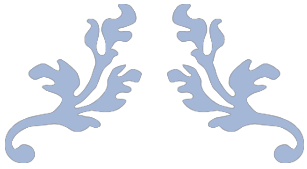


COURSEWORK

Feng Xie 30502322

Table of Contents

Part 1 Basic Search Methods

1 Description

In this part, we are asked to solve the problem of "Blocks-world tile puzzle" with different types of search algorithms including depth-first search, breadth-first search, iterative deepening search and A* heuristic search. Then we need to examine how the computational time increases with the difficulty of the problem and plot a figure of the time complexity.

The programming language used for this part is C++. The cost of every move made by the agent is considered as one.

2 Methods

2.1 Presentation of the problem

The state was interpreted as a 2-dimention array. The blocks of A, B and C were represented by the numbers of 1, 2, and 3. And the agent and every white tile were represented by the numbers of -1 and 0. For example, the start state was interpreted as an array shown below.

```
int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {1, 2, 3, -1}
};
```

2.2 Breadth-first Search

Breadth-first Search expands the shallowest node that hasn't been expanded. It was implemented with a FIFO queue. The goal test was applied when a node was generated rather than when it was selected for expansion. This decision was made so that the number of nodes generated was in accordance with the way of how we compute the time complexity.
The search will proceed endlessly until the memory is full or it finds the goal state or the queue is empty. But in the tree search implementation, we do not store the nodes visited and the agent always has valid moves in every location so the queue will never be empty.

2.3 Depth-first Search

Depth-first Search expands the deepest node that hasn't been expanded. The search will not "back up" to the next deepest node until it reaches a leaf node that cannot be expanded. It was implemented in two versions. In the recursive version, the search function will call itself after it expands a node. A segmentation fault was encountered when the code was run because the depth first search was stuck in a loop, making the stack of the computer become full. In the iterative version, a LIFO stack was used to store the unexpanded nodes. Although no error was thrown, the program cannot stop because it cannot find a solution.

In this problem, since we do not save the nodes that have been visited and the agent can always move to some directions in every location., the search will not reach a leaf node. By doing experiments, we can find that depth-first search cannot solve the puzzle with the given start state.

2.4 Iterative Deepening Search

Iterative deepening search starts with depth-first search with the depth limit of zero. If it cannot find a solution, it will increase the depth limit by 1 and restart the search. The search will proceed until a solution is found or the memory is full. A recursive function was used to implement the depth-limited search and an endless loop was used to implement the iterative deepening part.

2.5 A* heuristic search

A* search used an evaluation function f(n) = g(n) + h(n) where g(n) represents the cost so far and h(n) refers to the estimated cost from node n to the goal. The search will always select the node with the smallest f value from the unexpanded nodes to expand. The sum of Manhattan distances was chosen as the heuristic function.

In the code, a single linked list was implemented to store the unexpanded nodes. Every time the code will scan the whole list and remove the node with the smallest f value. If the node cannot pass the goal test, it will be used to expand nodes. The goal test was not applied when a node was generated because in that way, the solution found may not be the one with the smallest f value.

3 Evidence

3.1 The solutions of the algorithms

In terms of the original set up of the puzzle, breadth-first search, iterative deepening search and A* search are able to give the optimal solution as shown below. The action sequence points out what action was taken from the start state to the goal state. The path costs are the same for the three algorithms. The depth-first search is not able to solve the puzzle because it will be stuck in an endless loop.

```
Breadth-first Search:
Action sequence:  up left left down left up right down right up up left down left
Path cost: 14
Nodes expanded: 6179535

Iterative Deepening Search:
Action sequence:  up left left down left up right down right up up left down left
Path cost: 14
Nodes generated: 8943130

A* Heuristic Search:
Action sequence:  up left left down left up right down right up up left down left
Path cost: 14
Nodes generated: 84242
```

3.2 The debugging outputs of the algorithms

3.2.1 Breadth-first Search

To demonstrate how breadth-first search works using the debugging outputs, an initial state was changed to the one shown as below:

```
int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 1, 0, 0},
        {2, 0, -1, 0},
        {0, 3, 0, 0}
    };
```

The states of the nodes expanded by breadth-first search are shown below. In each state, the children nodes are expanded in the direction of “up, down, left, right”.

