

# CS205: Artificial Intelligence, Dr. Eamonn Keogh

## Project 1

Huy Dinh Tran

SID: 862325308

Email: [htran197@ucr.edu](mailto:htran197@ucr.edu)

Date: 05/13/2022

GitHub Repo: <https://github.com/huydinhtran/8-Puzzle-Solver>

In completing this assignment I consulted:

- All of the lecture slides and undergraduate video lectures from Dr. Eamonn Keogh
- C++ documentation from <https://www.cplusplus.com/>, programming Q&A forum <https://stackoverflow.com/>, and as well as programming tutorial websites such as <https://www.geeksforgeeks.org/> and <https://www.w3schools.com/>

All important core code is written by myself originally. Here is a list of mandatory and additional libraries which have functions that were used to assist me implementing the algorithm.

- `#include <stdio.h>`
- `#include <iostream>`
- `#include <algorithm>`
- `#include <queue>`
- `#include <vector>`
- `#include <list>`
- `#include <string>`
- `#include <cstdlib>`
- `#include <chrono>`

## Outline of the report:

- Cover page: Page 1 (this page)
- Report:
  - Introduction: Pages 2
  - Comparison of Algorithms: Page 3
  - Comparison of Algorithms on Sample Puzzle: Page 4 to 6
  - Conclusion: Page 7
  - Source code: Page 8 to 16

# CS205 Project 1

## The 8-Puzzle Solver

Huy Dinh Tran, SID: 862325308

### Introduction

The 8-puzzle is a 3 by 3 sliding tile puzzle that consists of 8 square tiles that are numbered from 1 to 8 and an empty tile. The tiles will be randomly shuffled by moving each square tile into the empty tile. The goal of this puzzle is to revert the board back to its original state which counts from 1 to 8 starting from row 1, column 1 down to the empty tile in row 3, column 3. Figure 1 shows an example of the initial state and the goal state.

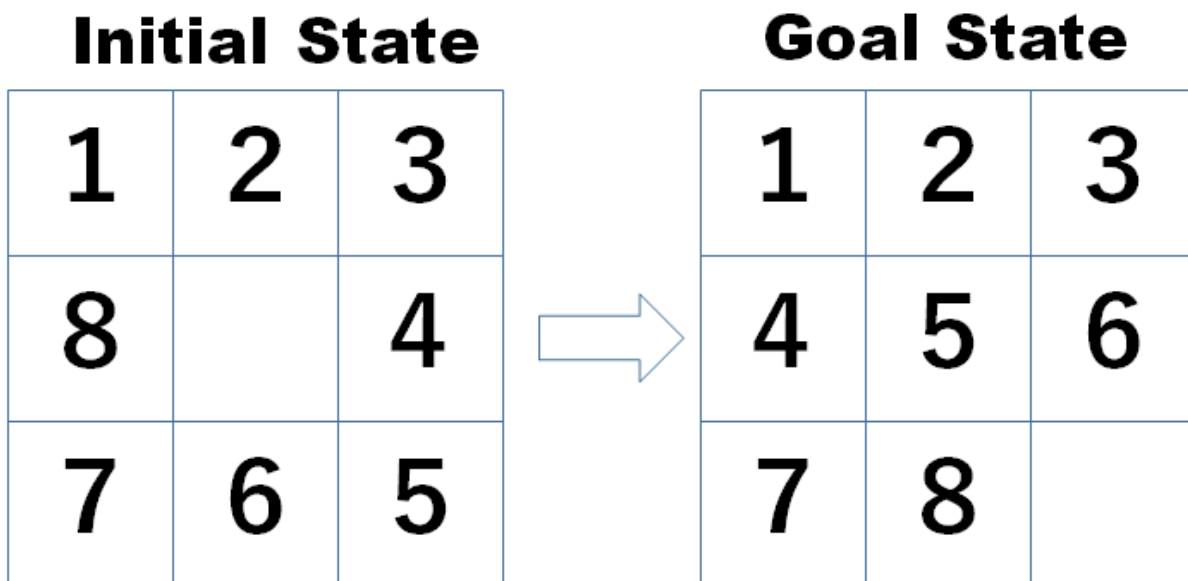


Figure 1. Initial state vs goal state of the 8-puzzle

In this project that was assigned by Dr. Eamonn Keogh, we are assigned to implement a program to solve the 8-puzzle. There are 3 main searching algorithms to implement which are Uniform Cost Search, A-star with Misplaced Tile Heuristic, and A-star with Manhattan Distance Heuristic. I implemented the algorithm using C++ and screenshots of my source code are included in the last section of this report.

# Comparison of Algorithms

## Uniform Cost Search

As mentioned in lectures and the project handout, Uniform Cost Search is similar to an A-star search but with the heuristic set as 0. We only care about the movement cost when determining which node to expand which is always 1 for each depth. With this in mind, since each movement has the same cost, we can view this search as a Breath-First Search for this particular puzzle.

## A-Star with Misplaced Tile Heuristic (Hamming)

A-Star is a very optimal and complete searching algorithm that can search straight to the goal node by using a calculated heuristic. For this version of A-star, we are using the Misplaced Tile Heuristic which compares each tile of the current board to the goal board. Each misplaced tile will increment the heuristic number, as well as the movement cost, which is also added to the equation. When selecting a node to expand, we will compare all of the heuristics of all the nodes within the queue, and the one with the smallest heuristic number will be chosen to expand.

## A-Star with Manhattan Distance Heuristic

For the Manhattan Distance Heuristic, it is quite similar to the Misplaced Tile Heuristic but it gives us a little more detail on the closeness of the state. For each tile, it calculates the minimum number of steps to reach the goal tile. For example, tile 1 should be at row 0, column 0 but the current board has tile 1 at row 2, column 2. The distance to reach the goal position is 4 which is going up 2 steps and left 2 steps. We would calculate for all of the tiles and output the heuristic number. Similar to the Misplaced Tile Heuristic, the node with the smallest heuristic will be chosen to expand. Figure 2 gives examples of the calculation of both heuristics.

