

Demetrius Johnson

UM-Dearbron CIS-479

Program 1 Report: A * Search

With Prof. Dr. Shenquan Wang

Summer II 2022

7-21-22

Data structure: priority queue for frontier set and hash table for explored set (See quadratic probing.cpp)

For the hash table, I used a library I built using quadratic probing and a hash string function. I wrote it for CIS-350. Since I stored each node/state in a 2D array, I needed to convert and store them into a string, which was then passed into the hash table data structure (which served as my explore set) and hashed.

- It took a lot of time to adapt it and refresh myself on quadratic probing, but in the end, it works perfectly and was integrated into the program successfully.

```
/**
 * 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 = 3 * 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;
}
```

- }
- Above is the name of the hash function I needed to add into the library, specifically for my state node class I created to store all nodes:

```
class StateNode
{
public:
    int heuristicVal;
    int pathCost;
    int EvalFunction;
    int table[3][3];
    int blankTile_row_location;
    int blankTile_col_location;
    StateNode* parent;
    std::string hash_string;

    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);
}
```

- };
- NOTE: I had to change the size of the prime number used since my string was so large and was causing my integer to overflow.

I also added a function that returns a reference to an object in the hash table (provided it is in the hash table). This allowed me to track the parents of each node placed in the hash table:

```
HashedObj* getOBJ_reference(const HashedObj & x)
{
    if ( contains( x ) )
    {
        int tableLocation = findPos( x );
        return & array[ tableLocation ].element;
    }
    else
        return nullptr;
}
•
•
```

Calculation of $f(n) = g(n) + h(n)$

- To calculate $f(n)$, I needed to of course calculate g and h .
- To calculate g (path cost), I simply added the step of the non blank tile to the cost of the parent:
- For example, if a blank tile moved SOUTH == 1, child = parent cost + SOUTH:

```
/**next 6 lines**:  
    //perform swap.  
    //save time: calculate child heuristic value based on heuristic value of parent.  
    //set  $g(n)$  == pathcost (based on non-blank tile): SOUTHWARD move of non-blank  
tile (with the North wind) (==NORTHWARD move of blank tile) = cost of 1.  
    //g(n) and h(n) set; now set  $f(n) = g(n) + f(n)$ .  
    //track parent.  
    //set new hash string now that all tiles are in correct location  
    //add child to frontier set (priority queue)  
    swap_blankTile_nonBlankTile(parentNode, childNode);  
    calcHeuristic_based_on_parent(parentNode, childNode);  
    childNode.pathCost = parentNode.pathCost + SOUTH;  
    childNode.setEval_value();  
    childNode.parent = exploreSet_HashTable.getOBJ_reference(parentNode);  
    childNode.setHash_string();  
    frontierSet_PQ.insert(childNode);  
•    numChildren_generated++;
```

To calculate h , I first calculate the h value for the entire initial state, based on the goal state, using Manhattan distance – which allowed me to treat the row and column heuristic calculation and return as separately, and then add them together:

```
int calcHeuristic_full_state(StateNode& stateStart, StateNode& stateGoal) {  
  
    int heuristic = 0;  
  
    cout << "\ninitial state:\n";  
    for (int i = 0; i < 3; i++) {  
        for (int j = 0; j < 3; j++) {  
  
            cout << stateStart.table[i][j];
```

```

        cout << " ";
    }
    cout << endl;
}

cout << "\ngoal state:\n";
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {

        cout << stateGoal.table[i][j];
        cout << " ";
    }
    cout << endl;
}

//calculate heuristic for entire state
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {

        heuristic += calcHeuristic_single_tile(stateStart, stateGoal, i, j);
    }
}
cout << endl << endl;

return heuristic;
}

int calcHeuristic_single_tile(StateNode& stateStart, StateNode& stateGoal, int& rowStart,
int& colStart) {

    int heuristic = 0;

    if (stateStart.table[rowStart][colStart] != 0) { //0 is used to represent blank tile
location

        //get row and col locations in goal table for the current value at location
(rowStart, colStart) of the start state
        int rowGoal = getValueLocation_row(stateGoal,
stateStart.table[rowStart][colStart]);
        int colGoal = getValueLocation_col(stateGoal,
stateStart.table[rowStart][colStart]);
        //get difference between goal and start state
        int rowDifference = rowGoal - rowStart;
        int colDifference = colGoal - colStart;

        //positive difference: move down (south, with the north wind);
        //negative difference: move up (north, against/into the north wind).
        if (rowDifference < 0) {

            rowDifference *= -1; //need positive cost values
            heuristic += (rowDifference * 3); //each move northward = +3 cost
        }
        else {

            heuristic += (rowDifference * 1); //each move southward = +1 cost
        }
    }
}