0 0 0 0
0 1 -1 0
2 0 0 0
0 3 0 0

0 0 0 0
0 1 0 0
2 0 0 0
0 3 -1 0

0 0 0 0
0 1 0 0
2 -1 0 0
0 3 0 0

0 0 0 0
0 1 0 0
2 0 0 -1
0 3 0 0

0 0 -1 0
0 1 0 0
2 0 0 0
0 3 0 0

0 0 0 0
0 1 0 0
2 0 -1 0
0 3 0 0

0 0 0 0
0 -1 1 0
2 0 0 0
0 3 0 0

0 0 0 0
0 1 0 -1
2 0 0 0
0 3 0 0

0 0 0 0
0 1 0 0
2 0 -1 0
0 3 0 0

0 0 0 0
0 1 0 0
2 0 0 0
0 -1 3 0

0 0 0 0
0 1 0 0
2 0 0 0
0 3 0 -1

0 0 0 0
0 -1 0 0
2 1 0 0
0 3 0 0

0 0 0 0
0 1 0 0

2 3 0 0
0 -1 0 0

0 0 0 0
0 1 0 0
-1 2 0 0
0 3 0 0

Breadth-first Search:
Path cost: 2
Nodes expanded: 15

3.2.2 Depth-first Search

To demonstrate how depth-first search works using the debugging outputs, an initial state was changed to the one shown as below:

```
int puzzle[4][4] = {
        {0, 1, 0, 0},
        {0, 2, 0, 0},
        {0, 3, 0, 0},
        {0, -1, 0, 0}
    };
```

The states of the nodes expanded by depth-first search are shown below. In each state, the children nodes are expanded in the direction of "up, down, left, right". Because the depth-first search was implemented in a recursive way, the function called itself after a node was generated. The tree will not back up and search other branches until the agent is not able to move in the current direction.

0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0

0 1 0 0
0 -1 0 0
0 2 0 0
0 3 0 0

0 -1 0 0
0 1 0 0
0 2 0 0
0 3 0 0

Path cost: 3
Nodes expanded: 4

3.2.2 Iterative Deepening Search

To demonstrate how iterative deepening search works using the debugging outputs, an initial state was changed to the one shown as below:

```
int puzzle[4][4] = {
        {0, 1, 0, 0},
        {0, 2, 0, 0},
        {0, 3, 0, 0},
        {0, -1, 0, 0}
```

```
    };
```

The states of the nodes expanded by iterative deepening search are shown below. In each state, the children nodes are expanded in the direction of “up, down, left, right”.

Depth limit: 0

Depth limit: 1

0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0

0 1 0 0
0 2 0 0
0 3 0 0
-1 0 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 0 -1 0

Depth limit: 2

0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0

0 1 0 0
0 -1 0 0
0 2 0 0
0 3 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 -1 0 0

0 1 0 0
0 2 0 0
-1 0 0 0
0 3 0 0

0 1 0 0
0 2 0 0
0 0 -1 0
0 3 0 0

0 1 0 0
0 2 0 0
0 3 0 0
-1 0 0 0

0 1 0 0

0 2 0 0
-1 3 0 0
0 0 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 -1 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 0 -1 0

0 1 0 0
0 2 0 0
0 3 -1 0
0 0 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 -1 0 0

0 1 0 0
0 2 0 0
0 3 0 0
0 0 0 -1

Depth limit: 3
0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0

0 1 0 0
0 -1 0 0
0 2 0 0
0 3 0 0

0 -1 0 0
0 1 0 0
0 2 0 0
0 3 0 0

Iterative Deepening Search:
Path cost: 3
Nodes generated: 19

3.2.2 A* Heuristic Search
To demonstrate how A* search works using the debugging outputs, an initial state was changed to the one shown as below:

```
    int puzzle[4][4] = {
        {0, 1, 0, 0},
        {0, 2, 0, 0},
        {0, 3, 0, 0},
        {0, -1, 0, 0}
    };
```

The states of the nodes expanded by A* search are shown below. In each state, the children nodes are expanded in the direction of “up, down, left, right”.

0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0
The sum of Manhattan distances: 2
Current path cost: 1

0 1 0 0
0 2 0 0
0 3 0 0
-1 0 0 0
The sum of Manhattan distances: 3
Current path cost: 1

0 1 0 0
0 2 0 0
0 3 0 0
0 0 -1 0
The sum of Manhattan distances: 3
Current path cost: 1

0 1 0 0
0 -1 0 0
0 2 0 0
0 3 0 0
The sum of Manhattan distances: 1
Current path cost: 2

0 1 0 0
0 2 0 0
0 3 0 0
0 -1 0 0
The sum of Manhattan distances: 3
Current path cost: 2

0 1 0 0
0 2 0 0
-1 0 0 0
0 3 0 0
The sum of Manhattan distances: 2
Current path cost: 2

0 1 0 0
0 2 0 0

0 0 -1 0
0 3 0 0
The sum of Manhattan distances: 2
Current path cost: 2

0 -1 0 0
0 1 0 0
0 2 0 0
0 3 0 0
The sum of Manhattan distances: 0
Current path cost: 3

0 1 0 0
0 2 0 0
0 -1 0 0
0 3 0 0
The sum of Manhattan distances: 2
Current path cost: 3