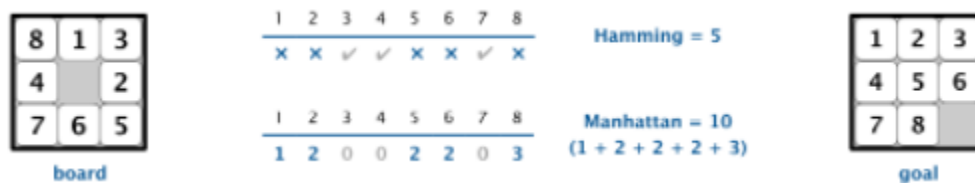


Figure 2. Calculations for Misplaced Tile and Manhattan Distance Heuristic

## Comparison of Algorithms on Sample Puzzle

I would be using the test cases provided by Dr. Eamon Keogh shown in Figure 3 to run and graph the results for comparison. For all the runs, I will eliminate printing the puzzle each iteration to give us precise execution time measurements. I included the graph and the result table below.

Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123 456 780	123 456 078	123 506 478	136 502 478	136 507 482	167 503 482	712 485 630	072 461 358

Figure 3. Provided test cases with depths

Total Node Expanded								
Depth	0	2	4	8	12	16	20	24
Uni Cost Search	0	10	997	542901	23285	542901		
A* Misplace Tile	0	5	34	295	159	908	32795	30488
A* Manhattan Distance	0	5	25	104	53	84	1763	144

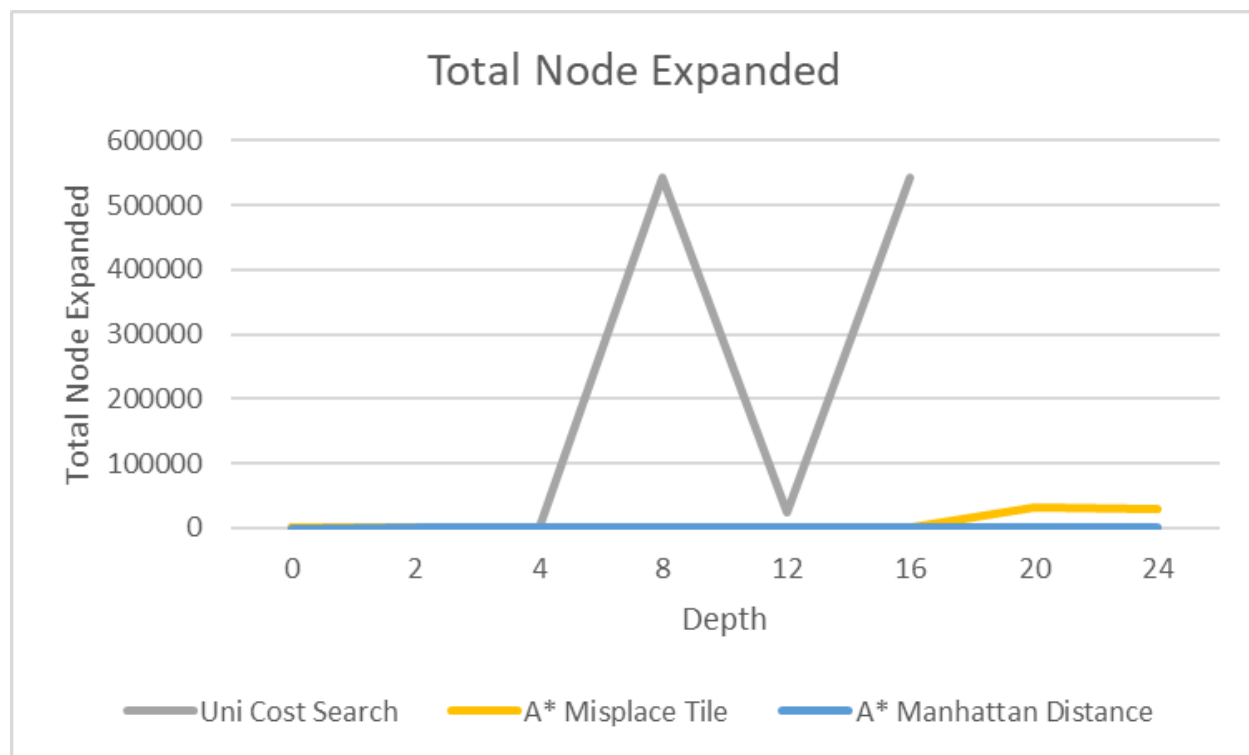


Figure 4. Total node expanded comparison between 3 searching algorithms

Total Queue Size								
Depth	0	2	4	8	12	16	20	24
Uni Cost Search	1	8	792	430464	18464	430464		
A* Misplace Tile	1	4	25	229	132	721	26391	23412
A* Manhattan Distance	1	4	20	82	45	71	1385	119