```

```

        //make sure col difference is always positive since east and west movement is
always +2 for wind cost
        if (colDifference < 0) { colDifference *= -1; }
        heuristic += (colDifference * 2); //each west or eastward movement
== +2 wind cost
    }

    return heuristic;
    • }

```

Then for the rest of the nodes simply calculate the h value based on parent, by subtracting the h value of where the nonblank tile was from the parent, then calculating and adding the h value for the new location of the non-blank tile:

```

void calcHeuristic_based_on_parent(StateNode& parentNode, StateNode& childNode) {

    //save time: calculate child heuristic value based on heuristic value of parent
    //formula is:      (parent's full h value)
    //                - (h value of single location where child swapped in blank tile)
    //                + (h value of single location where child swapped in non-blank
tile)
    childNode.heuristicVal =
    (
        (parentNode.heuristicVal)
        -
        (calcHeuristic_single_tile(parentNode, stateGoal,
childNode.blankTile_row_location, childNode.blankTile_col_location))
        +
        (calcHeuristic_single_tile(childNode, stateGoal,
parentNode.blankTile_row_location, parentNode.blankTile_col_location))
    );
    • }

```

Adding leaves for expansion: using min Eval F(x) and FIFO (see minHeapPQ.cpp and .h)

- As instructed, I used a priority queue. For my case, I used a Minimum Heap Priority queue, which I wrote for CIS-350. I had to fix it as this project caused me to find out it was broken. But, now I know it works 100% through a test module program. As objects are deleted, the bubble down until they are they have children that are larger. As they are added, the bubble up until their parent is smaller.

```

template<typename T>
class minHeapPQ
{
private:
    std::vector<T> heapArray; //store heap in this array
    std::vector<T>* vectorPtr; //use this to point to a specific element in the
heap in order to perform swap operations and track parent/child
    int currentPos; //the current element position (i)
    int parent; //keep track of a given elements
parent (i/2)

```

```

        int leftChild;                                //keep track of a given elements
left child (2*i)
        int rightChild;                              //keep track of a given elements
right child (2*i + 1)
        T nullObject;                                //use this to return null objects and to
set first element in heap as a place holder

public:
    minHeapPQ();                                    //default constructor
    void insert(T insertElement);                    //insert elements into PQ heap array
    T deleteMin(void);                               //delete the root; return the element
that was stored at the deleted root
    bool isEmpty(void);                              //determine if queue is empty or not
    T currentMin(void);                              //return a copy the current minimum of
the heap without popping it from the heap; returns default constructor copy if heap is
empty.
    //perhaps add some heap initialization algorithm to this template class I have
written in the future to make this class even more useful!
    •   };

```

```

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 nullptr; } //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

```

Demetrius Johnson – Program 1 – A* Search

```
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;
}
• }
```

The priority queue function already handled making sure the minimum is popped from the queue and kept it in order as well as insuring FIFO order.

Picking the smallest $f(n)$

For this, I made sure to use the priority queue and the hash table. It is in my A * search function. I used graph search as A * requires to avoid possible redundancy loops from the frontier set. I check if an object is in the hash table (explore set), and the object is already selected based on the smallest $f(n)$ from the priority queue. From there, I call my WNES expansion order function to expand the node:

```
bool A_Star_search(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 non-blank 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(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;

    • }

```

Here is the expansion order function, with just only of the expansion functionality shown and a summary of the logic to implement in proper order:

```

enum PathCostDirection {WEST = 2, NORTH = 3, EAST = 2, SOUTH = 1};
/* below is an example with instruction and visualization of how to move the blank tile
W, N, E, S

```

```

*move blank tile WEST --> j - 1 (j >= 1)
*move blank tile NORTH --> i - 1 (i >= 1)
*move blank tile EAST --> j + 1 (j <= 1 )
*move blank tile SOUTH --> i + 1 (i <= 1)

