**Erik Pantoja**

**Project 2**

**CS 4346**

**4/24/2020**

**Table of Contents**

* **Problem Description**
* **Distribution of the Work**
* **Methodologies**
* **Heuristic Functions**
* **Source Code Implementation**
* **Source Code**
* **Copy of the Program Run**
* **Analysis of the Program**
* **Analysis of the Results**
* **Conclusion**
* **References**

**Problem Description**

We were tasked with conducting research on the A star heuristic algorithm. Our research was to include coding the algorithm out, implementing multiple heuristics with the function, and gathering data on performance of the function. Heuristic functions, are a type of informed function, and in this case are search functions.

Many different algorithms exist that we conducted research on. The A star method uses heuristics to base potential best paths which include h(n). This can be calculated by several methods, but some methods typically provide better results than others. These functions that calculate h(n) are classified as admissible if they do not over-estimate the cost of the correct path. If the value is over-estimated then the function may overlook the correct solution, and not be as optimal. Our problem included coding out A star method in C++.

We programmed this search algorithm by using multiple files, and header files for neatness. We stored the values for each state in a state array. We also used constructors to create potential other states, or moves. We started at our initial state and branched off as nodes with arrays of the possible moves. We implemented h1 or otherwise known as hamming distance, using how many tiles were not in the correct position and created that as a cost. We then used that cost as a guide for the algorithm to follow. We did a similar thing for h2 or otherwise known as Manhattan distance. This was programed to calculate the distance to the correct positions. We had several heuristic functions that were created to modify the A\* values to provide a variation of the informed search, which will be discussed more in-depth.

**Distribution of Work**

For this project we met several times virtually to discuss the project. We all presented ideas and thoughts on the project to accomplish the tasks at hand. I created a group-chat and added both other members, so we can discuss the project and ideas. We started working on this project early on, well before the due date. We agreed to each try to implement A star algorithm independently, and then evaluate to see which version may be the better version based on our ideology. I created a version of A star in C++ independently, with functions for displaying, moving spaces, and helper functions. Upon meeting and discussing our progress, we decided to use the version that team member Shervone created, as it was ahead of the progress of mine.

We combined our ideas on what to implement next, and how to do it. We discussed more ideas together at additional virtual meetings using Zoom. We split up the heuristic functions at the end. I researched, created, and tested several heuristic functions including the non-admissible heuristic function of H(n) = h1(n) + 3 \*S(n) to calculate how a non-admissible heuristic function would affect the results of the program. This function was implemented into the program, and provided results. These results from s(n) were not used as part of the required heuristic functions, but were helpful in understanding the features of heuristics. I also implemented two other heuristic functions including one that calculates based of H(Db) which is disjoint pattern database. The other function I created was admissible heuristic function that calculated H(n) by checking to see if each value is in the correct row to match the goal state.

In general, the breakdown of contribution was that Shervone created the majority of the A\* star algorithm with h1, and h2 with input and ideas provided by the group. I created several heuristic functions, coded them, and modified the main program code to work with the functions I created. Our other team member Madison created a heuristic function basing the h(n) calculation off a relaxed problem of using diagonal distance.

**Methodologies**

We used several methodologies for our project including creating several charts, diagrams on white boards to map out the problems, and follow the algorithms. We created helper functions, and tested each step of the progress. We evaluated each other’s work on if there exists a need for improvements. We collected data off each heuristic algorithm and recorded the data in charts. These charts are reflected in this report with the data obtained. This data was compared and conclusions were drawn from the results. The algorithms were tested on multiple computers, and IDES with only differences between them was the speed, and memory use. We tested implementing other search algorithms including IDA, SMA as well as required research to gain more experience and knowledge. We were able to create the A star algorithm to run with all required algorithms and heuristics.

**Heuristic Functions**

We implemented several heuristic functions into our program to gather data. Our project required a total of five heuristic functions due to having three members. We created our required five, and did some additional testing on others. Our program included h1, which is hamming distance. This was calculated by checking how many tiles were not in the correct space. If a tile was not in the correct space the function would add one to the calculation. The A\* star would then pick the most promising algorithm by checking which has the least cost.

The second heuristic function that was created was h2, or Manhattan distance. This function calculated how far a tile was from its correct position. The distance from the correct position was then added as the value of h(n). The A star search evaluated the costs and picked the node with the least distance of the tiles from the correct positions .

The third heuristic function used was an admissible heuristic based off tiles being in the correct rows. This function was created to scan through the state and search each value. It would then check to see if that tile was at least in the correct row. If the tile was not in the correct row, it would add one to cost, as it would take a minimum of one move to get the tile to the correct position. If the tile was in the correct row, then no amount would be saved into the cost. A\* search would then go through checking h(n) cost and picking the states that have the most nodes in the correct rows as the goal state would want.

Another heuristic function that was created used disjoint pattern database function. This function breaks down the problem into a subproblem and uses that to make a comparison. This subproblem can be any set up tiles in the correct order to the goal states. It could also be noted, that you would not be able to combine one set being half, and then another set being the other half. If this was done it would not be effective, because changing states for one set would also affect the other. This was implemented with the values of 3,4,5,6. We wrote a function that would check to see if these values were in the correct position. If these values were not in the correct position, it would calculate the Manhattan distance of how far away they are to their respective goal state. This calculation was then saved as cost. We would then have A star to base its best search prospect off this cost.

Our last admissible heuristic function uses a method of relaxing the problem to create a calculation. This information came from the textbook on page 105, and describes different methods of relaxing the problem to build a heuristic function. We built this function by using a variation of Manhattan distance. This would typically produce lower costs, due to having the ability to calculate diagonally instead of just up, down, left ,and right movements. This new cost is was then implemented to run into A\* search algorithm.

Another heuristic function we ran was a non-admissible function to gain more information about the effect of non-admissible heuristics. Typically, A star needs an admissible function, to be classified as an optimal search method. This was implemented so we could see the difference of how it can perform compared to other heuristic functions that are admissible. This function calculated h(n) by computing the following, h(n) =h1(n) + 3 \* s(n). In other words, this uses hamming distance added to three times the value of s(n). S(n) was calculated by adding one to cost if there was a number in the middle of the state. The function would then work in a circular motion. The function would add two to cost for any tile that is not followed by its correct successor. This was then all calculated and passed to A star to run the search.

**Source Code Implementation**

Our source code was implemented using constructors that would create different goal states based off possible moves. We also built functions to implement and guide which moves are possible from each state. We saved the goal states into a variable that can be compared against.

We used several classes, switches, and functions to implement each heuristic function. This was made easy because in order to switch heuristic function, we would just choose a different option for the switch. We attempted to make our code as efficient as possible using data structures fit for each scenario. Some functions limited the ability of loops due to the goal states not being in consecutive order of numbers, but that was worked around use if/else statements. Several parts of out code were split up, worked on by different people, and then placed back together. This required work to make sure any new modification would function smoothly with all preexisting code. Our function ended up working smoothly for each heuristic function. Some functions were added near the end to implement functions that would be able to calculate data measurements of performance.

**Source Code**

**Main.cpp**

#include <iostream>

#include <vector>

#include <ctime>

#include <fstream>

#include "State.h"

#include "Neighbors.h"

#include "Node.h"

#include "Solver.h"

#include <chrono>

using namespace std::chrono;

void memory\_process(double &vm)

{

vm = 0.0;

unsigned long vsize;

{

std::string ignore;

std::ifstream ifs("/proc/self/stat", std::ios\_base::in);

ifs >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore

>> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore

>> ignore >> ignore >> vsize;

}

vm = vsize / 1024.0;

}

int main(int argc, char \*argv[])

{

int timer = (clock() \* 1000) / CLOCKS\_PER\_SEC;

int runtime = 0;

double vm;

Neighbors g;

auto startClock = high\_resolution\_clock::now();

State goal(3, std::vector<int>{1, 2, 3, 8, 0, 4, 7, 6, 5});

// CHANGE BELOW TO CHANGE INITIAL STATE

// State start(3, std::vector<int>{2, 8, 3 ,1, 6, 4, 0, 7, 5});

State start(3, std::vector<int>{2, 1, 6, 4, 0, 8, 7, 5, 3});

//\*\*\*\*\*\*\*\*CHANGE ME TO RUN DIFFERENT SOLVER ALGORITHM\*\*\*\*\*\*\*//

// Change Solver:: (heuristic you would like to run)

// Options:Diagonal,ASTAR\_H1,ASTAR\_H2,rowsHeuristic,patternDatabase

std::shared\_ptr<Node> node;

Solver solver(start, goal, Solver::ASTAR\_H1);

if (!solver.isSolvable())

{

std::cout << "Puzzle state is unsolvable..!\n";

return 0;

}

while (!solver.isSolved())

{

node = solver.GetNextNode();

solver.ExpandNode(node, g);

}

// accumulate the nodes for the solution.

std::vector<NodePtr> solution;

NodePtr s = node;

do

{

solution.push\_back(s);

s = s->GetParent();

} while (s != NULL);

// Get duration. Substart timepoints to get duration. To cast it to proper unit

// use duration cast method

auto stopClock = high\_resolution\_clock::now();

auto duration = duration\_cast<microseconds>(stopClock - startClock);

// print the solution.

std::cout <<endl<< "The puzzle can be solved in " << solution.size() - 1 << " steps. Solution below\n";

std::cout <<endl<< "PERFORMANCE DATA: " << endl;

std::cout << "Execution Time (ET): " << duration.count() << " microseconds\n";

std::cout << "Number of Nodes Generated (NG): " << solver.countNG << endl;

std::cout << "Number of Nodes Expanded (NE): " << solver.countNE << endl;

std::cout << "Depth of Tree (d): " << solver.depth << endl;

std::cout << "Effective Branching Factor b\* (b\*): " << solver.countNG / solver.depth << endl;

memory\_process(vm);

std::cout << "Memory used " << vm << " \n" <<endl;

cout << "The path taken for solution was: " <<endl<<endl;

cout << "Initial State" <<endl;

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

{

solution[i]->GetState().print(std::cout, false);

}

/\*

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

{

solution[i]->GetParent()->print(std::cout, false);

}\*/

cout << "Goal State" <<endl<<endl;

cout << "Implemeted for Artificial Intelligence 2020"<<endl;

cout << "By group - Erik pantoja, Shervone Mays, Madison Crowns" <<endl;

std::cout << "\n";

return 0;