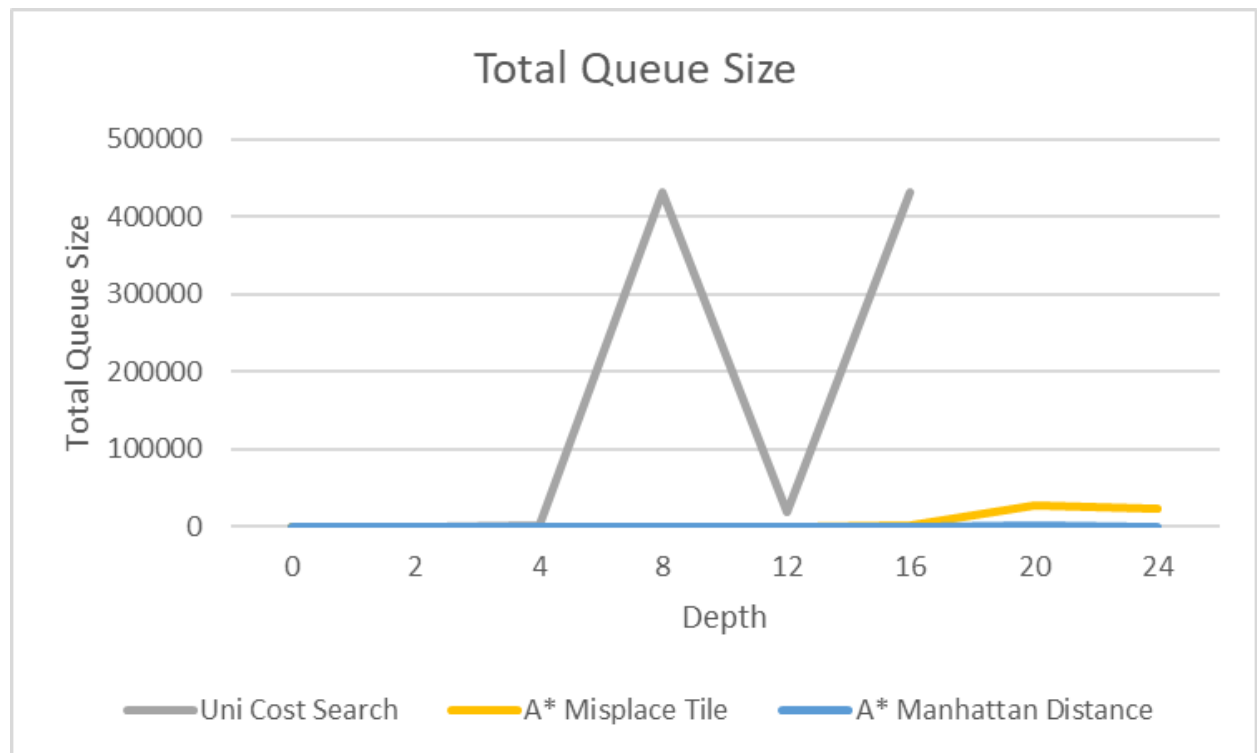


Figure 5. Total queue size comparison between 3 searching algorithms

Execution Time (in ms)								
Depth	0	2	4	8	12	16	20	24
Uni Cost Search	0.3	0.3	1.7	45243.4	108.932	45280.3		
A* Misplace Tile	0.3	0.3	0.4	0.6	0.5	1.3	40.9	40.7
A* Manhattan Distance	0.4	0.3	0.4	0.4	0.4	0.4	2.1	0.6

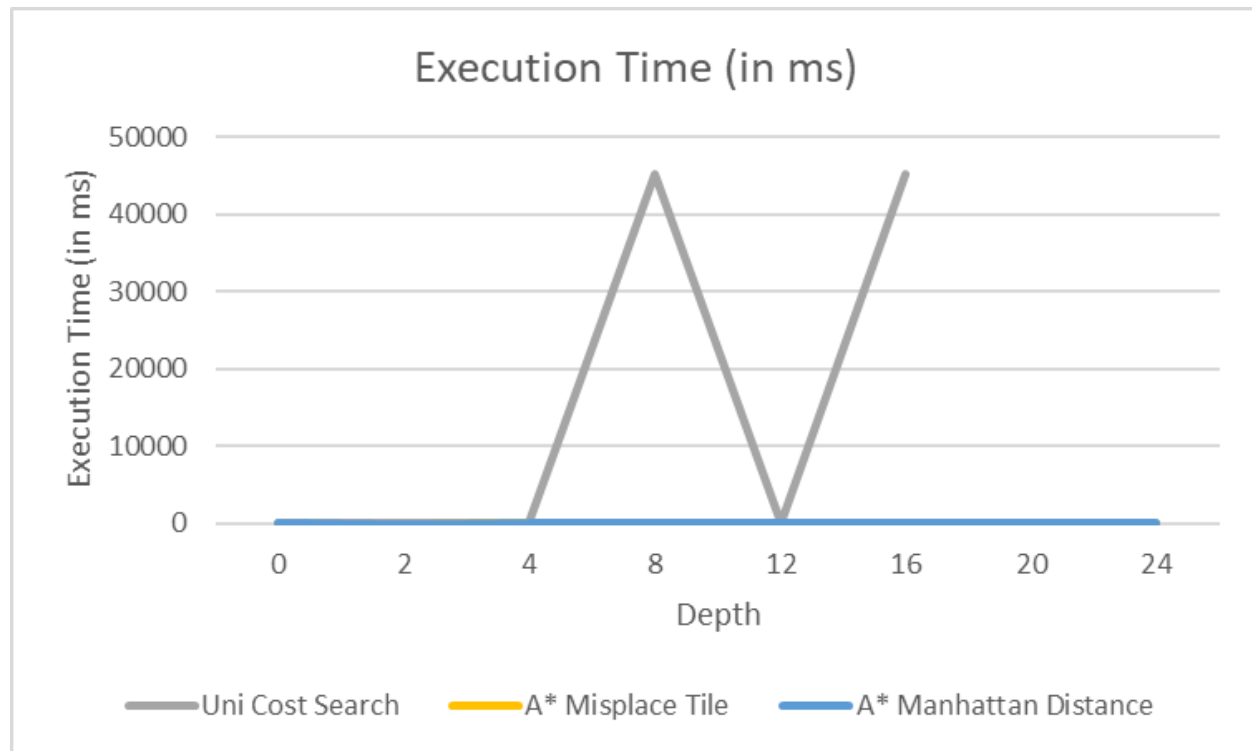


Figure 6. Execution time (in ms) comparison between 3 searching algorithms

Looking at Figures 4, 5, and 6, we can see the big difference in terms of the performance of Uniform Cost Search and A-star Search algorithms. This is due to A-star searches having heuristics that help the algorithm to know how close the state is to the goal node. This gives many hints to search all the way down the tree without caring about other nodes that have higher heuristics. Out of the 2 heuristic calculations, the Manhattan Distance Heuristic is a better implementation since it gives a better and more precise measurement of the distance to the goal state.

