos( const HashedObj & x ) const //findPos == find position to insert the hashed object; use quadratic probing insert method

{

int offset = 1;

int currentPos = myhash( x );

// Assuming table is half-empty, and table length is prime,

// this loop terminates

while( array[ currentPos ].info != EMPTY &&

array[ currentPos ].element != x )

{

// Compute ith probe

//quadratic probing; simplified version of f(i^2); derived from f(i) - f(i-1),

//which shows offset is increased by 2 + previous-offset value to get quadratic probe i^2 value

//cpu can process arithmetic much faster and efficiently than multiplication operations

currentPos += offset;

offset += 2;

if( currentPos >= array.size( ) )

currentPos -= array.size( );

}

return currentPos;

}

void rehash( ) //standard, generic re-hash function used by the default template of this quadratic probing class/program

{

num\_rehash\_calls++;

vector<HashEntry> oldArray = array;

// Create new double-sized, empty table; size needs to be a prime number for hash function to be efficient

array.resize( nextPrime( 2 \* oldArray.size( ) ) );

for( int j = 0; j < array.size( ); j++ )

array[ j ].info = EMPTY; //now, all DELETED (unused) buckets will be "freed" and availble for use as they will be set to EMPTY now

// Copy table over

currentSize = 0;

for( int i = 0; i < oldArray.size( ); i++ )

if( oldArray[ i ].info == ACTIVE )

insert( oldArray[ i ].element );

}

int myhash( const HashedObj & x ) const

{

int hashVal = hash1( x );

hashVal %= array.size( );

if( hashVal < 0 )

hashVal += array.size( );

return hashVal;

}

};

#endif

# QUADRATIC PROBING.CPP

//Hash Table function provided by Professor Dr. Junhua Guo -- UM-DEARBORN --

//Modfied by Demetrius Johnson --for CIS-350 Summer 2021 and CIS-479 Summer 2022

//this version of the class was adapted for the Windy Puzzle Program 1 for CIS-479 with Doctor Shenquan Wang

#include "QuadraticProbing.h"

#include <iostream>

using namespace std;

/\*\*

\* Internal method to test if a positive number is prime.

\* Not an efficient algorithm.

\*/

bool isPrime( int n )

{

if( n == 2 || n == 3 )

return true;

if( n == 1 || n % 2 == 0 )

return false;

for( int i = 3; i \* i <= n; i += 2 )

if( n % i == 0 )

return false;

return true;

}

/\*\*

\* Internal method to return a prime number at least as large as n.

\* Assumes n > 0.

\*/

int nextPrime( int n )

{

if( n <= 0 )

n = 3;

if( n % 2 == 0 )

n++;

for( ; !isPrime( n ); n += 2 )

;

return n;

}

/\*\*

\* A hash routine for string objects.

\*/

int hash1( const string & key )

{

int hashVal = 0;

for(unsigned int i = 0; i < key.length( ); i++ )

hashVal = 37 \* hashVal + key[ i ];

return hashVal;

}

/\*\*

\* A hash routine for ints.

\*/

int hash1( int key )

{

return key;

}

/\*\*

\* A hash routine for StateNode objects.

\*/

int hash1(const StateNode& key)

{

int hashVal = 0;

for (unsigned int i = 0; i < key.hash\_string.length(); i++)

hashVal = 17 \* hashVal + key.hash\_string[i]; //use a prime number for the multiplication value: i.e. 2, 3, 17, 37, 59, 89, 97

//if you expect large strings, then use a smaller prime number

return hashVal;

}

/\*\*

\* A hash routine for FIFO.