}

**State.h**

|  |
| --- |
| // |
|  | // Created by Shervone Mayes on 4/6/2020. |
|  | // |
|  |  |
|  | #ifndef PROJECT2\_STATE\_H |
|  | #define PROJECT2\_STATE\_H |
|  |  |
|  | #include <vector> |
|  | #include <random> |
|  | #include <map> |
|  | #include <iostream> |
|  | #include <cassert> |
|  | #include <algorithm> |
|  |  |
|  | using namespace std; |
|  |  |
|  | //type definitions |
|  | typedef std::vector<int> IntArray; |
|  |  |
|  | //State class represents a unique combination of tiles |
|  | class State |
|  | { |
|  | private: |
|  | IntArray \_state; //represent tiles on puzzle; unique combination of tiles |
|  | unsigned int \_rows\_or\_cols; //defines puzzle dimensions (e.g. want 3x3 so value is 3) |
|  |  |
|  | public: |
|  | //constructor: default accepts value to define puzzle dimension |
|  | explicit State(unsigned int rows\_or\_cols) : \_rows\_or\_cols(\_rows\_or\_cols) |
|  | { |
|  | \_state.resize(\_rows\_or\_cols \* rows\_or\_cols); |
|  | for (unsigned int i = 0; i < \_state.size(); i++) |
|  | { |
|  | \_state[i] = i; |
|  | } |
|  | } |
|  |  |
|  | //constructor: that takes in the num\_rows\_or\_cols and an initial state of the array |
|  | State(unsigned int rows\_or\_cols, const IntArray &arr) : \_rows\_or\_cols(rows\_or\_cols) |
|  | { |
|  | assert(arr.size() == \_rows\_or\_cols \* rows\_or\_cols); |
|  | \_state = arr; |
|  | } |
|  |  |
|  | //constructor: copy |
|  | State(const State &other) |
|  | { |
|  | \_rows\_or\_cols = other.\_rows\_or\_cols; |
|  | \_state = other.\_state; |
|  | } |
|  |  |
|  | //assignment operator |
|  | State &operator=(const State &other) |
|  | { |
|  | if (this != &other) |
|  | { |
|  | \_rows\_or\_cols = other.\_rows\_or\_cols; |
|  | \_state = other.\_state; |
|  | } |
|  | return \*this; |
|  | } |
|  |  |
|  | //equal to operator |
|  | friend bool operator==(const State &a, const State &b) |
|  | { |
|  | return (a.\_state == b.\_state); |
|  | } |
|  |  |
|  | //not equal to operator |
|  | friend bool operator!=(const State &a, const State &b) |
|  | { |
|  | return (a.\_state != b.\_state); |
|  | } |
|  |  |
|  | //determine the index of blank tile. iterate until determine index of empty tile |
|  | inline int FindEmptyTileIndex() const |
|  | { |
|  | for (int i = 0; i < \_state.size(); i++) |
|  | { |
|  | if (\_state[i] == 0) |
|  | { return i; } |
|  | } |
|  | return (int) \_state.size(); |
|  | } |
|  |  |
|  | //slide empty tile |
|  | inline void SwapWithEmpty(int i0, int i1) |
|  | { |
|  | int tmp = \_state[i1]; |
|  | \_state[i1] = \_state[i0]; |
|  | \_state[i0] = tmp; |
|  | } |
|  |  |
|  | //randomize state of puzzle |
|  | inline void Randomize() |
|  | { |
|  | random\_shuffle(\_state.begin(), \_state.end()); |
|  | } |
|  |  |
|  | //get array state |
|  | inline const IntArray &GetArray() const |
|  | { |
|  | return \_state; |
|  | } |
|  |  |
|  | //set array state |
|  | void SetArray(const IntArray &arr) |
|  | { |
|  | \_state = arr;; |
|  | } |
|  |  |
|  |  |
|  | inline unsigned int GetNumRowsOrCols() const |
|  | { |
|  | return \_rows\_or\_cols; |
|  | } |
|  |  |
|  | //print state |
|  | void print(ostream &str, bool flat = false) const |
|  | { |
|  | for (unsigned int i = 0; i < \_rows\_or\_cols; ++i) |
|  | { |
|  | for (unsigned int j = 0; j < \_rows\_or\_cols; ++j) |
|  | { |
|  | unsigned int index = i \* \_rows\_or\_cols + j; |
|  | if (flat) |
|  | { |
|  | str << \_state[index]; |
|  | } |
|  | else |
|  | { |
|  | str << \_state[index] << " "; |
|  | } |
|  | } |
|  | if (!flat) |
|  | { |
|  | str << "\n"; |
|  | } |
|  | } |
|  | str << "\n"; |
|  | } |
|  |  |
|  | }; |
|  |  |
|  |  |
|  | #endif //PROJECT2\_STATE\_H |

**State.cpp**

|  |
| --- |
| // |
|  | // Created by WYNI on 4/6/2020. |
|  | // |
|  |  |
|  | #include "State.h" |

**Solver.h**

//

// Created by WYNI on 4/10/2020.

//

#ifndef PROJECT2\_SOLVER\_H

#define PROJECT2\_SOLVER\_H

#include <vector>

#include "State.h"

#include "Neighbors.h"

#include "Node.h"

inline bool isInArray(const State& state, const std::vector<std::shared\_ptr<Node> >& values)

{

unsigned int i = 0;

for (; i < values.size(); ++i)

{

if (state == values[i]->GetState())

{

return true;

}

}

return false;

}

//(h1) : number of misplaced tiles for \_state

inline int GetHammingCost(const State& st)

{

int cost = 0;

const IntArray& state = st.GetArray();

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

{

if (state[i] == 0)

{

continue;

}

if (state[i] != i + 1)

{

cost += 1;

}

}

return cost;

}

//(h2): sum of the distances of each tile from desired goal state

inline int GetManhattanCost(const State& st)

{

int cost = 0;

const IntArray& state = st.GetArray();

unsigned int rows\_or\_cols = st.GetNumRowsOrCols();

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

{

int v = state[i];

if (v == 0)

{

continue;

}

// actual index of v should be v-1

v = v - 1;

int gx = v % rows\_or\_cols;

int gy = v / rows\_or\_cols;

int x = i % rows\_or\_cols;

int y = i / rows\_or\_cols;

int mancost = abs(x - gx) + abs(y - gy);

cost += mancost;

int z = 0;

}

return cost;

}

/\*

inline int GetThreshold(const std::shared\_ptr<Node> &n1)

{

{

const State &state1 = n1->GetState();

int cost1 = GetManhattanCost(state1) + n1->GetDepth();

std::cout << " h1(n):" << GetHammingCost(state1) << " g(n):" << n1->GetDepth() << " f(n): "

<< GetHammingCost(state1) + n1->GetDepth() << endl;

return cost1;

}

}\*/

/\*

inline int Search(std::shared\_ptr<Node> node, int gScore, int threshold)

{

{

std::shared\_ptr<Node> node;

const State &state1 = node->GetState();

int gScore = node->GetDepth();

int f = gScore + GetManhattanCost(state1);

if (f > threshold)

{

return f;

}

{

}

}

}\*/

// ERIK ADDED HERE BELOW

// Function to Calculate S(n) to be implemented into the A\* Algorithm

// Rules are to Check around non-central squares - alloting 2 for each tile not followed by its successor

// alotting 0 for every other tile

// Piece in the center scores one

inline int calculateSofN(const State& st) // Non-admissable heuristic function

{

int SofN = 0; // creating variable to hold S(n)

const IntArray& state = st.GetArray(); // Creating reference to the array

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

{

// Multiple Rules, and not saved in order due to goal state -Using if/else

// Check successors in a circle to follow function

if (i == 0)

{ // Pos 0

if (state[0] + 1 != state[1])

{ // Checks if pos 0 is 1 less then pos 1

SofN = SofN + 2;

} // Adds 2

}

else

{

if (i == 1)

{ // Pos 1

if (state[1] + 1 != state[2])

{ // Checks if pos 1 is 1 less then pos 2

SofN = SofN + 2;

}

}

else

{

if (i == 2)

{ // Pos 2 // Checking Down as circular function

if (state[2] + 1 != state[5])

{ // Checks if pos 2 is 1 less then pos 5

SofN = SofN + 2;

}

}

else

{

if (i == 3)

{ // Pos 3

if (state[3] + 1 != state[0])

{ // Checks if pos 0 is 1 less then pos 3

SofN = SofN + 2;

}

}

else

{

if (i == 4)

{ // MIDDLE SQUARE GETS 1 if not 0/blank

if (state[i] != 0)

{ // Middle is not the blank

SofN = SofN + 1;

}// Add one for middle

}

else

{

if (i == 5)

{ // Pos 5

if (state[5] + 1 != state[8])

{ // Checks if pos 5 is 1 more then pos 8

SofN = SofN + 2;

}

}

else

{

if (i == 6)

{ // Pos 6

if (state[6] + 1 != state[3])

{ // Checks if pos 6 is 1 less then pos 3

SofN = SofN + 2;

}

}

else

{

if (i == 7)

{ // Pos 7

if (state[7] + 1 != state[6])

{ // Checks if pos 7 is 1 more then pos 8

SofN = SofN + 2;

}

}

else

{

if (i == 8)

{ // Pos 8

if (state[8] + 1 != state[7])

{ // Checks if pos 7 is 1 more then 8

SofN = SofN + 2;

}

}

}

}

}

}

}

}

}

}

}

return SofN;

}

// Function to calculate admissable heuristic based off a subproblem. This function uses the pattern database method

// to create heuristic. Similar to manhattan distance - but can be more effective only using subproblem calculations

// Set pattern of a subproblem of correct values, then calculate distances of to get to that pattern

// For this pattern we will use subproblem of 3,4,5,6 in correct respective positions

inline int calculatePatternDatabase(const State& st) // Subproblem Admissable heuristic function

{

int cost = 0; // creating variable to hold cost

const IntArray& state = st.GetArray(); // Creating reference to the array

unsigned int rows\_or\_cols = st.GetNumRowsOrCols();

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

{

// Locate 3,4,5,6

// Calculate distance from correct positions

// Save that item as cost

if (state[i] == 3)

{

if (i != 2) // If correct position - does not add to cost

{

int v = state[i]; // Calculation of distance for this only pattern\*

int gx = v % rows\_or\_cols;

int gy = v / rows\_or\_cols;

int x = i % rows\_or\_cols;

int y = i / rows\_or\_cols;

int mancost = abs(x - gx) + abs(y - gy);

cost += mancost; // Creating Cost

}

}

else

{

if (state[i] == 4)

{

if (i != 5) // If correct position - does not add to cost

{

int v = state[i]; // Calculation of distance for this only pattern\*

int gx = v % rows\_or\_cols;

int gy = v / rows\_or\_cols;

int x = i % rows\_or\_cols;

int y = i / rows\_or\_cols;

int mancost = abs(x - gx) + abs(y - gy);

cost += mancost; // Creating Cost

}

}

else

{

if (state[i] == 5)

{

if (i != 8) // If correct position - does not add to cost

{

int v = state[i]; // Calculation of distance for this only pattern\*

int gx = v % rows\_or\_cols;

int gy = v / rows\_or\_cols;

int x = i % rows\_or\_cols;

int y = i / rows\_or\_cols;

int mancost = abs(x - gx) + abs(y - gy);

cost += mancost; // Creating Cost

}

}

else

{

if (state[i] == 6)

{ // Checking value 6 in correct position

if (i != 7) // If correct position - does not add to cost

{

int v = state[i]; // Calculation of distance for this only pattern\*

int gx = v % rows\_or\_cols;

int gy = v / rows\_or\_cols;

int x = i % rows\_or\_cols;

int y = i / rows\_or\_cols;

int mancost = abs(x - gx) + abs(y - gy);

cost += mancost; // Creating Cost

}

}

else

{

continue;

}

}

}

}

}

return cost;