0 1 0 0
-1 0 0 0
0 2 0 0
0 3 0 0
The sum of Manhattan distances: 1
Current path cost: 3

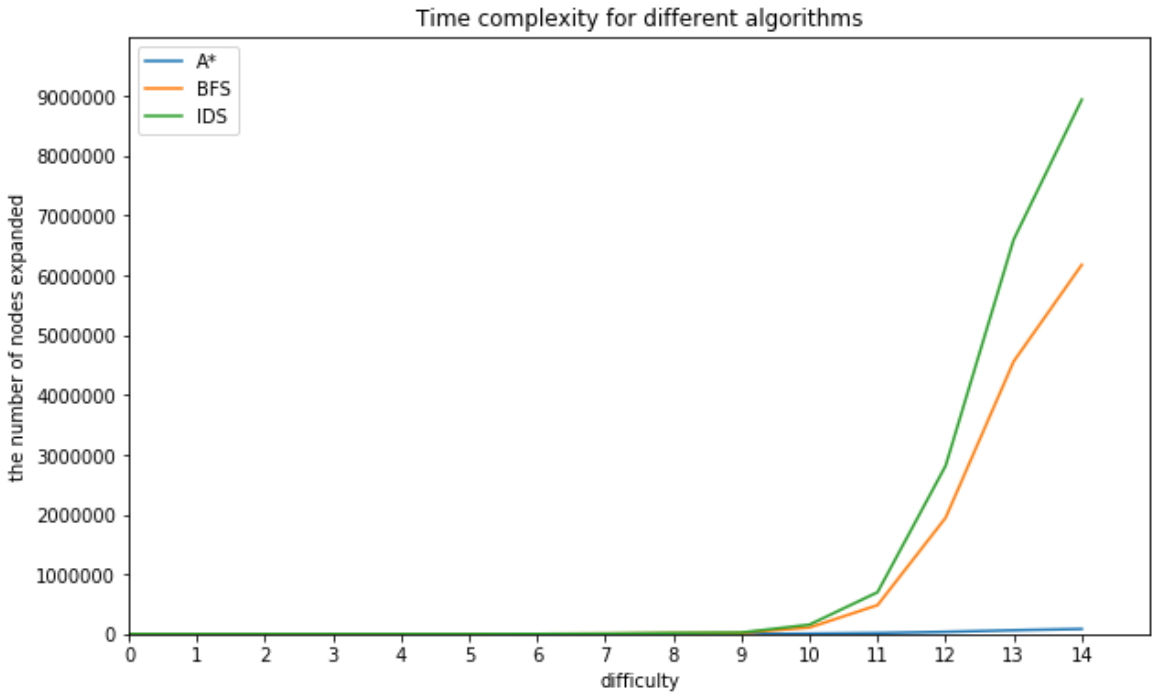
0 1 0 0
0 0 -1 0
0 2 0 0
0 3 0 0
The sum of Manhattan distances: 1
Current path cost: 3

A* Heuristic Search:
Path cost: 3
Nodes generated: 12

4 Scalability

The problem difficulty was controlled by controlling the depth of the optimal solution. First, an optimal solution was found by running one of the algorithms that can solve the puzzle with the given start state and then a sequence of actions was recorded. In this problem, the least path cost is 14 so that the maximum number of difficulties was set to 14. When the difficulty is zero, the agent is in the goal state while a difficulty of 14 means that the agent is in the start state.

The plot demonstrates how the number of expanded nodes to reach a solution scales up with the problem difficulty. It is easy to find that A* search is able to reach a solution with minimal number of nodes expanded given the same difficulty. In terms of BFS(breadth-first search) and IDS(iterative deepening search), although IDS expanded a larger number of nodes than BFS given the same difficulty, its space complexity is linear while the space complexity of BFS is exponential.



5 Limitations

The depth-first search is not able to solve this problem because it will be stuck in one direction and doing endless loops. Th depth-first search will possibly work when the start state is made closer to the goal state or a random order is applied when expanding nodes.

In A* search, a single linked list was implemented to store the unexpanded nodes but the method used to find the node with the minimal f value had a bad performance. A more effective sorting algorithm or a priority queue can be applied to achieve better performance.

6 Conclusion

By implementing the four different types of basic search algorithms and applying them in the problem of "Blocks-world tile puzzle", we can understand how their performances differ from each other and how the computational time scales up with the problem size.