However, I have to admit that my implementation of the algorithm is not the best so it gives us results that are not in the range of the theoretical expectation. I could have got the logic wrong or my calculation is off. But, my implementation can still search for the goal state and display each algorithm's strength or weakness.

## Conclusion

The good sides of the 3 searching algorithms are they are both complete and optimal by a degree. However, after taking measurements and data of the test cases for the 3 algorithms, we can see clearly that both A-star searching algorithms run much more optimal compared to the Uniform Cost Search algorithm. One of the main reasons is the A-star algorithms have an idea of the distance or closeness to the goal state by calculating the heuristic numbers while Uniform Cost Search is just expanding every child node. We can also conclude that A-star with Manhattan Distance Heuristic performs quite better compared to the Misplaced Tile Heuristic. This is because the calculation of the Manhattan Distance Heuristic gives the algorithm a better and much more precise sense of whether the current state is close to the goal state or not. After working on this project, we can conclude that the A-star search is probably the best, most optimal searching algorithm out of all the algorithms we learn in class.

## Source code

```
1  /*
2  |Huy Dinh Tran
3  |ID: 862325308
4  |Email: htran197@ucr.edu
5  |
6  |CS205 AI Spring 2022
7  |Project 1
8  |Dr. Eamonn Keogh
9  |*/
10 #include <stdio.h>
11 #include <iostream>
12 #include <algorithm>
13 #include <queue>
14 #include <vector>
15 #include <list>
16 #include <string>
17 #include <stdlib.h>
18 #include <chrono>
19
20 using namespace std;
21
22 struct Node { //struct of a Node which has pointers to childs and parent with attribute of heuristic, movement cost and location of the blank within matrix.
23     int matrix[3][3];
24     Node* child1;
25     Node* child2;
26     Node* child3;
27     Node* child4;
28     Node* parent;
29     int blankX, blankY;
30     int hMan;
31     int hTile;
32     int moveCost;
33 };
34
35 struct compareTile { //used for priority_queue, place lowest MisTile heuristic at the top
36     bool operator()(Node* a, Node* b)
37     {
38         return a->hTile > b->hTile;
39     }
40 };
41
42 struct compareUni { //used for priority_queue, place lowest movement cost at the top
43     bool operator()(Node* a, Node* b)
44     {
45         return a->moveCost > b->moveCost;
46     }
47 };
```

```
48 struct compareMan { //used for priority_queue, place lowest Manhattan heuristic at the top
49     bool operator()(const Node* a, const Node* b) const
50     {
51         return a->hMan > b->hMan;
52     }
53 };
54
55 bool checkGoal(int input[3][3], int goal[3][3]) { //return true if input matrix the same as goal matrix, else, return false
56     for (int i = 0; i < 3; i++)
57     {
58         for (int j = 0; j < 3; j++)
59         {
60             if (input[i][j] != goal[i][j]) {
61                 return false;
62             }
63         }
64     }
65     return true;
66 }
67
68 int hMisTile(int input[3][3], int goal[3][3]) { //return the heuristic value for misplace tiles
69     int h=0;
70     for (int i = 0; i < 3; i++) {
71         for (int j = 0; j < 3; j++) {
72             if (input[i][j] != goal[i][j] && goal[i][j] != 0)
73                 h++;
74         }
75     }
76     return h;
77 }
78 }
```



```

80 int hManhattan(int input[3][3], int goal[3][3]) { //return the heuristic value for Manhattan
81     int h = 0;
82     for (int i = 0; i < 3; i++) {
83         for (int j = 0; j < 3; j++) {
84             if (input[i][j] != goal[i][j]) {
85                 switch (input[i][j]) {
86                     case 1:
87                         h += abs(i - 0) + abs(j - 0);
88                         break;
89                     case 2:
90                         h += abs(i - 0) + abs(j - 1);
91                         break;
92                     case 3:
93                         h += abs(i - 0) + abs(j - 2);
94                         break;
95                     case 4:
96                         h += abs(i - 1) + abs(j - 0);
97                         break;
98                     case 5:
99                         h += abs(i - 1) + abs(j - 1);
100                        break;
101                     case 6:
102                        h += abs(i - 1) + abs(j - 2);
103                        break;
104                     case 7:
105                        h += abs(i - 2) + abs(j - 0);
106                        break;
107                     case 8:
108                        h += abs(i - 2) + abs(j - 1);
109                        break;
110                 }
111             }
112         }
113     }
114     return h;
115 }
116
117 void printBoard(int input[3][3]) { //print the current board
118     for (int i = 0; i < 3; i++) {
119         cout << " ";
120         for (int j = 0; j < 3; j++) {
121             cout << input[i][j] << " ";
122         }
123         cout << "]\n" << endl;
124     }
125 }
126
127 void copyBoard(int (&target)[3][3], int (&matrix)[3][3]) { //copying matrix from input to target
128     for (int i = 0; i < 3; i++) {
129         for (int j = 0; j < 3; j++) {
130             target[i][j] = matrix[i][j];
131         }
132     }
133 }
134
135 bool isDup(vector<Node*> &vec, Node* input) { //searching through vector to see if there is any duplication
136     for (int i = 0; i < vec.size(); i++) {
137         if (vec[i] == input) {
138             return true;
139         }
140     }
141     return false;
142 }
143
144 void assignChild1(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int &nodeExpand) { //making child1
145     Node* temp = new Node;
146     int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
147     temp->moveCost = curr->moveCost + 1;
148     copyBoard(temp->matrix, childMatrix);
149     temp->parent = curr;
150     temp->blankX = blankX;
151     temp->blankY = blankY;
152     temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
153     temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
154     nodeExpand++;
155     curr->child1 = temp;
156 }
157
158 void assignChild2(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int &nodeExpand) { //making child2
159     Node* temp = new Node;
160     int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
161     temp->moveCost = curr->moveCost + 1;
162     copyBoard(temp->matrix, childMatrix);
163     temp->parent = curr;
164     temp->blankX = blankX;
165     temp->blankY = blankY;
166     temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
167     temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
168     nodeExpand++;
169     curr->child2 = temp;
170 }
171
172 void assignChild3(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int &nodeExpand) { //making child3
173     Node* temp = new Node;
174     int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
175     temp->moveCost = curr->moveCost + 1;
176     copyBoard(temp->matrix, childMatrix);
177     temp->parent = curr;
178     temp->blankX = blankX;
179     temp->blankY = blankY;
180     temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
181     temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
182     nodeExpand++;
183     curr->child3 = temp;
184 }
185
186 void assignChild4(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int &nodeExpand) { //making child4
187     Node* temp = new Node;
188     int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
189     temp->moveCost = curr->moveCost + 1;
190     copyBoard(temp->matrix, childMatrix);
191     temp->parent = curr;
192     temp->blankX = blankX;
193     temp->blankY = blankY;
194     temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
195     temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
196     nodeExpand++;
197     curr->child4 = temp;
198 }