}

// A heuristic function for 8 square to calculate if numbers are in the correct rows or not

// This function will be admissable because the cost of not being in the correct row would not be overestimated

// Function will search each value, and then do a check if its in the correct row

// If the value is not in the correct row, we will add a cost of 1 - as it would have to move at least 1 to get to correct position

inline int heuristicOfRows(const State& st)

{

// Admissable - does not over estimate cost

int cost = 0;

const IntArray& state = st.GetArray();

for (unsigned int i = 0; i < state.size(); ++i) // Going through full state

{

// Multiple Rules, and not saved in order due to goal state -Using if/else vs loop

if (i == 0)

{ // Pos 0

if (state[3] && state[4] && state[5] != 0)

{ // Checks if 0 is in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 1)

{ // 1

if (state[0] && state[1] && state[2] != 1)

{ // Checks if 1 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 2)

{ // 2

if (state[0] && state[1] && state[2] != 2)

{ // Checks if 2 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 3)

{ // 3

if (state[0] && state[1] && state[2] != 3)

{ // Checks if 3 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 4)

{ // 4

if (state[3] && state[4] && state[5] != 4)

{ // Checks if 4 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 5)

{ // 5

if (state[6] && state[7] && state[8] != 5)

{ // Checks if 5 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 6)

{ // 6

if (state[6] && state[7] && state[8] != 6)

{ // Checks if 6 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 7)

{ // 7

if (state[6] && state[7] && state[8] != 7)

{ // Checks if 7 in in the correct ROW

cost = cost + 1;

}

}

else

{

if (i == 8)

{ // 8

if (state[3] && state[4] && state[5] != 8)

{ // Checks if 8 in in the correct ROW

cost = cost + 1;

}

}

}

}

}

}

}

}

}

}

}

return cost;

}

//ERIK ADDED HERE ABOVE

// Madison Start

inline int GetDiagonalCost(const State& st)

{

int cost = 0;

int temp = 0;

const IntArray& state = st.GetArray();

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

{

temp = state[i];

if (temp == 0)

{

continue;

}

if (temp == 1)

{

if (i == 0)

{

continue;

}

if (i == 1 || i == 4 || i == 3)

{

cost = cost + 1;

}

else

{

cost = cost + 2;

}

}

if (temp == 2)

{

if (i == 1)

{

continue;

}

if (i == 6 || i == 7 || i == 8)

{

cost = cost + 2;

}

else

{

cost = cost + 1;

}

}

if (temp == 3)

{

if (i == 2)

{

continue;

}

if (i == 1 || i == 4 || i == 5)

{

cost = cost + 1;

}

else

{

cost = cost + 2;

}

}

if (temp == 4)

{

if (i == 3)

{

continue;

}

if (i == 2 || i == 5 || i == 8)

{

cost = cost + 2;

}

else

{

cost = cost + 1;

}

}

if (temp == 5)

{

if (i == 4)

{

continue;

}

else

{

cost = cost + 1;

}

}

if (temp == 6)

{

if (i == 5)

{

continue;

}

if (i == 0 || i == 3 || i == 6)

{

cost = cost + 2;

}

else

{

cost = cost + 1;

}

}

if (temp == 7)

{

if (i == 6)

{

continue;

}

if (i == 7 || i == 4 || i == 3)

{

cost = cost + 1;

}

else

{

cost = cost + 2;

}

}

if (temp == 8)

{

if (i == 7)

{

continue;

}

if (i == 2 || i == 1 || i == 0)

{

cost = cost + 2;

}

else

{

cost = cost + 1;

}

}

}

return cost;

}

// Madison End

//find minimum value of the openlist elements based on GREEDYBESTFIRST algorithm

class CompareFunctorForGreedyBestFirst

{

public:

bool operator()(

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = GetManhattanCost(state1) + GetHammingCost(state1);

const State& state2 = n2->GetState();

int cost2 = GetManhattanCost(state2) + GetHammingCost(state2);

return cost1 < cost2;

}

};

class CompareFunctorForAStar\_H1

{

public:

//return bool for compare nodes f(n) value

bool operator()(const std::shared\_ptr<Node>& n1, const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = GetHammingCost(state1) + n1->GetDepth();

/\* std::cout << " h1(n):" << GetHammingCost(state1) << " g(n):" << n1->GetDepth() << " f(n): "

<< GetHammingCost(state1) + n1->GetDepth() << endl;\*/

const State& state2 = n2->GetState();

int cost2 = GetHammingCost(state2) + n2->GetDepth();

/\* std::cout << " h1(n):" << GetHammingCost(state2) << " g(n):" << n2->GetDepth() << " f(n): "

<< GetHammingCost(state1) + n2->GetDepth() << endl;\*/

return cost1 < cost2;

}

};

//find minimum value of the openlist elements based on ASTAR algorithm

//ASTAR selects node with the least f-score

class CompareFunctorForAStar\_H2

{

public:

bool operator()(

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = GetManhattanCost(state1) + n1->GetDepth();

/\* std::cout << "h2(n):" << GetManhattanCost(state1) << " g(n):" << n1->GetDepth() << " f(n): "

<< GetManhattanCost(state1) + n1->GetDepth()

<< endl;\*/

const State& state2 = n2->GetState();

int cost2 = GetManhattanCost(state2) + n2->GetDepth();

/\* std::cout << "h2(n):" << GetManhattanCost(state2) << " g(n):" << n2->GetDepth() << " f(n): "

<< GetManhattanCost(state2) + n2->GetDepth()

<< endl;\*/

return cost1 < cost2;

}

};

class CompareFunctorForAStar\_IDA

{

public:

bool operator()(std::shared\_ptr<Node>& n1, std::shared\_ptr<Node>& n2)

{

const State& state1 = n1->GetState();

int cost1 = GetManhattanCost(state1) + n1->GetDepth();

const State& state2 = n2->GetState();

int cost2 = GetManhattanCost(state2) + n2->GetDepth();

return cost1 < cost2;

}

};

class CompareFunctorForAStar

{

public:

bool operator()(

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = GetManhattanCost(state1) + GetHammingCost(state1) + n1->GetDepth();

const State& state2 = n2->GetState();

int cost2 = GetManhattanCost(state2) + GetHammingCost(state2) + n2->GetDepth();

return cost1 < cost2;

}

};

// ERIK ADDED BELOW

class CompareFunctorForAStarSN

{

public:

bool operator()(

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& SN) const

{

// Function for h(n) using S(n)

// H(n) = h1(n)+ 3\*S(n)

const State& state1 = n1->GetState();

int cost1 = GetHammingCost(state1) + 3 \* calculateSofN(state1) + n1->GetDepth();

const State& state2 = SN->GetState();

int cost2 = GetHammingCost(state2) + 3 \* calculateSofN(state2) + SN->GetDepth();

return cost1 < cost2;

}

};

class ComparePatternDatabase

{

public:

bool operator()(

// Using only a portion of manhattan distance - for subproblem

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = calculatePatternDatabase(state1) + n1->GetDepth(); // Cost is based off distance of pattern values

const State& state2 = n2->GetState();

int cost2 = calculatePatternDatabase(state2) + n2->GetDepth(); // Cost is based off distance of pattern values

return cost1 < cost2;

}

};

class CompareheuristicOfRows

{

public:

bool operator()(

// Using only a portion of manhattan distance - for subproblem

const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = heuristicOfRows(state1) + n1->GetDepth(); // Cost is based off distance of pattern values

const State& state2 = n2->GetState();

int cost2 = heuristicOfRows(state2) + n2->GetDepth(); // Cost is based off distance of pattern values

return cost1 < cost2;

}

};

// ERIK ADDED ABOVE

// Madison add

class DiagonalCompare

{

public:

bool operator()

(const std::shared\_ptr<Node>& n1,

const std::shared\_ptr<Node>& n2) const

{

const State& state1 = n1->GetState();

int cost1 = GetDiagonalCost(state1) +

n1->GetDepth(); // Cost is based off distance from home and allows for diagonals

const State& state2 = n2->GetState();

int cost2 = GetDiagonalCost(state2) +

n2->GetDepth(); // Cost is based off distance from home and allows for diagonals

return cost1 < cost2;

}

};

//Madison add end

class Solver

{

public:

enum Type

{

DEPTH\_FIRST = 0,

BREADTH\_FIRST,

GREEDY\_BEST\_FIRST,

ASTAR\_H1,

ASTAR\_H2,

//SMA,

//IDA,

AStarSN, // ERIK ADDED

patternDatabase, // ERIK ADDED

rowsHeuristic, // ERIK ADDED

Diagonal //Madison added

};

Solver(const State& start, const State& goal, Type type = Type::ASTAR\_H2)

: \_goal(goal), \_solved(false), \_type(type)

{

NodePtr root(new Node(start, 0, 0));

\_openlist.push\_back(root);

}

virtual ~Solver()

{

}

inline bool isSolved() const

{

return \_solved;

}

inline bool isSolvable() const

{

///TODO

return true;

}

/\* Was used for testing of IDA\*

inline int Search(std::shared\_ptr<Node> node, int gScore, int threshold)

{

{

std::shared\_ptr<Node> node;

const State &state1 = node->GetState();

int gScore = node->GetDepth();

int f = gScore + GetManhattanCost(state1);

int FOUND = 0;

int MAX\_INT = 100;

if (f > threshold)

{

return f;

}

if (node->GetState() == \_goal)

{

return FOUND;

}

int min = MAX\_INT;

NodePtr current;

current = \_openlist[0];

NodeList::iterator current\_itr(\_openlist.begin());

if (current\_itr == \_openlist.end())

{

return 0;

}

else

{

int temp = Search(GetNextNode(), gScore, threshold);

if (temp == FOUND)

{

return FOUND;

}

if (temp < min)

{

min = temp;

}

}

}

// Return something here

}\*/

///Returns next node in the search.

//template<class Compare> osg::ref\_ptr<Node> GetNextNode(Compare cmp)

NodePtr GetNextNode()

{

if (\_openlist.empty())

{

return 0;

}

NodePtr current;

switch (\_type)

{

//ASTAR = F(n) = g(n) + h(n)

// where g(n) is total cost to reach n along path or total cost from ground node to current node.