```

	j	c0	c1	c2
i				
row 0		#	#	#
row 1		#	-	#
row 2		#	#	#

```

*/

```

```

//case: check WEST of blank tile; swap blank tile to that location in a newly created
child node

```

```

if (parentNode.blankTile_col_location >= 1) {

```

```

    childNode = parentNode;
    childNode.blankTile_col_location = parentNode.blankTile_col_location - 1; //move
blank tile WEST == move non-blank tile EAST

```

```

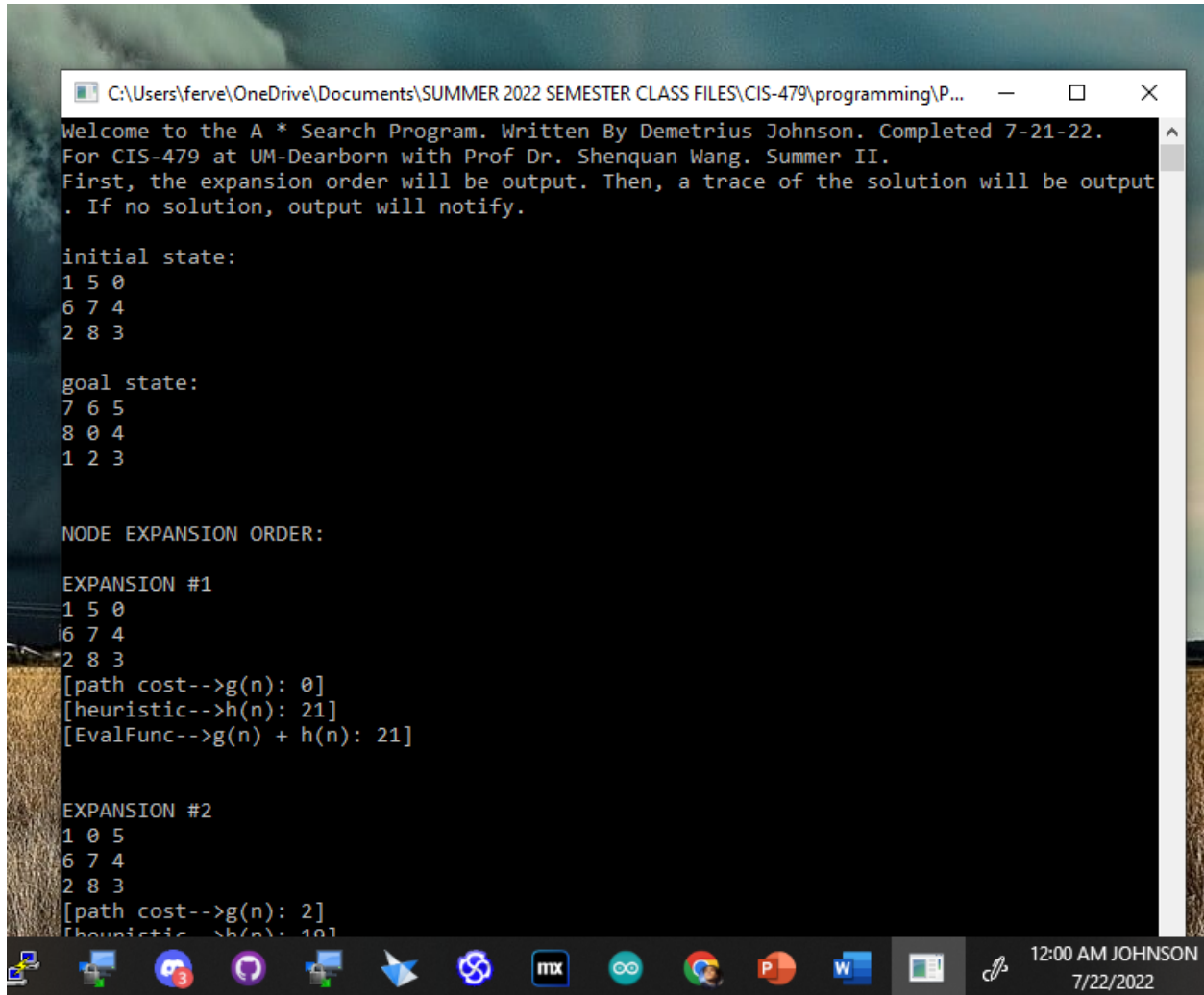
        /**next 6 lines**:
        //perform swap.
        //save time: calculate child heuristic value based on heuristic value of parent.
        //set g(n) == pathcost (based on non-blank tile): EASTWARD move of nonblank tile
(==WESTWARD move of blank tile) = cost of 2.
        //g(n) and h(n) set; now set f(n) = g(n) + f(n).
        //track parent.
        //set new hash string now that all tiles are in correct location
        //add child to frontier set (priority queue)
        swap_blankTile_nonBlankTile(parentNode, childNode);
        calcHeuristic_based_on_parent(parentNode, childNode);
        childNode.pathCost = parentNode.pathCost + EAST;