```



```

323     case 2:
324         switch (node->blankY) {
325             case 0:
326                 copyBoard(temp, node->matrix);
327                 swap(temp[2][0], temp[1][0]);
328                 if (node->parent->matrix != temp) {
329                     assignChild1(node, temp, 1, 0, nodeExpand);
330                     q.push(node->child1);
331                 }
332                 copyBoard(temp, node->matrix);
333                 swap(temp[2][0], temp[2][1]);
334                 if (node->parent->matrix != temp) {
335                     assignChild2(node, temp, 2, 1, nodeExpand);
336                     q.push(node->child2);
337                 }
338                 break;
339             case 1:
340                 copyBoard(temp, node->matrix);
341                 swap(temp[2][1], temp[2][0]);
342                 if (node->parent->matrix != temp) {
343                     assignChild1(node, temp, 2, 0, nodeExpand);
344                     q.push(node->child1);
345                 }
346                 copyBoard(temp, node->matrix);
347                 swap(temp[2][1], temp[1][1]);
348                 if (node->parent->matrix != temp) {
349                     assignChild2(node, temp, 1, 1, nodeExpand);
350                     q.push(node->child2);
351                 }
352                 copyBoard(temp, node->matrix);
353                 swap(temp[2][1], temp[2][2]);
354                 if (node->parent->matrix != temp) {
355                     assignChild3(node, temp, 2, 2, nodeExpand);
356                     q.push(node->child3);
357                 }
358                 break;
359             case 2:
360                 copyBoard(temp, node->matrix);
361                 swap(temp[2][2], temp[2][1]);
362                 if (node->parent->matrix != temp) {
363                     assignChild1(node, temp, 2, 1, nodeExpand);
364                     q.push(node->child1);
365                 }
366                 copyBoard(temp, node->matrix);
367                 swap(temp[2][2], temp[1][2]);
368                 if (node->parent->matrix != temp) {
369                     assignChild2(node, temp, 1, 2, nodeExpand);
370                     q.push(node->child2);
371                 }
372                 break;
373             }
374         }
375     }
376 }
377
378 void moveBlankMan(Node* n, priority_queue<Node*, vector<Node*>, compareMan> &q, int& nodeExpand, int& queueSize) { //moving empty tile, creating childs and also pushing into queue for A*Manhattan
379     int temp[3][3];
380     switch (node->blankX) {
381         case 0:
382             switch (node->blankY) {
383                 case 0:
384                     copyBoard(temp, node->matrix);
385                     swap(temp[0][0], temp[0][1]);
386                     if (node->parent->matrix != temp) {
387                         assignChild1(node, temp, 0, 1, nodeExpand);
388                         q.push(node->child1);
389                     }
390                     copyBoard(temp, node->matrix);
391                     swap(temp[0][0], temp[1][0]);
392                     if (node->parent->matrix != temp) {
393                         assignChild2(node, temp, 1, 0, nodeExpand);
394                         q.push(node->child2);
395                     }
396                     break;
397                 case 1:
398                     copyBoard(temp, node->matrix);
399                     swap(temp[0][1], temp[0][0]);
400                     if (node->parent->matrix != temp) {
401                         assignChild1(node, temp, 0, 0, nodeExpand);
402                         q.push(node->child1);
403                     }
404                     copyBoard(temp, node->matrix);
405                     swap(temp[0][1], temp[0][2]);
406                     if (node->parent->matrix != temp) {
407                         assignChild2(node, temp, 0, 2, nodeExpand);
408                         q.push(node->child2);
409                     }
410                     copyBoard(temp, node->matrix);
411                     swap(temp[0][1], temp[1][1]);
412                     if (node->parent->matrix != temp) {
413                         assignChild3(node, temp, 1, 1, nodeExpand);
414                         q.push(node->child3);
415                     }
416                     break;

```

```

416 case 2:
417     copyBoard(temp, node->matrix);
418     swap(temp[0][2], temp[0][1]);
419     if (node->parent->matrix != temp) {
420         assignChild1(node, temp, 0, 1, nodeExpand);
421         q.push(node->child1);
422     }
423     copyBoard(temp, node->matrix);
424     swap(temp[0][2], temp[1][2]);
425     if (node->parent->matrix != temp) {
426         assignChild2(node, temp, 1, 2, nodeExpand);
427         q.push(node->child2);
428     }
429     break;
430 }
431 case 1:
432     switch (node->blankY) {
433     case 0:
434         copyBoard(temp, node->matrix);
435         swap(temp[1][0], temp[0][0]);
436         if (node->parent->matrix != temp) {
437             assignChild1(node, temp, 0, 0, nodeExpand);
438             q.push(node->child1);
439         }
440         copyBoard(temp, node->matrix);
441         swap(temp[1][0], temp[1][1]);
442         if (node->parent->matrix != temp) {
443             assignChild2(node, temp, 1, 1, nodeExpand);
444             q.push(node->child2);
445         }
446         copyBoard(temp, node->matrix);
447         swap(temp[1][0], temp[2][0]);
448         if (node->parent->matrix != temp) {
449             assignChild3(node, temp, 2, 0, nodeExpand);
450             q.push(node->child3);
451         }
452         break;