// where h1(n) is the number of misplaced tiles

//where h2(n) is sum of distances of tiles from their goal positions

case ASTAR\_H1:

{

NodeList::iterator current\_itr(

std::min\_element(\_openlist.begin(), \_openlist.end(), CompareFunctorForAStar\_H1()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

case ASTAR\_H2:

{

//min\_element function returns an iterator to element with the smallest value in range [\_openlist.begin(), openlist.end()) so does not include the element pointed by openlist.end()

//comp() binary function that accepts two elements in the range as arguments, returns a value convertible to bool. value returned indicates whether the element passed as first argument is less than second argument

NodeList::iterator current\_itr(

std::min\_element(\_openlist.begin(), \_openlist.end(), CompareFunctorForAStar\_H2()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

case GREEDY\_BEST\_FIRST:

{

NodeList::iterator current\_itr(std::min\_element(

\_openlist.begin(),

\_openlist.end(),

CompareFunctorForGreedyBestFirst()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

case BREADTH\_FIRST:

{

current = \_openlist[0];

\_openlist.erase(\_openlist.begin());

\_closedlist.push\_back(current);

break;

}

/\* case IDA:

{

Neighbors g;

std::shared\_ptr<Node> node;

const State &state1 = node->GetState();

int gScore = node->GetDepth();

int FOUND = 0;

current = \_openlist[0];

int threshold = GetThreshold(current);

std::cout << "Threshold: " << threshold << endl;

NodeList::iterator current\_itr(\_openlist.begin());

if (current\_itr == \_openlist.end())

{ return 0; }

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

while (1)

{

int temp = Search(node, gScore, threshold);

if (temp == FOUND)

{

return 0; //return temp

}

if (temp == INFINITY)

{

return 0; //or set time limit exceeded

}

threshold = temp;

//node = GetNextNode();

//ExpandNode(node, g);

}

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(

current); //doesn't keep track of visited node and therefore explores already explored nodes again.

break;

}\*/

case DEPTH\_FIRST:

{

//current = \_openlist[0];

NodeList::iterator current\_itr(\_openlist.begin());

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

// \*\*\*\* ERIK Added here

case rowsHeuristic:

{

NodeList::iterator current\_itr(std::min\_element(

\_openlist.begin(),

\_openlist.end(),

CompareheuristicOfRows())); // Calling function for Heuristic Of Rows

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

case patternDatabase:

{

NodeList::iterator current\_itr(std::min\_element(

\_openlist.begin(),

\_openlist.end(),

ComparePatternDatabase()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

case AStarSN:

{

NodeList::iterator current\_itr(std::min\_element(

\_openlist.begin(),

\_openlist.end(),

CompareFunctorForAStar()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

// \*\*\*\* ERIK Added here

// Madison Start

case Diagonal:

{

NodeList::iterator current\_itr(

std::min\_element(\_openlist.begin(), \_openlist.end(), DiagonalCompare()));

if (current\_itr == \_openlist.end())

{

return 0;

}

//copy the value first to a shared pointer and then erase from the open list.

current = \*current\_itr;

// now erase from the open list.

\_openlist.erase(current\_itr);

\_closedlist.push\_back(current);

break;

}

// Madison End

}

\_typePrint = \_type;

return current;

}

// expand the graph by looking into the neighbors for the given node.

void ExpandNode(NodePtr current, const Neighbors& graph)

{

if (current->GetState() == \_goal)

{

\_solved = true;

return;

}

int zero = current->GetState().FindEmptyTileIndex();

const IntArray& neighbours = graph.GetNeighbors(zero);

for (int next : neighbours)

{

State state = current->GetState();

state.SwapWithEmpty(zero, next);

if (!isInArray(state, \_closedlist))

{

NodePtr n(new Node(state, current, current->GetDepth() + 1));

\_openlist.push\_back(n);

countNG++;

static int s\_lineNum = 1;

n->print(std::cout, s\_lineNum++);

depth = current->GetDepth() + 1;

//\_closedlist.push\_back(n);

}

}

countNE = \_closedlist.size();

}

private:

typedef std::vector<NodePtr> NodeList;

NodeList \_openlist;

NodeList \_closedlist;

const State& \_goal;

bool \_solved;

Type \_type;

public:

int countNE = 0;

int countNG = 0;

int depth = 0;

Type \_typePrint;

};

#endif //PROJECT2\_SOLVER\_H

**Solver.cpp**

Top of Form

Bottom of Form

Top of Form

Bottom of Form

|  |  |
| --- | --- |
|  | // |
|  | // Created by WYNI on 4/10/2020. |
|  | // |
|  |  |
|  | #include "Solver.h" |

**Node.h**

|  |
| --- |
| // |
|  | // Created by WYNI on 4/10/2020. |
|  | // |
|  |  |
|  | #ifndef PROJECT2\_NODE\_H |
|  | #define PROJECT2\_NODE\_H |
|  |  |
|  |  |
|  | #include <memory> |
|  | #include "State.h" |
|  |  |
|  | //represent tree elements |
|  |  |
|  | class Node |
|  | { |
|  | private: |
|  | State \_state; |
|  | std::shared\_ptr<Node> \_parent; |
|  | int \_depth; |
|  |  |
|  | public: |
|  | Node(const State &state, std::shared\_ptr<Node> parent, int depth = 0) |
|  | : \_state(state), \_depth(depth) |
|  | { |
|  | \_parent = parent; |
|  | } |
|  |  |
|  | void SetParent(Node \*node) |
|  | { |
|  | \_parent.reset(node); |
|  | } |
|  |  |
|  | void SetParent(std::shared\_ptr<Node> node) |
|  | { |
|  | \_parent = node; |
|  | } |
|  |  |
|  | std::shared\_ptr<Node> GetParent() |
|  | { |
|  | return \_parent; |
|  | } |
|  |  |
|  | const std::shared\_ptr<Node> GetParent() const |
|  | { |
|  | return \_parent; |
|  | } |
|  |  |
|  | const State &GetState() const |
|  | { |
|  | return \_state; |
|  | } |
|  |  |
|  | int GetDepth() const |
|  | { |
|  | return \_depth; |
|  | } |
|  |  |
|  | void print(std::ostream &out, int lineNum) const |
|  | { |
|  | out << lineNum << " - Node { "; |
|  | for (unsigned int i = 0; i < \_state.GetArray().size(); ++i) |
|  | { |
|  | out << \_state.GetArray()[i]; |
|  | } |
|  | out << " | Depth: " << \_depth; |
|  | out << " }" << "\n"; |
|  | } |
|  | }; |
|  |  |
|  | typedef std::shared\_ptr<Node> NodePtr; |
|  |  |
|  |  |
|  | #endif //PROJECT2\_NODE\_H |

**Node.cpp**

|  |
| --- |
| // |
|  | // Created by WYNI on 4/10/2020. |
|  | // |
|  |  |
|  | #include "Node.h" |

**Neighbors.h**

|  |
| --- |
| // |
|  | // Created by WYNI on 4/6/2020. |
|  | // |
|  |  |
|  | #ifndef PROJECT2\_NEIGHBORS\_H |
|  | #define PROJECT2\_NEIGHBORS\_H |
|  | using namespace std; |
|  |  |
|  | #include <vector> |
|  | #include <random> |
|  | #include <map> |
|  | #include <iostream> |
|  | #include <cassert> |
|  | #include <algorithm> |
|  |  |
|  | // define neighbors based on where empty tile can move |
|  | class Neighbors |
|  | { |
|  | public: |
|  | typedef map<int, vector<int> > IndexNeighborMap; |
|  | IndexNeighborMap \_edges; |
|  |  |
|  | //constructor |
|  | Neighbors() |
|  | { |
|  | CreateGraphFor8Puzzle(); |
|  | } |
|  |  |
|  | //returns vector of index of neighboring tiles where empty tile can move; input parameter is index to empty tile |
|  | const vector<int> &GetNeighbors(int id) const |
|  | { |
|  | IndexNeighborMap::const\_iterator itr(\_edges.find(id)); |
|  | if (itr != \_edges.end()) |
|  | { |
|  | return itr->second; |
|  | } |
|  | static vector<int> s; |
|  | return s; |
|  | } |
|  |  |
|  | private: |
|  | //neighbors of empty tile stored in map (stores collection of pairs (key,value) also known as a dictionary whose keys are unique. |
|  | //recall two vertices are adjacent if there is an edge between them so insert edge(s) for neighboring tiles |
|  | //make\_pair() It constructs a pair object with its first element set to x and its second element set to y which is vector of neighboring tiles here. |
|  | void CreateGraphFor8Puzzle() |
|  | { |
|  | \_edges.insert(make\_pair(0, vector < int > {1, 3})); |
|  | \_edges.insert(make\_pair(1, vector < int > {0, 2, 4})); |
|  | \_edges.insert(make\_pair(2, vector < int > {1, 5})); |
|  | \_edges.insert(make\_pair(3, vector < int > {4, 0, 6})); |
|  | \_edges.insert(make\_pair(4, vector < int > {3, 5, 1, 7})); |
|  | \_edges.insert(make\_pair(5, vector < int > {4, 2, 8})); |
|  | \_edges.insert(make\_pair(6, vector < int > {7, 3})); |
|  | \_edges.insert(make\_pair(7, vector < int > {6, 8, 4})); |
|  | \_edges.insert(make\_pair(8, vector < int > {7, 5})); |
|  | } |
|  | }; |
|  |  |
|  | #endif //PROJECT2\_NEIGHBORS\_H |

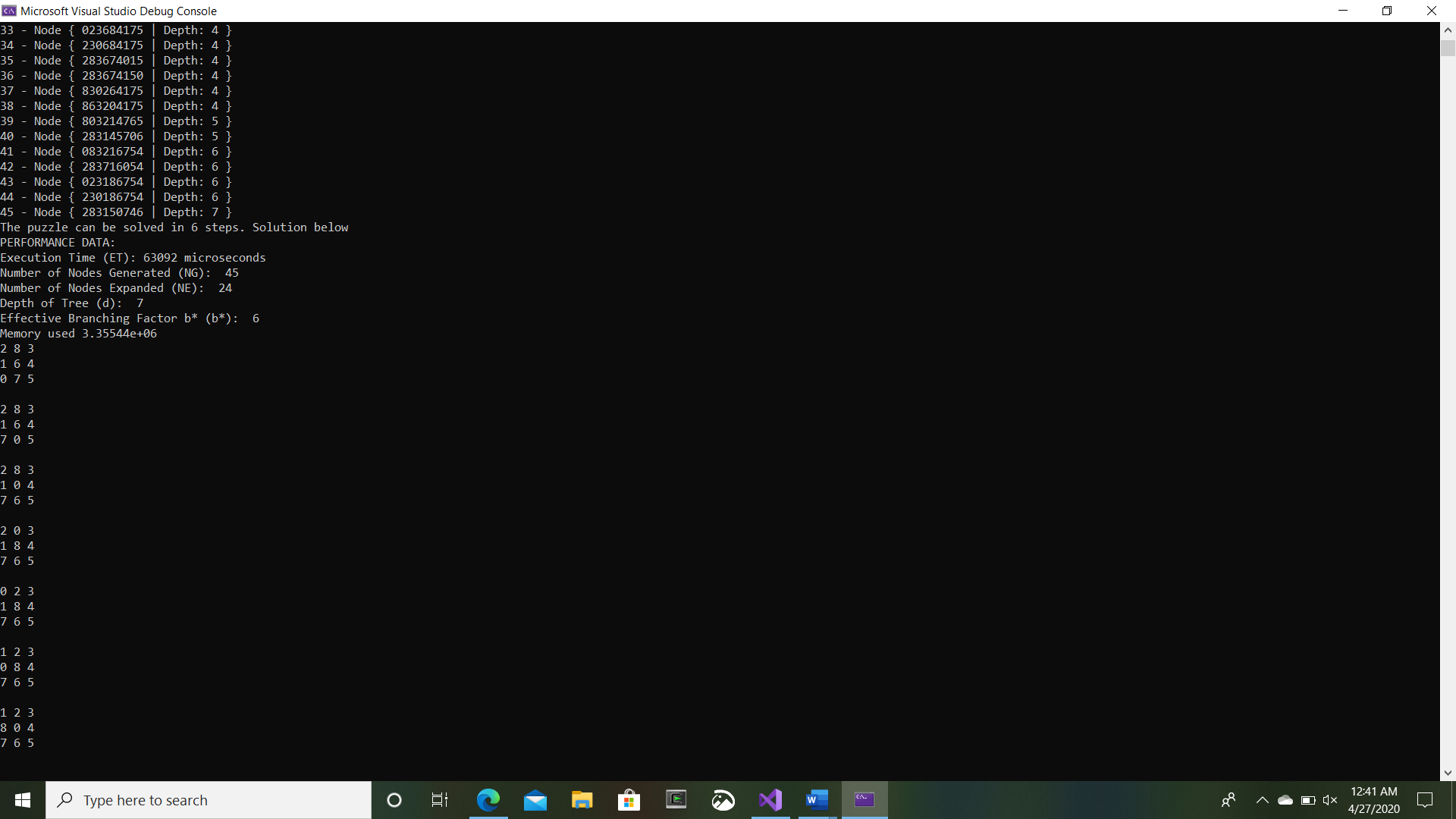
**Neighbors.cpp**

|  |
| --- |
| // |
|  | // Created by WYNI on 4/6/2020. |
|  | // |
|  |  |
|  | #include "Neighbors.h" |