```

```
        childNode.setEval_value();  
        childNode.parent = exploreSet_HashTable.getOBJ_reference(parentNode);  
        childNode.setHash_string();  
        frontierSet_PQ.insert(childNode);  
        numChildren_generated++;  
    }  
}
```

Screenshots of solution output. Notice my last name and the time on the bottom right of each screenshot:

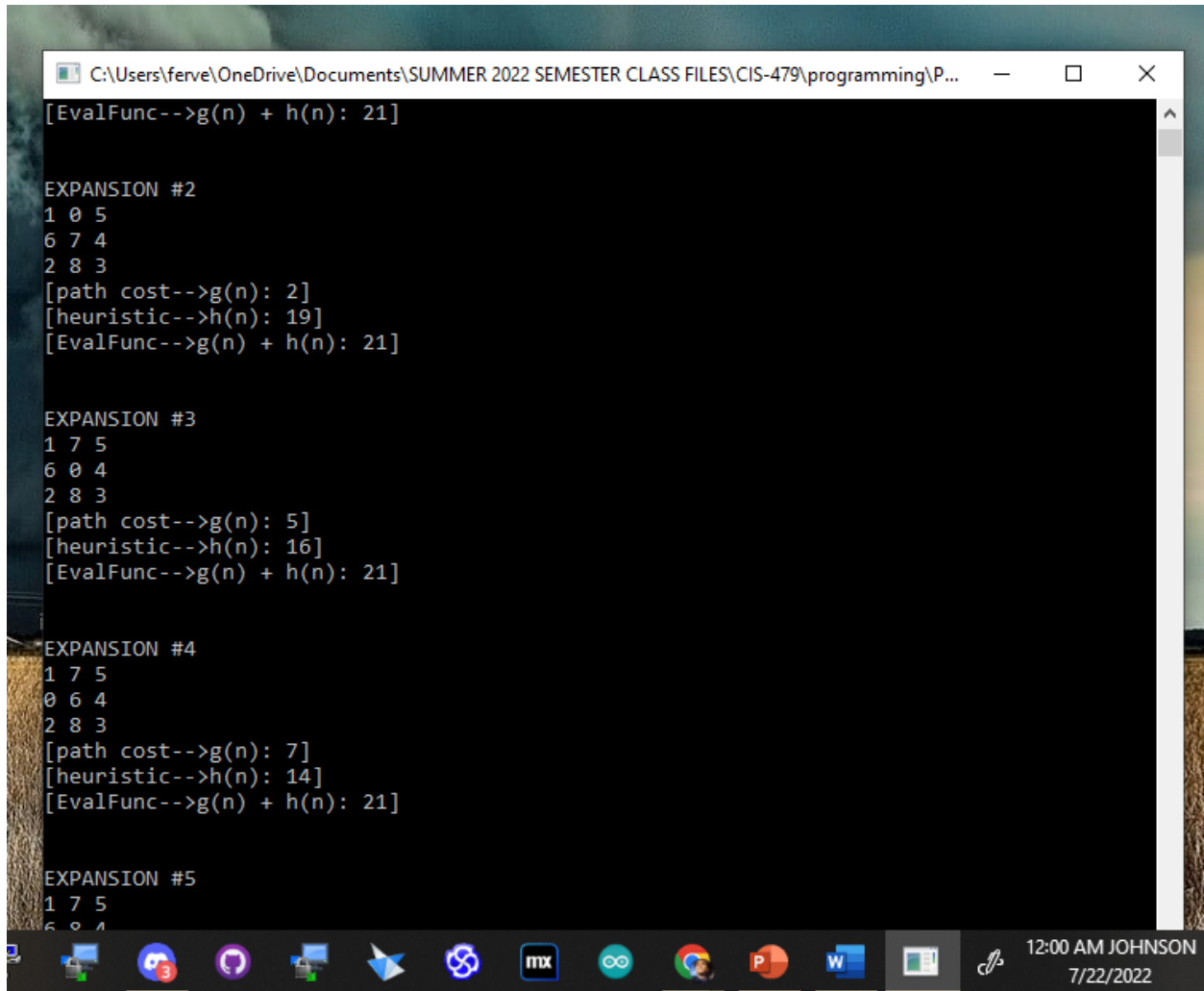
EXPANSION ORDER SCREENSHOTS:



```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P... Welcome to the A * Search Program. Written By Demetrius Johnson. Completed 7-21-22. For CIS-479 at UM-Dearborn with Prof Dr. Shenquan Wang. Summer II. First, the expansion order will be output. Then, a trace of the solution will be output. If no solution, output will notify.  
  
initial state:  
1 5 0  
6 7 4  
2 8 3  
  
goal state:  
7 6 5  
8 0 4  
1 2 3  
  
NODE EXPANSION ORDER:  
  
EXPANSION #1  
1 5 0  
6 7 4  
2 8 3  
[path cost-->g(n): 0]  
[heuristic-->h(n): 21]  
[EvalFunc-->g(n) + h(n): 21]  
  
EXPANSION #2  
1 0 5  
6 7 4  
2 8 3  
[path cost-->g(n): 2]  
[heuristic-->h(n): 10]
```