```

```

453 case 1:
454     copyBoard(temp, node->matrix);
455     swap(temp[1][1], temp[1][0]);
456     if (node->parent->matrix != temp) {
457         assignChild1(node, temp, 1, 0, nodeExpand);
458         q.push(node->child1);
459     }
460     copyBoard(temp, node->matrix);
461     swap(temp[1][1], temp[0][1]);
462     if (node->parent->matrix != temp) {
463         assignChild2(node, temp, 0, 1, nodeExpand);
464         q.push(node->child2);
465     }
466     copyBoard(temp, node->matrix);
467     swap(temp[1][1], temp[1][2]);
468     if (node->parent->matrix != temp) {
469         assignChild3(node, temp, 1, 2, nodeExpand);
470         q.push(node->child3);
471     }
472     copyBoard(temp, node->matrix);
473     swap(temp[1][1], temp[2][1]);
474     if (node->parent->matrix != temp) {
475         assignChild4(node, temp, 2, 1, nodeExpand);
476         q.push(node->child4);
477     }
478     break;
479 case 2:
480     copyBoard(temp, node->matrix);
481     swap(temp[1][2], temp[1][1]);
482     if (node->parent->matrix != temp) {
483         assignChild1(node, temp, 1, 1, nodeExpand);
484         q.push(node->child1);
485     }
486     copyBoard(temp, node->matrix);
487     swap(temp[1][2], temp[0][2]);
488     if (node->parent->matrix != temp) {
489         assignChild2(node, temp, 0, 2, nodeExpand);
490         q.push(node->child2);
491     }
492     copyBoard(temp, node->matrix);
493     swap(temp[1][2], temp[2][2]);
494     if (node->parent->matrix != temp) {
495         assignChild3(node, temp, 2, 2, nodeExpand);
496         q.push(node->child3);
497     }
498     break;
499 }
500 case 2:

```

```

501 case 2:
502     switch (node->blankY) {
503     case 0:
504         copyBoard(temp, node->matrix);
505         swap(temp[2][0], temp[1][0]);
506         if (node->parent->matrix != temp) {
507             assignChild1(node, temp, 1, 0, nodeExpand);
508             q.push(node->child1);
509         }
510         copyBoard(temp, node->matrix);
511         swap(temp[2][0], temp[2][1]);
512         if (node->parent->matrix != temp) {
513             assignChild2(node, temp, 2, 1, nodeExpand);
514             q.push(node->child2);
515         }
516         break;
517     case 1:
518         copyBoard(temp, node->matrix);
519         swap(temp[2][1], temp[0][0]);
520         if (node->parent->matrix != temp) {
521             assignChild1(node, temp, 2, 0, nodeExpand);
522             q.push(node->child1);
523         }
524         copyBoard(temp, node->matrix);
525         swap(temp[2][1], temp[1][1]);
526         if (node->parent->matrix != temp) {
527             assignChild2(node, temp, 1, 1, nodeExpand);
528             q.push(node->child2);
529         }
530         copyBoard(temp, node->matrix);
531         swap(temp[2][1], temp[2][2]);
532         if (node->parent->matrix != temp) {
533             assignChild3(node, temp, 2, 2, nodeExpand);
534             q.push(node->child3);
535         }
536         break;

```

```

536     case 2:
537         copyBoard(temp, node->matrix);
538         swap(temp[2][2], temp[2][1]);
539         if (node->parent->matrix != temp) {
540             assignChild1(node, temp, 2, 1, nodeExpand);
541             q.push(node->child1);
542         }
543         copyBoard(temp, node->matrix);
544         swap(temp[2][2], temp[1][2]);
545         if (node->parent->matrix != temp) {
546             assignChild2(node, temp, 1, 2, nodeExpand);
547             q.push(node->child2);
548         }
549         break;
550     }
551 }
552 }

554 void moveBlankUni(Node& node, priority_queue<Node*, vector<Node*>, compareUni> &q, int& nodeExpand, int& queueSize) { //moving empty tile, creating childs and also pushing into queue for UniCostSearch
555     int temp[3][3];
556     switch (node->blankX) {
557     case 0:
558         switch (node->blankY) {
559         case 0:
560             copyBoard(temp, node->matrix);
561             swap(temp[0][0], temp[0][1]);
562             if (node->parent->matrix != temp) {
563                 assignChild1(node, temp, 0, 1, nodeExpand);
564                 q.push(node->child1);
565             }
566             copyBoard(temp, node->matrix);
567             swap(temp[0][0], temp[1][0]);
568             if (node->parent->matrix != temp) {
569                 assignChild2(node, temp, 1, 0, nodeExpand);
570                 q.push(node->child2);
571             }
572             break;
573         case 1:
574             copyBoard(temp, node->matrix);
575             swap(temp[0][1], temp[0][0]);
576             if (node->parent->matrix != temp) {
577                 assignChild1(node, temp, 0, 0, nodeExpand);
578                 q.push(node->child1);
579             }
580             copyBoard(temp, node->matrix);
581             swap(temp[0][1], temp[0][2]);
582             if (node->parent->matrix != temp) {
583                 assignChild2(node, temp, 0, 2, nodeExpand);
584                 q.push(node->child2);
585             }
586             copyBoard(temp, node->matrix);
587             swap(temp[0][1], temp[1][1]);
588             if (node->parent->matrix != temp) {
589                 assignChild3(node, temp, 1, 1, nodeExpand);
590                 q.push(node->child3);
591             }
592             break;
593         case 2:
594             copyBoard(temp, node->matrix);
595             swap(temp[0][2], temp[0][1]);
596             if (node->parent->matrix != temp) {
597                 assignChild1(node, temp, 0, 1, nodeExpand);
598                 q.push(node->child1);
599             }
600             copyBoard(temp, node->matrix);
601             swap(temp[0][2], temp[1][2]);
602             if (node->parent->matrix != temp) {
603                 assignChild2(node, temp, 1, 2, nodeExpand);
604                 q.push(node->child2);
605             }
606             break;
607         }
608     case 1:
609         switch (node->blankY) {
610         case 0:
611             copyBoard(temp, node->matrix);
612             swap(temp[1][0], temp[0][0]);
613             if (node->parent->matrix != temp) {
614                 assignChild1(node, temp, 0, 0, nodeExpand);
615                 q.push(node->child1);
616             }
617             copyBoard(temp, node->matrix);
618             swap(temp[1][0], temp[1][1]);
619             if (node->parent->matrix != temp) {
620                 assignChild2(node, temp, 1, 1, nodeExpand);
621                 q.push(node->child2);
622             }
623             copyBoard(temp, node->matrix);
624             swap(temp[1][0], temp[2][0]);
625             if (node->parent->matrix != temp) {
626                 assignChild3(node, temp, 2, 0, nodeExpand);
627                 q.push(node->child3);
628             }
629             break;
630         case 1:
631             copyBoard(temp, node->matrix);
632             swap(temp[1][1], temp[1][0]);
633             if (node->parent->matrix != temp) {
634                 assignChild1(node, temp, 1, 0, nodeExpand);
635                 q.push(node->child1);
636             }
637             copyBoard(temp, node->matrix);
638             swap(temp[1][1], temp[0][1]);
639             if (node->parent->matrix != temp) {
640                 assignChild2(node, temp, 0, 1, nodeExpand);
641                 q.push(node->child2);
642             }
643             copyBoard(temp, node->matrix);
644             swap(temp[1][1], temp[1][2]);
645             if (node->parent->matrix != temp) {
646                 assignChild3(node, temp, 1, 2, nodeExpand);
647                 q.push(node->child3);
648             }
649             copyBoard(temp, node->matrix);
650             swap(temp[1][1], temp[2][1]);