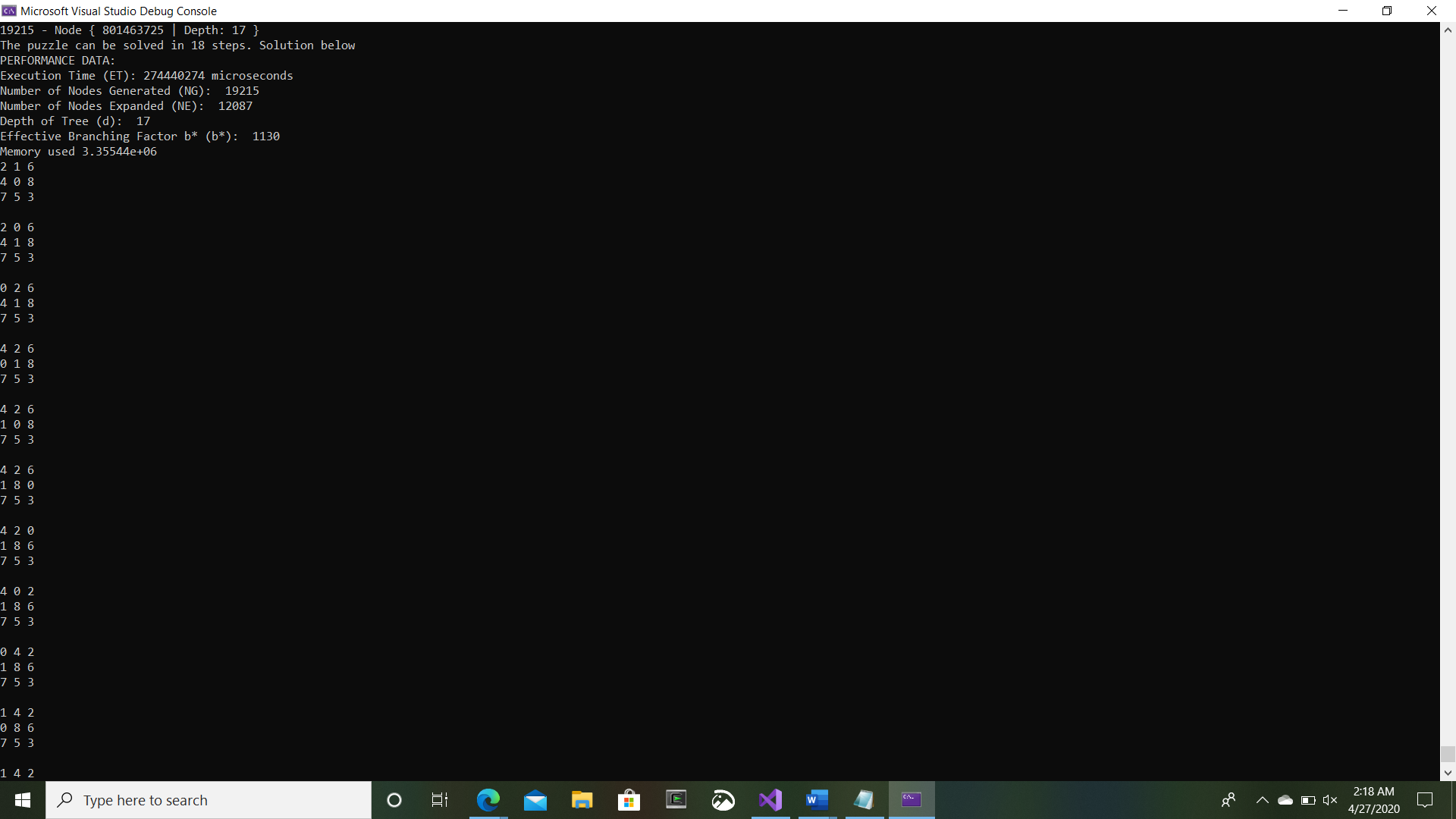
**Copy of the Program Run**

**H1**

State 1:

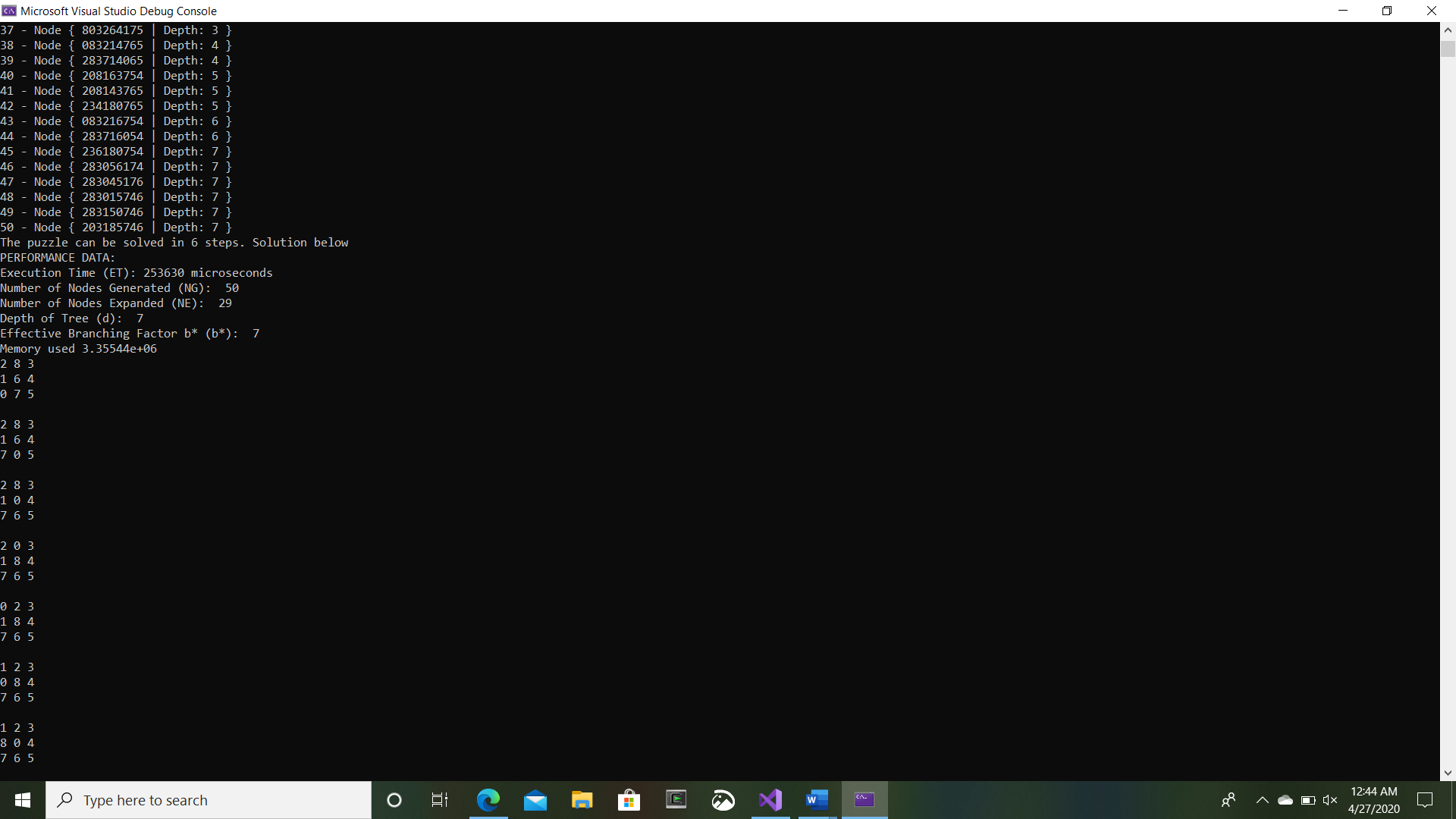


State 2:

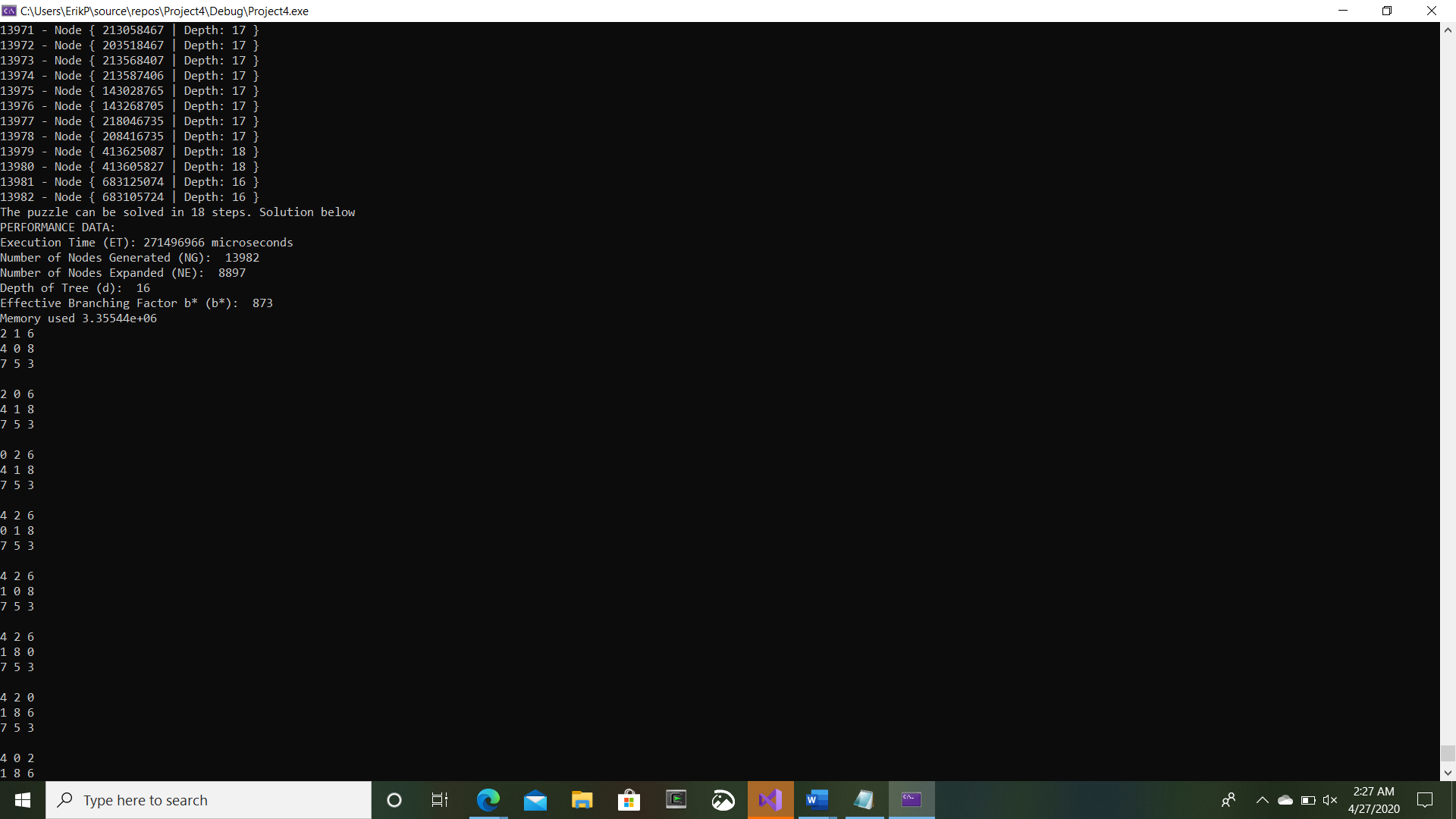


**H2**

State 1:

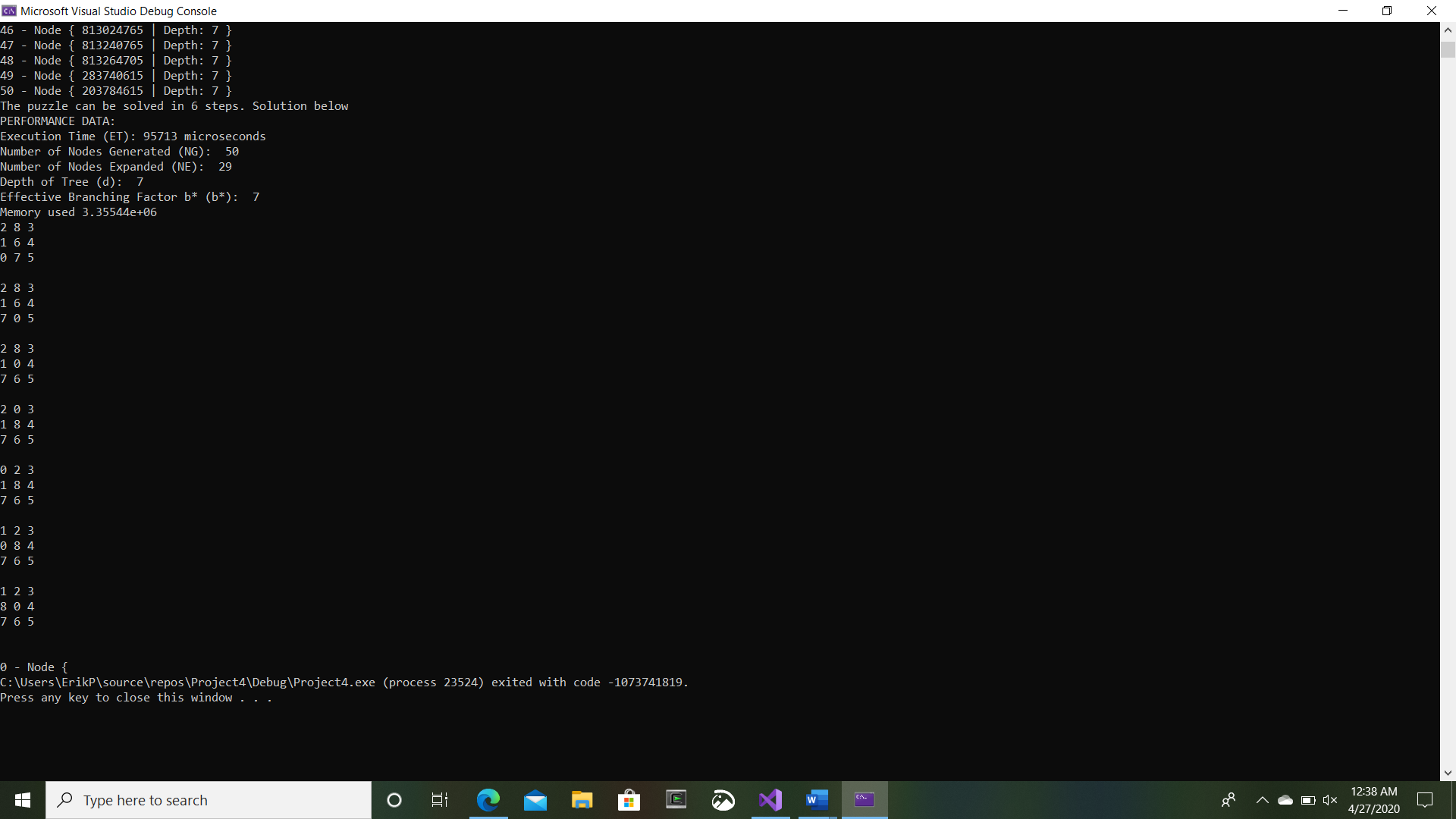


State 2:

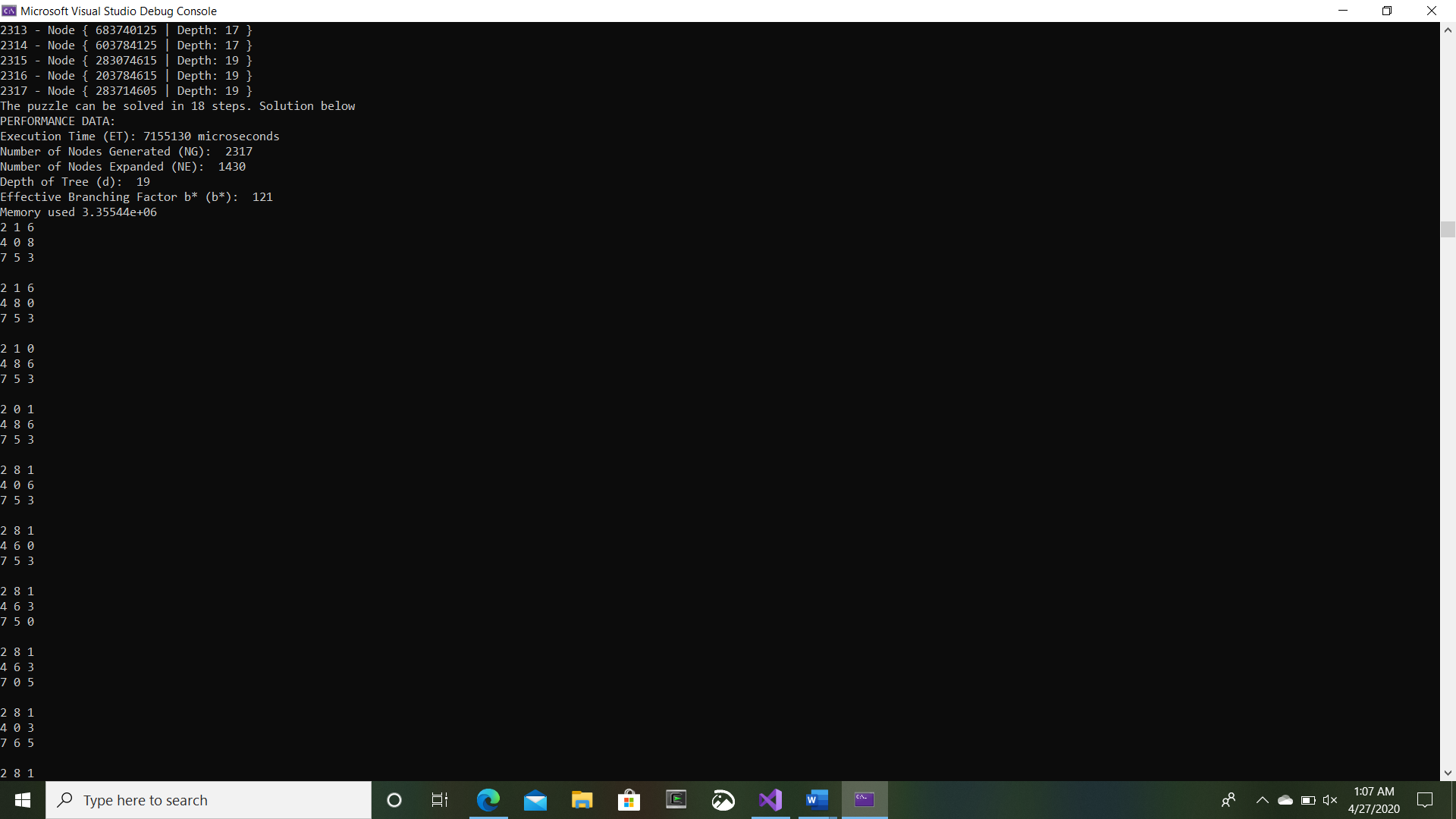


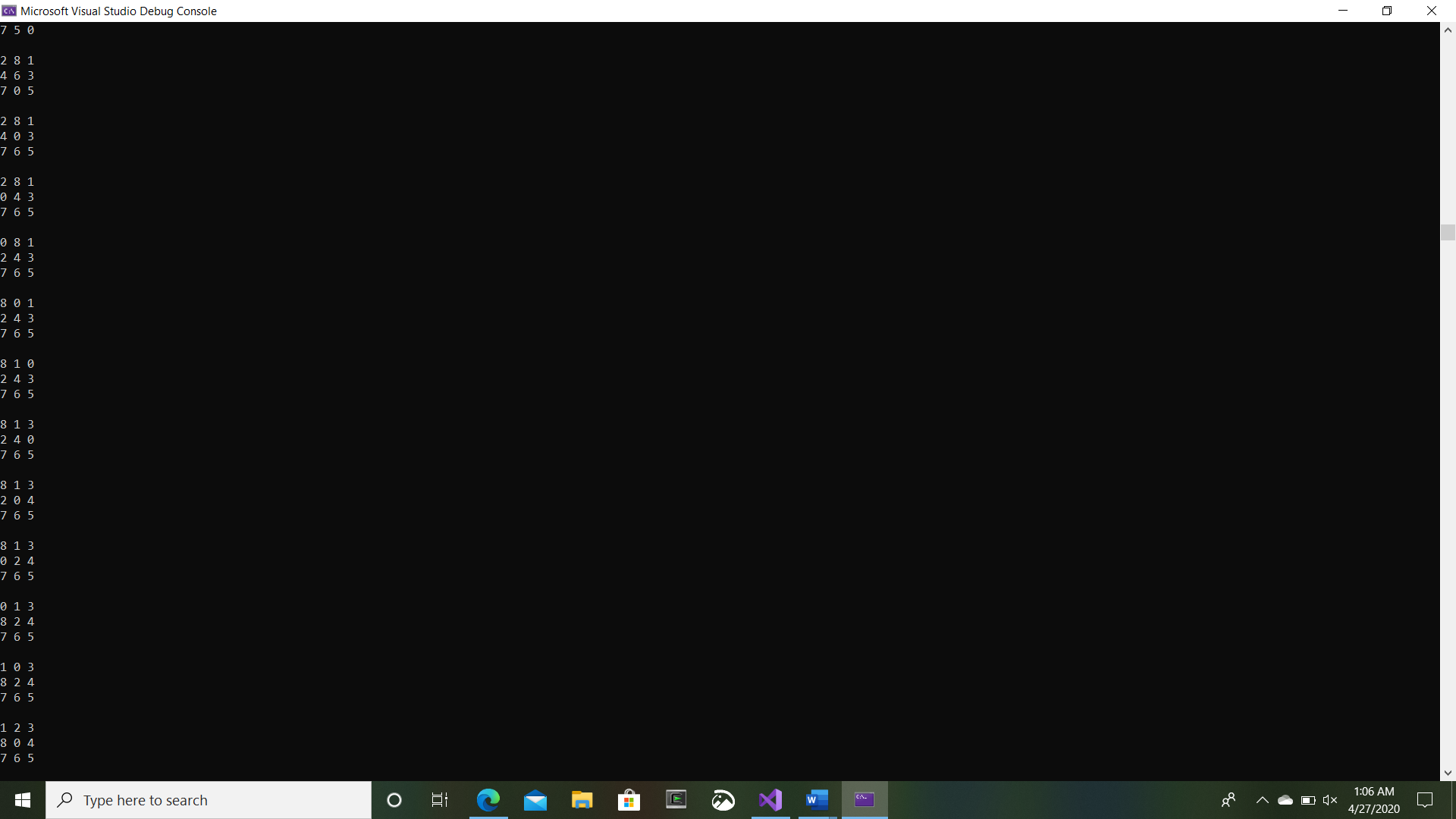
**Pattern Heuristic**

State 1:



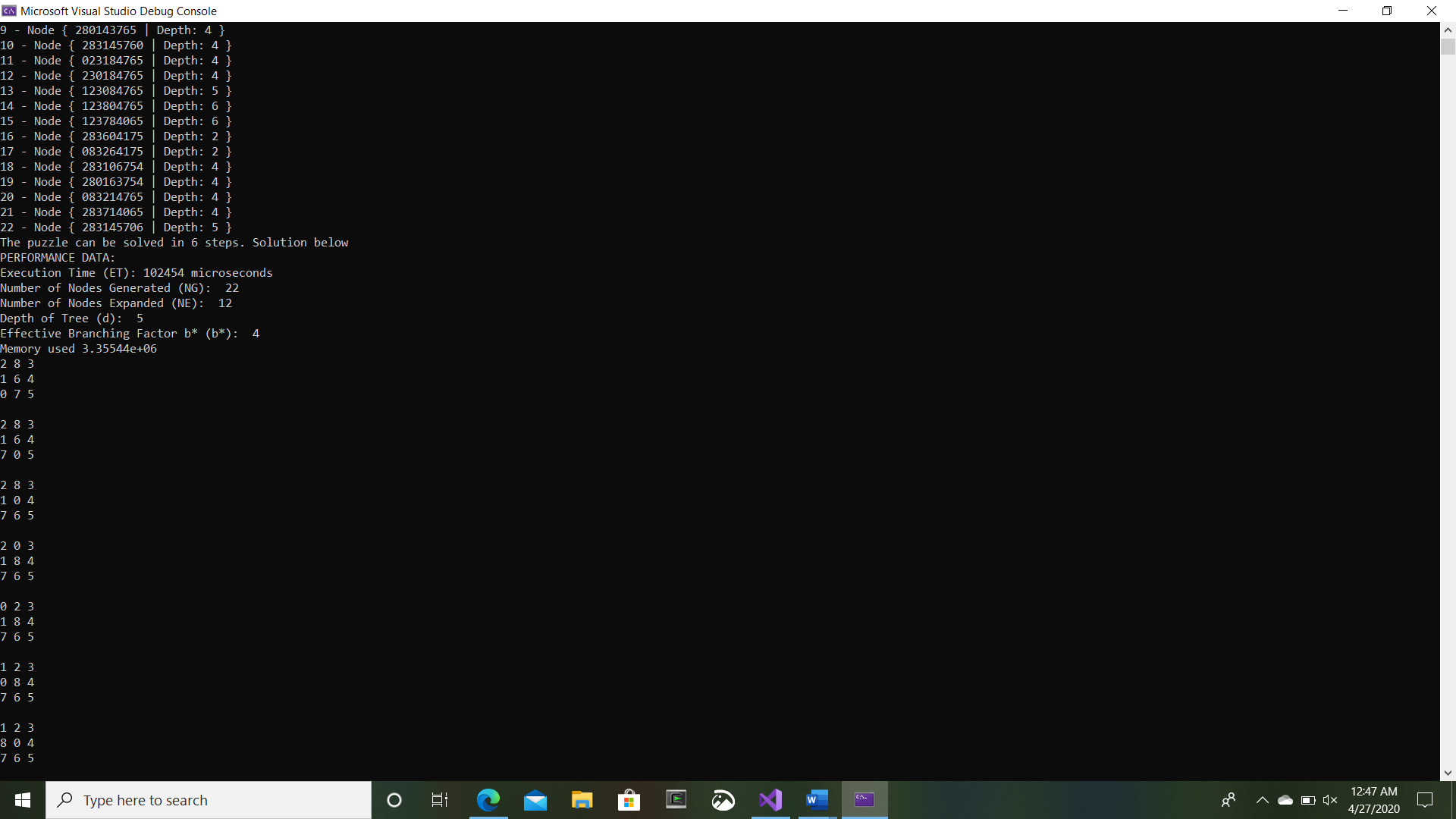
State 2:



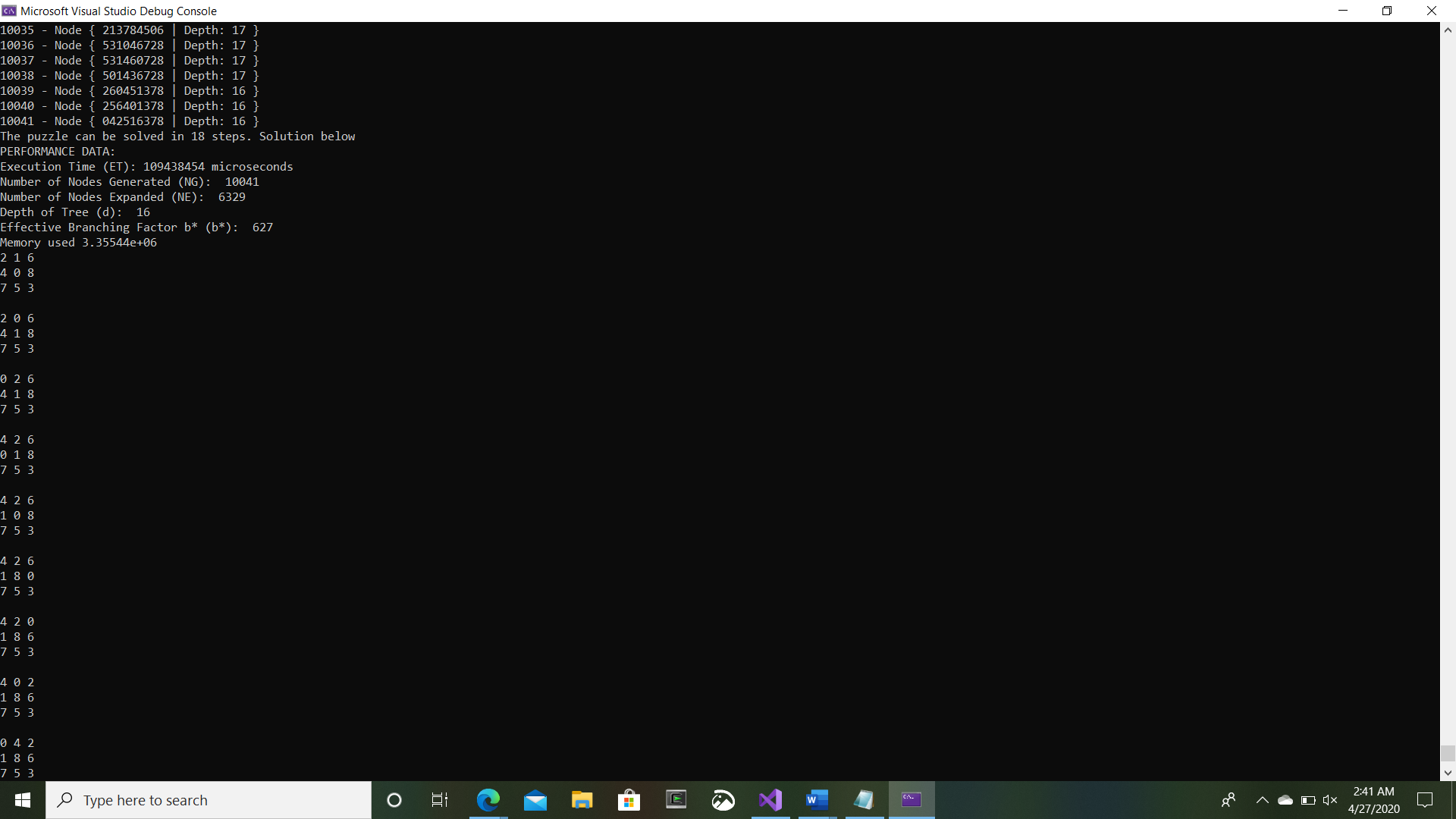


**Diagonal Heuristic**

State 1:

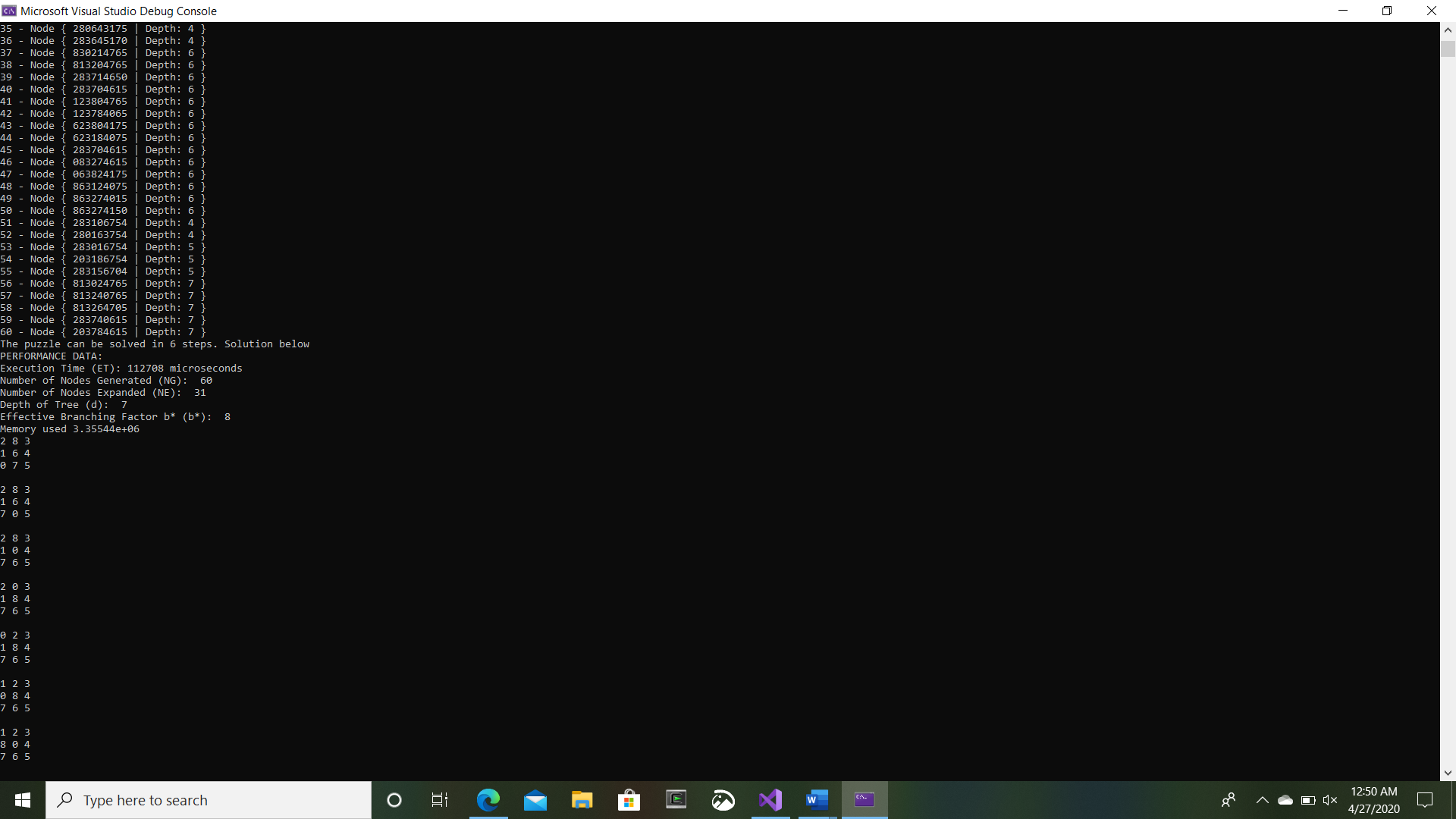


State 2:

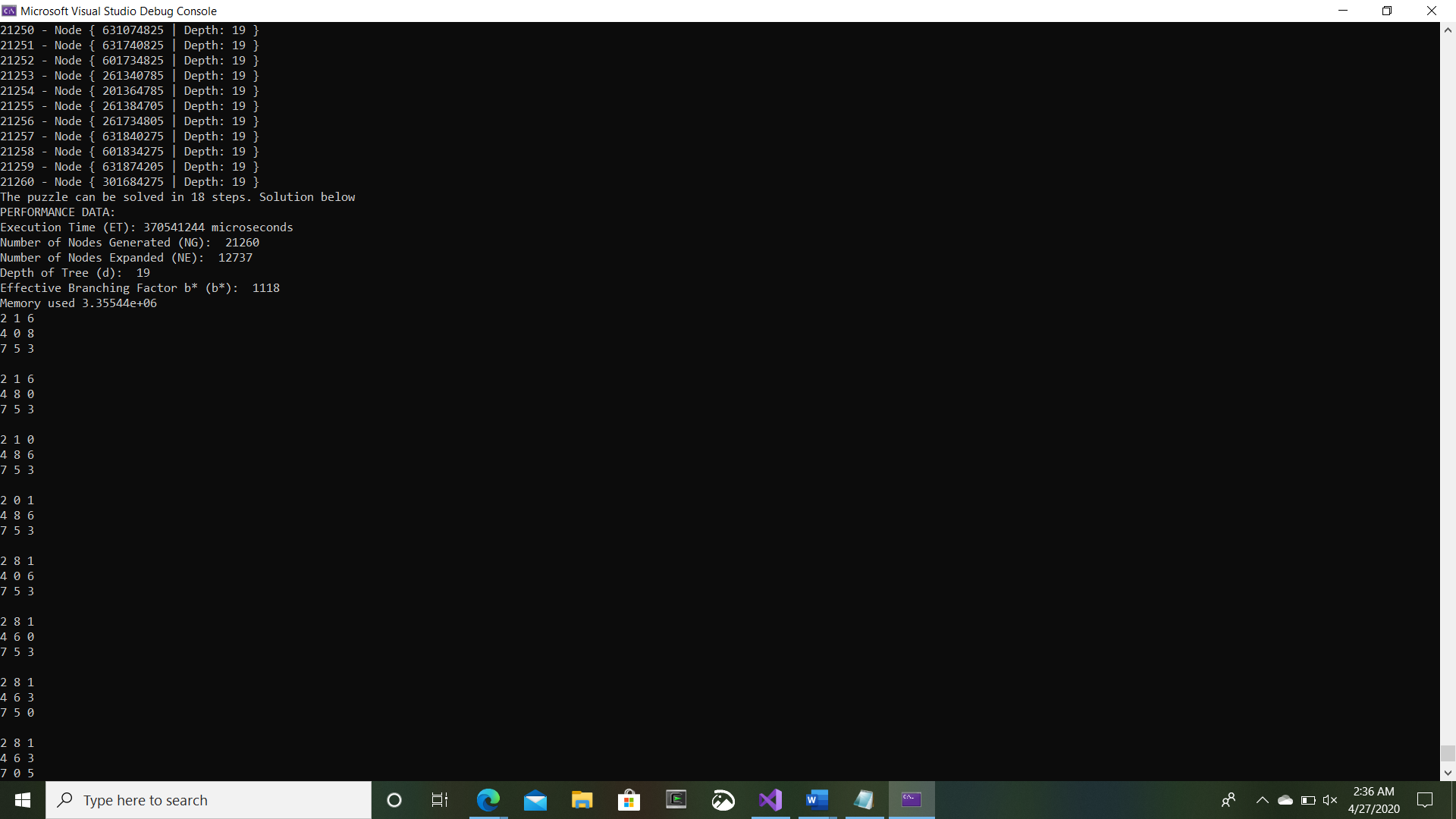


**Rows Heuristic**

State 1:



State 2:



**Analysis of the program**

The program provides key information about the implementation of A star search algorithm and several heuristics. The program was made neatly, with comments in many sections to describe functions, interactions, and purpose. The function runs smoothly, without any errors or problems. After testing the program, it was observed to work smoothly. I believe with additional time even more heuristic functions, and improvements could to conduct research on the A star algorithm.

We implemented the program so that it would be easy to switch between using different heuristic functions quickly, and easily. We used multiple files including several header files to keep the program neat, and efficient. The program provides a user-friendly output that shows information on the command line screen, including what is currently being performed, how many moves it will take to get to the goal, how long it took for the program to run and more.

We attempted to work on several additional algorithms and search methods in additional to the ones required. This provided us helpful information, and research into related algorithms and effects of heuristic functions.

We created several prototypes of the program, before we ended up with the version we eventually used for each heuristic. This allowed more ideas, to be brought forward to consider when creating this algorithm.

**Analysis of the results**

**Table 1**

Initial State #1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Heuristic Function | ET | NG | NE | d | b\* | MO |
| H1 | 63092 | 45 | 24 | 7 | 6 | 3.35544e+06 |
| H2 | 253630 | 50 | 29 | 7 | 7 | 3.35544e+06 |
| Rows Heuristic | 112708 | 60 | 31 | 7 | 8 | 3.35533e+06 |
| Pattern Heuristic | 95713 | 50 | 29 | 7 | 7 | 3.35544e+06 |
| Diagonal Heuristic | 102454 | 22 | 12 | 5 | 4 | 3.5544e+06 |

**Table 2**

Initial State #2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Heuristic Function | ET | NG | NE | d | b\* | MO |
| H1 | 274440274 | 19215 | 12087 | 17 | 1130 | 3.35544e+06 |
| H2 | 271496966 | 13982 | 8897 | 16 | 873 | 3.35544e+06 |
| Rows Heuristic | 370541244 | 21260 | 12737 | 19 | 1118 | 3.35544e+06 |
| Pattern Heuristic | 7155130 | 2317 | 1430 | 19 | 121 | 3.35544e+06 |
| Diagonal Heuristic | 109438454 | 10041 | 6329 | 16 | 627 | 3.35544e+06 |

After implementing all of our A star heuristic functions we gained many results and data that can be used for comparisons. The data we gained can help us see which functions are the fastest, most efficient, and more. According to the results the initial state 1 was the easiest for each the programs to run. They required less execution time, nodes generated, nodes expanded, depth, branching and memory. These algorithms were tested on 2 different computers to see the results myself. I noticed that the only changed when switching systems is in execution time, and memory.

According to the results provided from the programs, on average the heuristic function that uses diagonal distance, relaxing the problem was the most effective for all fields except memory. The pattern heuristic was only slightly better than the others with memory. Another key thing that was noticed, was how h2, had quite a bit better efficiency then h1 did on the calculations.

The second initial states took much longer for each search to work through. The data indicated that the pattern heuristic search provided typically results on the lower end compared to the others. The depth however was lowest with h2, and the diagonal function. This provides information that the pattern function is pretty effective.

When combining the data from both initial states it could be observed that the heuristic that were leading the others as being the most effective were the diagonal, and the pattern heuristic.

**Conclusion**

We created and tested A start using several different methods with heuristic functions, and each of these methods provided key information in learning about the different way’s artificial intelligent algorithms work, and how they can be improved upon. The program was difficult to create, however once created it was easy to see how the different heuristics would play a role in the overall effectiveness of the program. We all worked together as a group to create this program, sharing ideas, and gaining skills.

This program provided us much data about which heuristic were effective over others. One of the key differences is the amount of effectiveness that h2 has over h1. This was taught previously, however this allowed us to see real results. We also gathered information about how effective diagonal, and pattern heuristic can be when implemented with A star. I believe search functions can be very effective, especially informed searches with the right heuristic functions.

**References**

Russell, Stuart J., and Peter Norvig. Artificial Intelligence a Modern Approach. Third ed., Prentice Hall, 2010.

Rich Chapter 3 Heuristic Search (6) PowerPoint - Tracs

Non-admissible Heuristic Function.docx – Tracs

Chatterjee, Marina, et al. “A\* Search Algorithm in Artificial Intelligence: A\* (Star) Algorithm in AI.” GreatLearning, 24 Apr. 2020, [www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/](http://www.mygreatlearning.com/blog/a-search-algorithm-in-artificial-intelligence/).