12:00 AM JOHNSON
7/22/2022

Demetrius Johnson – Program 1 – A* Search



The screenshot shows a Windows terminal window with a dark background and white text. The window title bar indicates the file path: C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P... The terminal displays the output of an A* search algorithm, showing the expansion of nodes. Each expansion step includes a list of nodes (represented as arrays of three numbers) and their associated g(n), h(n), and f(n) values. The nodes are expanded in order of their f(n) value, which is the sum of g(n) and h(n). The terminal also shows the path cost and the heuristic value for each node.

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...  
[EvalFunc-->g(n) + h(n): 21]  
  
EXPANSION #2  
1 0 5  
6 7 4  
2 8 3  
[path cost-->g(n): 2]  
[heuristic-->h(n): 19]  
[EvalFunc-->g(n) + h(n): 21]  
  
EXPANSION #3  
1 7 5  
6 0 4  
2 8 3  
[path cost-->g(n): 5]  
[heuristic-->h(n): 16]  
[EvalFunc-->g(n) + h(n): 21]  
  
EXPANSION #4  
1 7 5  
0 6 4  
2 8 3  
[path cost-->g(n): 7]  
[heuristic-->h(n): 14]  
[EvalFunc-->g(n) + h(n): 21]  
  
EXPANSION #5  
1 7 5  
6 0 4
```

The Windows taskbar at the bottom shows various application icons, including a web browser, a file explorer, and a terminal. The system clock in the bottom right corner displays the time as 12:00 AM on 7/22/2022.

Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #5
1 7 5
6 8 4
2 0 3
[path cost-->g(n): 8]
[heuristic-->h(n): 13]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #6
0 7 5
1 6 4
2 8 3
[path cost-->g(n): 8]
[heuristic-->h(n): 13]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #7
1 7 5
6 8 4
0 2 3
[path cost-->g(n): 10]
[heuristic-->h(n): 11]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #8
7 0 5
1 6 4
```