```

```

651     if (node->parent->matrix != temp) {
652         assignChild4(node, temp, 2, 1, nodeExpand);
653         q.push(node->child4);
654     }
655     break;
656     case 2:
657         copyBoard(temp, node->matrix);
658         swap(temp[1][2], temp[1][1]);
659         if (node->parent->matrix != temp) {
660             assignChild1(node, temp, 1, 1, nodeExpand);
661             q.push(node->child1);
662         }
663         copyBoard(temp, node->matrix);
664         swap(temp[1][2], temp[0][2]);
665         if (node->parent->matrix != temp) {
666             assignChild2(node, temp, 0, 2, nodeExpand);
667             q.push(node->child2);
668         }
669         copyBoard(temp, node->matrix);
670         swap(temp[1][2], temp[2][2]);
671         if (node->parent->matrix != temp) {
672             assignChild3(node, temp, 2, 2, nodeExpand);
673             q.push(node->child3);
674         }
675     }
676     break;
677     case 2:
678         switch (node->blankY) {
679             case 0:
680                 copyBoard(temp, node->matrix);
681                 swap(temp[2][0], temp[1][0]);
682                 if (node->parent->matrix != temp) {
683                     assignChild1(node, temp, 1, 0, nodeExpand);
684                     q.push(node->child1);
685                 }
686                 copyBoard(temp, node->matrix);
687                 swap(temp[2][0], temp[2][1]);
688                 if (node->parent->matrix != temp) {
689                     assignChild2(node, temp, 2, 1, nodeExpand);
690                     q.push(node->child2);
691                 }
692             }
693             break;
694             case 1:
695                 copyBoard(temp, node->matrix);
696                 swap(temp[2][1], temp[2][0]);
697                 if (node->parent->matrix != temp) {
698                     assignChild1(node, temp, 2, 0, nodeExpand);
699                     q.push(node->child1);

```

```

700         }
701         copyBoard(temp, node->matrix);
702         swap(temp[2][1], temp[1][1]);
703         if (node->parent->matrix != temp) {
704             assignChild2(node, temp, 1, 1, nodeExpand);
705             q.push(node->child2);
706         }
707         copyBoard(temp, node->matrix);
708         swap(temp[2][1], temp[2][2]);
709         if (node->parent->matrix != temp) {
710             assignChild3(node, temp, 2, 2, nodeExpand);
711             q.push(node->child3);
712         }
713     }
714     break;
715     case 2:
716         copyBoard(temp, node->matrix);
717         swap(temp[2][2], temp[2][1]);
718         if (node->parent->matrix != temp) {
719             assignChild1(node, temp, 2, 1, nodeExpand);
720             q.push(node->child1);
721         }
722         copyBoard(temp, node->matrix);
723         swap(temp[2][2], temp[1][2]);
724         if (node->parent->matrix != temp) {
725             assignChild2(node, temp, 1, 2, nodeExpand);
726             q.push(node->child2);
727         }
728     }
729     break;
730 }
731 }
732 void UniCostSearch(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for Uniform Cost Search
733     priority_queue<Node*, vector<Node*>, compareUni> q;
734     priority_queue<Node*, vector<Node*>, compareUni> q2;
735     Node* temp = new Node;
736     vector<Node*> duplicate;
737     q.push(root);
738     while (q.size() > 0) {
739         if (q.empty() == true) {
740             cout << "Failure";
741             return;
742         }
743         q2 = q;
744         while (q.size() > 0) {
745             if (queueSize < q.size()) {
746                 queueSize = q.size();
747             }
748             temp = q.top();