7 Code

```
// Node.h
// The head file for blind search(BFS, DFS, IDS)

#ifndef NODE_H
#define NODE_H

class Node {
    public:
        // Node(){};
        Node(int puzzle[4][4], int depth);
        int* GetAgentLocation();
        bool IsGoal();
        int state_[4][4];
        Node* parent;
```

```
        int path_cost_;
        int depth_;
        int action_;
};


void Swap(int& a, int& b);
void PrintPuzzle(int puzzle[4][4]);
Node* MoveAgent(int direction, Node *current);
bool IsValidMove(int direction, Node *current);
void PrintResult(Node* node);
void PrintAction(const Node* node);

#endif


// Node.cpp
// An implementation file for "Node.h"
// Blind search

#include "Node.h"
#include <iostream>
using namespace std;

// initialize a node with the state of the puzzle
// and its depth in the tree structure
Node::Node(int puzzle[4][4], int depth) {
    parent = NULL;
    depth_ = depth;
    action_ = 0;
    path_cost_ = 0;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            state_[i][j] = puzzle[i][j];
        }
    }
}

// return true if the node contains the goal state
bool Node::IsGoal() {
    // goal state:
    //  {0, 0, 0, 0},
    //  {0, 1, 0, 0},
    //  {0, 2, 0, 0},
    //  {0, 3, 0, 0}
    return (state_[1][1] == 1 && state_[2][1] == 2 && state_[3][1] == 3);
}

int* Node::GetAgentLocation() {
```

```
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (state_[i][j] == -1) {
                int *res = new int[2];
                res[0] = i;
                res[1] = j;
                return res;
            }
        }
    }
    return NULL;
}

// Move the agent according to the current position and the direction.
// Return a new node after the move.
Node* MoveAgent(int direction, Node* current) {
    int* agent_loc = current -> GetAgentLocation();
    // Initial the representation of the new state
    // using the current state.
    int new_state_[4][4];
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++)
        {
            new_state_[i][j] = current -> state_[i][j];
        }
    }
    // Move the position of the agent.
    switch(direction) {
        case 1:
            Swap(new_state_[agent_loc[0]][agent_loc[1]], new_state_[agent_loc[0] -
1][agent_loc[1]]);
            break;
        case 2:
            Swap(new_state_[agent_loc[0]][agent_loc[1]], new_state_[agent_loc[0] +
1][agent_loc[1]]);
            break;
        case 3:
            Swap(new_state_[agent_loc[0]][agent_loc[1]],
new_state_[agent_loc[0]][agent_loc[1] - 1]);
            break;
        case 4:
            Swap(new_state_[agent_loc[0]][agent_loc[1]],
new_state_[agent_loc[0]][agent_loc[1] + 1]);
            break;
    }
    Node* new_node = new Node(new_state_, current -> depth_ + 1);
    new_node -> parent = current;  // Parent logged.
    new_node -> action_ = direction; // Action logged.
    new_node -> path_cost_ = current -> path_cost_ + 1;
    return new_node;
}
```

```
// A move is valid as long as the agent will not cross the boundaries.
bool IsValidMove(int direction, Node *current) {
    /*
    direction:
    1: up
    2: down
    3: left
    4: right
    */
    int* agent = current -> GetAgentLocation();
    int result;
    // In the normal way, we don't fix anything,
    // even the agent will go back and forth between and current node
    // and the node it just visited.
    switch(direction)
    {
        case 1:
            result = agent[0] - 1;
            break;
        case 2:
            result = agent[0] + 1;
            break;
        case 3:
            result = agent[1] - 1;
            break;
        case 4:
            result = agent[1] + 1;
            break;
    }
    return (result >= 0 && result < 4);
}

void PrintPuzzle(int puzzle[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", puzzle[i][j]);
        }
        printf("\n");
    }
}

void Swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void PrintAction(const Node* node) {
    int act = node -> action_;
    cout << "Action: ";
```

```
    switch(act) {
        case 0: break;
        case 1:
            cout << "up";
            break;
        case 2:
            cout << "down";
            break;
        case 3:
            cout << "left";
            break;
        case 4:
            cout << "right";
            break;
    }
    cout << endl << endl;
}

void PrintResult(Node* node) {
    // An recursive way to
    // print the result of a solution.
    if (node != NULL) {
        PrintResult(node -> parent);
        PrintAction(node);
        PrintPuzzle(node -> state_);  // print the puzzle state
    }
}


// bfs.cpp
// An implementation of Breadth-first Search

#include "Node.h"
#include <iostream>
#include <queue>
#include <fstream>

using namespace std;

// the number of nodes generated is a way to measure time complexity
int nodes_generated;

// an iterative way to implement breadth first search
// Breadth first search
// return a node which contains the goal state
Node* Bfs(Node* root) {
    queue<Node*> fringe_list;
    if (root -> IsGoal()) {
        return root;
    } else {
        fringe_list.push(root);
```

```
    }
    while(!fringe_list.empty()) {
        Node* front = fringe_list.front();
        fringe_list.pop();
        // expand the node, add its children to fringe_list
        for (int i = 1; i <= 4; i++) {
            if (IsValidMove(i, front)) {
                nodes_generated ++;  // update time complexity
                Node* child = MoveAgent(i, front);
                if (child -> IsGoal()) {
                    return child;