\*/

int hash1(const FIFO& key)

{

return key.evalFx\_val;

}

# STATENODE .H

//Author: Demetrius E Johnson

//Purpose:

//Date Created: 7/15/22

//Date Modified:

#ifndef STATENODE\_H

#define STATENODE\_H

#include <string>

#include <stdlib.h>

#include "FIFO.h"

class StateNode

{

public:

int row\_location;

int col\_location;

int heuristicVal;

int pathCost;

int EvalFunction;

StateNode\* parent;

std::string hash\_string;

FIFO tie\_breaker; //use this variable to keep track of redundant expansions and maintain FIFO among tie breaker nodes

StateNode();

void setEval\_value(void);

void setHash\_string(void);

bool operator< (const StateNode& RHoperand);

bool operator> (const StateNode& RHoperand);

bool operator== (const StateNode& RHoperand) const; //needed to make the == and != operator const qualify calling object (this->object) since Quadratic Probing findPos function uses const for calling obj

bool operator!= (const StateNode& RHoperand) const;

StateNode& operator= (const StateNode& RHoperand);

};

#endif // !1

# STATENODE.CPP

//Author: Demetrius E Johnson

//Purpose: adapted from my program I wrote in SUMMER II CIS-479 with Dr. Wang

//Date Created: 7/15/22

//Date Modified: 9/4/22

//note: each location in the windy maze is a state.

#include "StateNode.h"

//default constructor

StateNode::StateNode() {

row\_location = -1;

col\_location = -1;

heuristicVal = -1;

pathCost = -1;

EvalFunction = -1;

tie\_breaker.evalFx\_val = -1;

tie\_breaker.insert\_order\_val = -1;

hash\_string.resize(2); //I will convert the coordinates of a state into a string of size 2. For example, location (i,j) = (1,2), the string= "12".

parent = nullptr; //pointer works so that you can trace back to root; but rehash function causes this program to fail; safegaurd placed in main program.

}

//set eval function = path cost + hueristic value

void StateNode::setEval\_value(void) {

EvalFunction = (pathCost + heuristicVal);

tie\_breaker.evalFx\_val = EvalFunction; //also set this so we can use it for the FIFO hash table tracker

}

//convert table values for a given node to a uniquely ordered string value

//so that it can be used in the hash function (a part of the Quadratic Probing class)

void StateNode::setHash\_string(void) {

hash\_string[0] = row\_location + 48; //ASCII characters '0' - '9' begin at decimal 48

hash\_string[1] = col\_location + 48; //ASCII characters '0' - '9' begin at decimal 48

}

/////OPERATOR OVERLOADING///////

bool StateNode::operator< (const StateNode& RHoperand) {

//standard case:

if (EvalFunction < RHoperand.EvalFunction)

return true;

//check case for when they are equal, now we need to check insert order to maintain FIFO precedence between nodes:

else if (EvalFunction == RHoperand.EvalFunction && tie\_breaker.insert\_order\_val < RHoperand.tie\_breaker.insert\_order\_val)

return true;

else

return false;

}

bool StateNode::operator> (const StateNode& RHoperand) {

//standard case:

if (EvalFunction > RHoperand.EvalFunction)

return true;

//check case for when they are equal, now we need to check insert order to maintain FIFO precedence between nodes:

else if (EvalFunction == RHoperand.EvalFunction && tie\_breaker.insert\_order\_val > RHoperand.tie\_breaker.insert\_order\_val)

return true;

else

return false;

}

bool StateNode::operator== (const StateNode& RHoperand) const {

if (heuristicVal == RHoperand.heuristicVal)

return true;

else

return false;

}

bool StateNode::operator!= (const StateNode& RHoperand) const {

if (hash\_string != RHoperand.hash\_string)

return true;

else

return false;

}

StateNode& StateNode::operator= (const StateNode& RHoperand) {

row\_location = RHoperand.row\_location;

col\_location = RHoperand.col\_location;

heuristicVal = RHoperand.heuristicVal;

pathCost = RHoperand.pathCost;

EvalFunction = RHoperand.EvalFunction;

tie\_breaker.evalFx\_val = RHoperand.tie\_breaker.evalFx\_val;

tie\_breaker.insert\_order\_val = RHoperand.tie\_breaker.insert\_order\_val;

hash\_string = RHoperand.hash\_string;

parent = RHoperand.parent;

return \*this;

}