```

```

748     q.pop();
749     cout << "g(n) = " << temp->moveCost << " and h(n) = 0" << endl;
750     printBoard(temp->matrix);
751
752     if (checkGoal(temp->matrix, goal) == true) {
753         cout << "Solution found!" << endl;
754         cout << "Solution depth: " << temp->moveCost << endl;
755         cout << "Number of nodes expanded: " << nodeExpand << endl;
756         cout << "Max queue size: " << queueSize << endl;
757         return;
758     }
759
760     while (q2.size() > 0) {
761         temp = q2.top();
762         q2.pop();
763         if (isDup(duplicate, temp) == false) {
764             moveBlankini(temp, q, nodeExpand, queueSize);
765             duplicate.insert(duplicate.begin(), temp);
766         }
767     }
768 }
769
770
771 void AMisplacedTile(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for A star Misplaced Tile Heuristic
772     priority_queue<Node*, vector<Node*>, compareTile> q;
773     Node* temp = new Node;
774     vector<Node*> duplicate;
775     q.push(root);
776     while (q.size() > 0) {
777         if (q.empty() == true) {
778             cout << "Failure";
779             return;
780         }
781         if (queueSize < q.size()) {
782             queueSize = q.size();
783         }
784         temp = q.top();
785         q.pop();
786         cout << "g(n) = " << temp->moveCost << " and h(n) = " << temp->hTile - temp->moveCost << endl;
787         printBoard(temp->matrix);
788
789         if (checkGoal(temp->matrix, goal) == true) {
790             cout << "Solution found!" << endl;
791             cout << "Solution depth: " << temp->moveCost << endl;
792             cout << "Number of nodes expanded: " << nodeExpand << endl;
793             cout << "Max queue size: " << queueSize << endl;
794             return;
795         }
796         else {
797             moveBlankiTile(temp, q, nodeExpand, queueSize);
798             duplicate.insert(duplicate.begin(), temp);
799         }
800     }
801 }
802
803
804 void AManhattan(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for A star Manhattan Distance Heuristic
805     priority_queue<Node*, vector<Node*>, compareMan> q;
806     Node* temp = new Node;
807     vector<Node*> duplicate;
808     q.push(root);
809     while (q.size() > 0) {
810         if (q.empty() == true) {
811             cout << "Failure";
812             return;
813         }
814         if (queueSize < q.size()) {
815             queueSize = q.size();
816         }
817         temp = q.top();
818         q.pop();
819         cout << "g(n) = " << temp->moveCost << " and h(n) = " << temp->hMan - temp->moveCost << endl;
820         printBoard(temp->matrix);
821
822         if (checkGoal(temp->matrix, goal) == true) {
823             cout << "Solution found!" << endl;
824             cout << "Solution depth: " << temp->moveCost << endl;
825             cout << "Number of nodes expanded: " << nodeExpand << endl;
826             cout << "Max queue size: " << queueSize << endl;
827             return;
828         }
829         else {
830             moveBlankiMan(temp, q, nodeExpand, queueSize);
831             duplicate.insert(duplicate.begin(), temp);
832         }
833     }
834 }
835
836
837 int main() //Choosing and user inputing current state into searching algorithm
838 {
839     int initChoose;
840     int blankX=1, blankY=1;
841     int depth, nodeExpand, queueSize;
842     int matrix[3][3] = { { 1,2,3 }, { 5,0,6 }, { 4,7,8 } };
843     int goal[3][3] = { { 1,2,3 }, { 4,5,6 }, { 7,8,0 } };
844 }

```

```

807 cout << "Welcome to Huy Dinh Tran (862325388, htran197) 8-Puzzle Solver.\nType '1' to use a default puzzle, or '2' to create your own.\n";
808 cout << "Default puzzle: [ 1 2 3 ]" << endl;
809 cout << "      [ 5 0 6 ]\n";
810 cout << "      [ 4 7 8 ]\n";
811 cin >> initChoose;
812
813 if (initChoose == 2) {
814     cout << "Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. \nEnter the puzzle demilimiting the numbers with a space. Press ENTER only when finished.\n";
815     cout << "Enter the first row: ";
816     for (int i = 0; i < 3; i++) {
817         cin >> matrix[0][i];
818         if (matrix[0][i] == 0) {
819             blankX = 0;
820             blankY = i;
821         }
822     }
823     cout << "Enter the second row: ";
824     for (int i = 0; i < 3; i++) {
825         cin >> matrix[1][i];
826         if (matrix[1][i] == 0) {
827             blankX = 1;
828             blankY = i;
829         }
830     }
831     cout << "Enter the third row: ";
832     for (int i = 0; i < 3; i++) {
833         cin >> matrix[2][i];
834         if (matrix[2][i] == 0) {
835             blankX = 2;
836             blankY = i;
837         }
838     }
839 }
840
841 Node* node = new Node;
842 for (int i = 0; i < 3; i++){
843     for (int j = 0; j < 3; j++){
844         node->matrix[i][j] = matrix[i][j];
845     }
846 }
847
848 //initializing root node
849 node->parent = NULL;
850 node->hMan = 0;
851 node->hTile = 0;
852 node->moveCost = 0;
853 node->blankX = blankX;
854 node->blankY = blankY;
855 depth = 0;
856
857
858 nodeExpand = 0;
859 queueSize = 0;
860
861 int algoChoose;
862 cout << "Select algorithm: \n(1) for Uniform Cost Search \n(2) for the Misplaced Tile Heuristic \n(3) the Manhattan Distance Heuristic" << endl;
863 cin >> algoChoose;
864
865 auto start = chrono::steady_clock::now();
866
867 switch (algoChoose) {
868     case 1:
869         UniCostSearch(matrix, goal, node, depth, nodeExpand, queueSize);
870         break;
871     case 2:
872         AMisTile(matrix, goal, node, depth, nodeExpand, queueSize);
873         break;
874     case 3:
875         AManhattan(matrix, goal, node, depth, nodeExpand, queueSize);
876         break;
877 }
878
879 auto end = chrono::steady_clock::now();
880 auto diff = end - start;
881
882 cout << "Execution time (seconds): " << chrono::duration <double, milli>(diff).count() << " ms" << endl;
883
884 return 0;
885 }

```