12:02 AM JOHNSON
7/22/2022

Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...
EXPANSION #8
7 0 5
1 6 4
2 8 3
[path cost-->g(n): 10]
[heuristic-->h(n): 11]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #9
7 6 5
1 0 4
2 8 3
[path cost-->g(n): 13]
[heuristic-->h(n): 8]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #10
7 6 5
1 8 4
2 0 3
[path cost-->g(n): 16]
[heuristic-->h(n): 5]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #11
7 6 5
1 8 4
0 2 3
[path cost-->g(n): 18]
[heuristic-->h(n): 3]
[EvalFunc-->g(n) + h(n): 21]
```

12:02 AM JOHNSON
7/22/2022

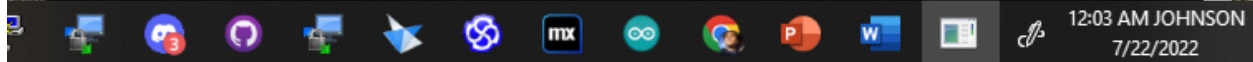
Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...
2 0 3
[path cost-->g(n): 16]
[heuristic-->h(n): 5]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #11
7 6 5
1 8 4
0 2 3
[path cost-->g(n): 18]
[heuristic-->h(n): 3]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #12
7 6 5
0 8 4
1 2 3
[path cost-->g(n): 19]
[heuristic-->h(n): 2]
[EvalFunc-->g(n) + h(n): 21]

EXPANSION #13: **GOAL STATE!**
7 6 5
8 0 4
1 2 3
[path cost-->g(n): 21]
[heuristic-->h(n): 0]
[EvalFunc-->g(n) + h(n): 21]
```



****BONUS****: SOLUTION TRACE SCREENSHOTS (using parent pointers):

```
SOLUTION TRACE:

TRACE #1
1 5 0
6 7 4
2 8 3
[path cost-->g(n): 0]
[heuristic-->h(n): 21]
[EvalFunc-->g(n) + h(n): 21]

TRACE #2
1 0 5
6 7 4
2 8 3
[path cost-->g(n): 2]
[heuristic-->h(n): 19]
[EvalFunc-->g(n) + h(n): 21]

TRACE #3
1 7 5
6 0 4
2 8 3
[path cost-->g(n): 5]
[heuristic-->h(n): 16]
[EvalFunc-->g(n) + h(n): 21]

TRACE #4
1 7 5
0 6 4
2 8 3
```

12:04 AM JOHNSON
7/22/2022

Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...

TRACE #4
1 7 5
0 6 4
2 8 3
[path cost-->g(n): 7]
[heuristic-->h(n): 14]
[EvalFunc-->g(n) + h(n): 21]

TRACE #5
0 7 5
1 6 4
2 8 3
[path cost-->g(n): 8]
[heuristic-->h(n): 13]
[EvalFunc-->g(n) + h(n): 21]

TRACE #6
7 0 5
1 6 4
2 8 3
[path cost-->g(n): 10]
[heuristic-->h(n): 11]
[EvalFunc-->g(n) + h(n): 21]

TRACE #7
7 6 5
1 0 4
2 8 3
```

12:04 AM JOHNSON
7/22/2022

Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...

TRACE #7
7 6 5
1 0 4
2 8 3
[path cost-->g(n): 13]
[heuristic-->h(n): 8]
[EvalFunc-->g(n) + h(n): 21]

TRACE #8
7 6 5
1 8 4
2 0 3
[path cost-->g(n): 16]
[heuristic-->h(n): 5]
[EvalFunc-->g(n) + h(n): 21]

TRACE #9
7 6 5
1 8 4
0 2 3
[path cost-->g(n): 18]
[heuristic-->h(n): 3]
[EvalFunc-->g(n) + h(n): 21]

TRACE #10
7 6 5
0 8 4
1 2 3
```