                }
                fringe_list.push(child);
            }
        }
    }
    return NULL;
}


int main() {
    int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {1, 2, 3, -1}
    };
    Node* root = new Node(puzzle, 0);
    // Run bfs.
    nodes_generated = 1;
    Node* solution = Bfs(root);
    cout << "Breadth-first Search:" << endl;
    if (solution != NULL) {
        // cout << "Action sequence: ";
        // PrintResult(solution);
        // cout << endl;
        cout << "Path cost: " << solution -> path_cost_ << endl;
        printf("Nodes expanded: %d\n", nodes_generated);
    }

    /* Run bfs by controling the depth of the solution
    and output the numbers of nodes to a file. */
    // int op_action_path[] = {1, 3, 3, 2, 3, 1, 4, 2, 4, 1, 1, 3, 2, 3};
    // ofstream nodes_log;
    // nodes_log.open("bfs.txt");
    // Node* tmp_root;
    // for (int i = -1; i < 14; i++) {
    //  if (i == -1) {
    //      // stay at the original location
    //      tmp_root = root;
    //  } else {
```

```
    //      // take a step to the goal state
    //      tmp_root = MoveAgent(op_action_path[i], tmp_root);
    //  }
    //  nodes_generated = 1;  // nodes_generated is a global variable
    //  Node* tmp_solution = Bfs(tmp_root);
    //  printf("Nodes generated: %d\n", nodes_generated);
    //  nodes_log << nodes_generated << endl;
    // }
    // nodes_log.close();
    return 0;

}



// dfs.cpp
// An implementation of Depth-first Search

#include "Node.h"
#include <iostream>
#include <stack>

using namespace std;

// the number of nodes generated is a way to measure time complexity
int nodes_generated;

// an recursive way to implement depth first search
Node* Dfs(Node *node) {
    if (node -> IsGoal())
        return node;
    for (int i = 1; i <= 4; i++) {
        if (IsValidMove(i, node)) {
            nodes_generated ++;
            Node* child = MoveAgent(i, node);
            PrintPuzzle(child -> state_);
            printf("\n");
            // Call the depth first search recursively,
            // and in this problem, this will be an endless loop.
            // There is not a chance that i will change from 1 to 2
            // in the outermost layer.
            Node* solution = Dfs(child);
            if (solution != NULL)
                return solution;
        }
    }
    return NULL;
}

// an iterative way to implement depth first search
// Node* Dfs(Node *root) {
```

```
//  stack<Node*> fringe_list;
//  fringe_list.push(root);
//  while(!fringe_list.empty()) {
//      Node* top = fringe_list.top();
//      fringe_list.pop();
//      if (top -> IsGoal()) {
//          return top;  // find the solution
//      } else {
//          // expand the node, add its children to fringe_list
//          for (int i = 1; i <= 4; i++) {
//              if (IsValidMove(i, top)) {
//                  nodes_generated ++;  // update time complexity
//                  Node* child = MoveAgent(i, top);
//                  fringe_list.push(child);
//              }
//          }
//      }
//  }
//  return NULL;
// }

int main() {
    // int puzzle[4][4] = {
    //  {0, 1, 0, 0},
    //  {0, 2, 0, 0},
    //  {0, 3, 0, 0},
    //  {0, -1, 0, 0}
    // };
    int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {1, 2, 3, -1}
    };
    Node* root = new Node(puzzle, 0);
    nodes_generated = 1;
    Node* solution = Dfs(root);
    if (solution != NULL) {
        cout << "Path cost: " << solution -> path_cost_ << endl;
        printf("Nodes expanded: %d\n", nodes_generated);
    }

    return 0;
}


// ids.cpp
// An implementation of Iterative Deepening Search

#include "Node.h"
#include <iostream>
```

```
#include <queue>
#include <fstream>
using namespace std;

// the number of nodes generated is a way to measure time complexity
int nodes_generated;

// an recursive way to implement limited-depth first search
Node* DfsLimited(Node *node, int limit) {
    if (node -> IsGoal())
        return node;   // the solution is found
    if (limit == 0)
        return NULL;   // the limited depth is reached while no solution is found
    for (int i = 1; i <= 4; i++) {
        if (IsValidMove(i, node)) {
            nodes_generated ++;  // update time complexity
            Node* child = MoveAgent(i, node);
            Node* result = DfsLimited(child, limit - 1);
            if (result != NULL) {
                return result;  // return the solution
            }
        }
    }
    return NULL;  // the moves of all directions are invalid
}


int main() {
    int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {1, 2, 3, -1}
    };
    Node* root = new Node(puzzle, 0);
    // /* dfs iterative */
    nodes_generated = 1;
    for (int i = 0; ;i++) {
        Node* solution = DfsLimited(root, i);
        if (solution != NULL) {
            cout << "Iterative Deepening Search:" << endl;
            // cout << "Action sequence: ";
            PrintResult(solution);
            cout << endl;
            printf("Path cost: %d\n", solution -> path_cost_);
            // printf("Current depth: %d\n", i);
            printf("Nodes generated: %d\n", nodes_generated);
            // the solution has been found
            // the depth stopped increasing
            break;
        }
```

```
    }


    /*Run idfs by controling the depth of the solution
    and output the numbers of nodes to a file.
    */
    // int op_action_path[] = {1, 3, 3, 2, 3, 1, 4, 2, 4, 1, 1, 3, 2, 3};
    // ofstream nodes_log;
    // nodes_log.open("idfs.txt");