12:04 AM JOHNSON
7/22/2022

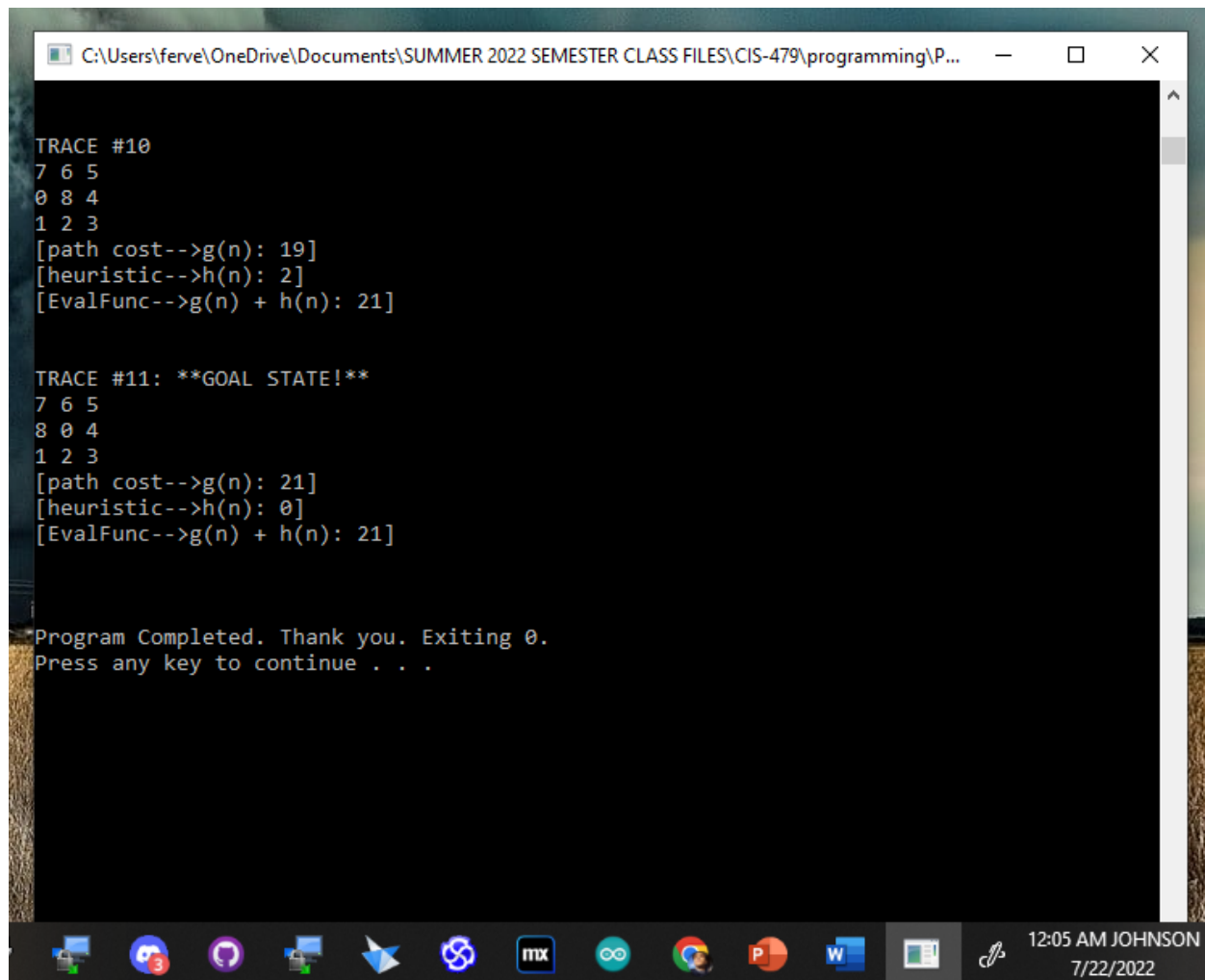
Demetrius Johnson – Program 1 – A* Search

```
C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P...

TRACE #10
7 6 5
0 8 4
1 2 3
[path cost-->g(n): 19]
[heuristic-->h(n): 2]
[EvalFunc-->g(n) + h(n): 21]

TRACE #11: **GOAL STATE!**
7 6 5
8 0 4
1 2 3
[path cost-->g(n): 21]
[heuristic-->h(n): 0]
[EvalFunc-->g(n) + h(n): 21]

Program Completed. Thank you. Exiting 0.
Press any key to continue . . .
```

The image is a screenshot of a Windows 10 desktop. A terminal window is open, displaying the output of an A* search algorithm. The window's title bar shows the file path: C:\Users\ferve\OneDrive\Documents\SUMMER 2022 SEMESTER CLASS FILES\CIS-479\programming\P... The terminal output shows two trace steps. Trace #10 shows a state with values 7, 6, 5 in the first row, 0, 8, 4 in the second, and 1, 2, 3 in the third. It calculates a path cost of 19, a heuristic of 2, and a total evaluation of 21. Trace #11 is labeled **GOAL STATE!** and shows the values 7, 6, 5 in the first row, 8, 0, 4 in the second, and 1, 2, 3 in the third. It calculates a path cost of 21, a heuristic of 0, and a total evaluation of 21. After the traces, the program prints "Program Completed. Thank you. Exiting 0." and "Press any key to continue . . .". The Windows taskbar is visible at the bottom, showing various application icons including a file explorer, a web browser, and several utility programs. The system clock in the bottom right corner indicates the time is 12:05 AM on 7/22/2022, with the user name JOHNSON.