    // Node* tmp_root;
    // for (int i = -1; i < 14; i++) {
    //  if (i == -1) {
    //      // stay at the original location
    //      tmp_root = root;
    //  } else {
    //      // take a step to the goal state
    //      tmp_root = MoveAgent(op_action_path[i], tmp_root);
    //  }
    //  nodes_generated = 1;   // nodes_generated is a global variable
    //  for (int j = 0; ;j++) {
    //      Node* solution = DfsLimited(tmp_root, j);
    //      if (solution != NULL) {
    //          nodes_log << nodes_generated << endl;
    //          printf("Current depth: %d\n", solution -> path_cost_);
    //          printf("Nodes generated: %d\n", nodes_generated);
    //          // the solution has been found
    //          // the depth stopped increasing
    //          break;
    //      }
    //  }
    // }
    // nodes_log.close();

    return 0;
}


// Node.h
// the head file for A* Search

#ifndef NODE_H
#define NODE_H

class Node {
    public:
        Node(){};
        Node(int puzzle[4][4], int depth);
        int* GetAgentLocation();
        bool IsGoal();
        int GetH1Cost();
        int state_[4][4];
```

```
        Node* parent;
        int depth_;
        int action_;
        int g_;
        int h_;
};

void PrintAction(const Node* node);
void PrintResult(Node* node);
void PrintPuzzle(const int puzzle[4][4]);
bool IsValidMove(int direction, Node *current);
Node* MoveAgent(int direction, Node* current);
void Swap(int&, int&);

#endif
```

```
// Node.cpp
// An implementation file for "Node.h"
// A* search

#include <iostream>
#include <cmath>
#include "Node.h"
using namespace std;

// initialize a node with the state of the puzzle
// and its depth in the tree structure
Node::Node(int puzzle[4][4], int depth) {
    parent = NULL;
    depth_ = depth;
    action_ = 0;   // action is set to 0 when a Node is initialized
    g_ = 0;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            state_[i][j] = puzzle[i][j];
        }
    }
    h_ = GetH1Cost();  // get the heuristic cost after the state is set
}

// use the sum of manhattan distances
// as the cost from the current state to the goal
int Node::GetH1Cost() {
    // The goal location of block A, B and C:
    // A: {1, 1}
    // B: {2, 1}
```

```
    // C: {3, 1}
    // The current location of A, B and C:
    int A_loc[2];
    int B_loc[2];
    int C_loc[2];
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if(state_[i][j] == 1) {
                A_loc[0] = i;
                A_loc[1] = j;
            } else if (state_[i][j] == 2) {
                B_loc[0] = i;
                B_loc[1] = j;
            } else if (state_[i][j] == 3) {
                C_loc[0] = i;
                C_loc[1] = j;
            }
        }
    }
    int h_cost = abs(A_loc[0] - 1) + abs(A_loc[1] - 1) + abs(B_loc[0] - 2) +
abs(B_loc[1] - 1)
            + abs(C_loc[0] - 3) + abs(C_loc[1] - 1);
    return h_cost;
}

// return true if the node contains the goal state
bool Node::IsGoal() {
    // goal state:
    //  {0, 0, 0, 0},
    //  {0, 1, 0, 0},
    //  {0, 2, 0, 0},
    //  {0, 3, 0, 0}
    return (state_[1][1] == 1 && state_[2][1] == 2 && state_[3][1] == 3);
}

int* Node::GetAgentLocation() {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            if (state_[i][j] == -1) {
                int *res = new int[2];
                res[0] = i;
                res[1] = j;
                return res;
            }
        }
    }
    return NULL;
}

void PrintAction(const Node* node) {
    int act = node -> action_;
```

```
    cout << "Action: ";
    switch(act) {
        case 0: break;
        case 1:
            cout << "up";
            break;
        case 2:
            cout << "down";
            break;
        case 3:
            cout << "left";
            break;
        case 4:
            cout << "right";
            break;
    }
    cout << endl << endl;
}

void PrintResult(Node* node) {
    if (node != NULL) {
        PrintResult(node -> parent);
        PrintAction(node);
        PrintPuzzle(node -> state_);
    }
}

// A move is valid as long as the agent will not cross the boundaries.
bool IsValidMove(int direction, Node *current) {
    /*
    direction:
    1: up
    2: down
    3: left
    4: right
    */
    int* agent = current -> GetAgentLocation();
    int result;
    switch(direction)
    {
        case 1:
            result = agent[0] - 1;
            break;
        case 2:
            result = agent[0] + 1;
            break;
        case 3:
            result = agent[1] - 1;
            break;
        case 4:
            result = agent[1] + 1;
```

```
            break;
    }
    return (result >= 0 && result < 4);
}

void PrintPuzzle(const int puzzle[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            printf("%d ", puzzle[i][j]);
        }
        printf("\n");
    }
    // printf("\n");
}

// Move the agent according to the current position and the direction.
// Return a new node after the move.
Node* MoveAgent(int direction, Node* current) {
    int* agent_loc = current -> GetAgentLocation();
    // Initial the representation of the new state
    // using the current state.
    int new_state_[4][4];
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++)
        {
            new_state_[i][j] = current -> state_[i][j];
        }
    }
    // Move the position of the agent.
    switch(direction) {
        case 1:
            Swap(new_state_[agent_loc[0]][agent_loc[1]], new_state_[agent_loc[0] -
1][agent_loc[1]]);
            break;
        case 2:
            Swap(new_state_[agent_loc[0]][agent_loc[1]], new_state_[agent_loc[0] +
1][agent_loc[1]]);
            break;
        case 3:
            Swap(new_state_[agent_loc[0]][agent_loc[1]],
new_state_[agent_loc[0]][agent_loc[1] - 1]);
            break;
        case 4:
            Swap(new_state_[agent_loc[0]][agent_loc[1]],
new_state_[agent_loc[0]][agent_loc[1] + 1]);
            break;
    }
    Node* new_node = new Node(new_state_, current -> depth_ + 1);
    new_node -> parent = current;  // Parent logged.
    new_node -> action_ = direction; // Action logged.
    new_node -> g_ = current -> g_ + 1;
```

```
    return new_node;
}

void Swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}


// SingleLinkedList.h
// An implementation of a single linked list,
// which is used for store the unexpanded nodes in A* Search

#ifndef SLL_H
#define SLL_H

#include<iostream>
using namespace std;

template <class T>
class LinkedNode {
public:
    LinkedNode(T element) {
        value = element;
        next = NULL;
    };
    ~LinkedNode();
    T value;
    LinkedNode* next;
};

template <class T>
class SingleLinkedList {
public:
    SingleLinkedList() {
        head = NULL;
    };
    // ~SingleLinkedList();
    void AddNode(T value);
    void DeleteNode(LinkedNode<T>* node);
    void TraverseList();
    bool IsEmpty();
    LinkedNode<T>* head;
};

template<class T>
bool SingleLinkedList<T>::IsEmpty() {
    return head == NULL;
}
```

```
template<class T>
void SingleLinkedList<T>::DeleteNode(LinkedNode<T>* node) {
    // delete the node with minimal value
    LinkedNode<T>*current = head;
    if (current == node) {
        head = current -> next;
    } else {
        while (current -> next != NULL) {
            if (current -> next == node) {
                current -> next = current -> next -> next;
                break;
            } else {
                current = current -> next;
            }
        }
    }
}

template<class T>
void SingleLinkedList<T>::TraverseList() {
    LinkedNode<T>* tmp = head;
    while (tmp != NULL) {
        cout << tmp -> value << endl;
        tmp = tmp -> next;
    }
}

template <class T>
void SingleLinkedList<T>::AddNode(T value) {
    LinkedNode<T>* new_node = new LinkedNode<T>(value);
    if (head == NULL) {
        head = new_node;
    } else {
        LinkedNode<T>* tmp = head;
        while (tmp -> next != NULL) {
            tmp = tmp -> next;
        }
        tmp -> next = new_node;
    }
}

#endif


// main.cpp
// An implementation of A* Search

#include "SingleLinkedList.h"
#include "Node.h"
#include <iostream>
#include <fstream>
```

```
using namespace std;

Node* AStar(Node* root);
// the number of nodes generated is a way to measure time complexity
int nodes_generated;


// A-star search
Node* AStar(Node* root) {
    SingleLinkedList<Node*> list;
    list.AddNode(root);
    while(!list.IsEmpty()) {
        LinkedNode<Node*>* min = list.head;
        LinkedNode<Node*>* tmp = list.head;
        // get the LinkedNode with minimal value
        while(tmp -> next != NULL) {
            tmp = tmp -> next;
            if ((tmp -> value -> h_ + tmp -> value -> g_) < (min -> value -> h_ +
min -> value -> g_)) {
                min = tmp;
            }
        }
        // delete the LinkedNode with the minimal value from the list
        list.DeleteNode(min);
        Node* current_min_node = min -> value;
        if (current_min_node -> IsGoal()) {
            return current_min_node;
        }  else {
            // expand the node, add its children to fringe_list
            for (int i = 1; i <= 4; i++) {
                if (IsValidMove(i, current_min_node)) {
                    nodes_generated ++;  // update time complexity
                    Node* child = MoveAgent(i, current_min_node);
                    list.AddNode(child);
                }
            }
        }
    }
    return NULL;
}


int main() {
    /*
    direction:
    1: up
    2: down
    3: left
    4: right
    */
```

```
    // 0: white tiles; -1: the agent;
    // 1: block A; 2: block B; 3: block C
    int puzzle[4][4] = {
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {1, 2, 3, -1}
    };

    Node* root = new Node(puzzle, 0);
    // run Astar
    nodes_generated = 1;
    Node* solution = AStar(root);
    if (solution != NULL) {
        cout << "A* Heuristic Search:" << endl;
        // cout << "Action sequence: ";
        PrintResult(solution);
        cout << endl;
        printf("Path cost: %d\n", solution -> g_);
        printf("Nodes generated: %d\n", nodes_generated);
    }

    /* Find the optimal action path by experiments
    and use it to control the depth of the solution. */
    /* Run Astar by controling the depth of the solution
    and output the numbers of nodes to a file. */
    // int op_action_path[] = {1, 3, 3, 2, 3, 1, 4, 2, 4, 1, 1, 3, 2, 3};
    // ofstream nodes_log;
    // nodes_log.open("nodes.txt");
    // Node* tmp_root;
    // for (int i = -1; i < 14; i++) {
    //  if (i == -1) {
    //      // stay at the original location
    //      tmp_root = root;
    //  } else {
    //      // take a step to the goal state
    //      tmp_root = MoveAgent(op_action_path[i], tmp_root);
    //  }
    //  nodes_generated = 1;  // nodes_generated is a global variable
    //  Node* tmp_solution = AStar(tmp_root);
    //  printf("Nodes generated: %d\n", nodes_generated);
    //  nodes_log << nodes_generated << endl;
    //  // root = tmp_root;
    // }
    // nodes_log.close();

    return 0;
}
```

Using neural networks for learning heuristics

1 Abstract

In A* search algorithm, a heuristic function is needed to compute the estimated cost from the current state to the goal state. This report explores the use of neural networks in the role of heuristic functions. Since neural networks are good at constructing learning algorithms, it is likely that an A* search with the heuristic function learned by the neural networks will give a better performance than a normal A* search.

2 Introduction

The state space grows significantly in many search-based algorithms. Although there are different kinds of methods to measure the heuristic costs, they might not be accurate enough and can be improved by learning from experience. There is a good chance that by using neural networks, a more efficient heuristic function can be learned and thus the computational time and state space needed for solving the problem will be reduced. To demonstrate this method, the eight-puzzle problem will be used.

3 Methodology

3.1 A* search

To learn from experience, firstly an A* search program was used to find the optimal solution and collect samples of states from the solution path and the actual cost of the solution from that point. In the problem of eight-puzzle, the blank tile can be considered as the agent that is moving around. The cost of one move made by the agent was considered as one.

Usually a tree search does not keep track of visited nodes and it is possible that a state will be visited for infinitive times. As we need to run A* for many times to prepare a training set, the agent was designed not be allowed to go back to the last node it visited to reduce time complexity. To improve the performance of searching for the next node to be expanded, a priority queue was used in the A* implementation.

The A* search was applied to the training part as well as the testing part. For the training part, it was used to prepare a training dataset. For the testing part, the heuristic functions learned by the neural networks were applied to the A* search and a number of randomly generated 8-puzzles were used for testing.

3.2 Construction of training dataset

To feed the neural networks, we need to collect features of the states as well as their target values. The actual path cost from the current state to the goal is considered to be the target. A learning algorithm will work better if it is provided with features that are relevant to predicting the actual costs. Consider "the number of misplaced tiles" as $h_1(n)$ and "the sum of Manhattan distances" as $h_2(n)$. They are admissible heuristics and for every node n, $h_2(n) \geq h_1(n)$, which means h_2 dominates h_1 and is better for search. Take these two heuristics as features. There is a chance that when combining them in some way, a better heuristic can be learned by the neural networks.

To get enough data for feeding the neural networks, we need to run A* search for enough times with different initial states of eight-puzzle. In addition, the eight-puzzle is only solvable when the number of inversions is not odd. To achieve this, A* search was run with all the possible cases of eight-puzzles until it solved the puzzle for 1000 times. In terms of each time

when A* found a solution, it collected features from the solution path and the actual cost of the solution from that point.

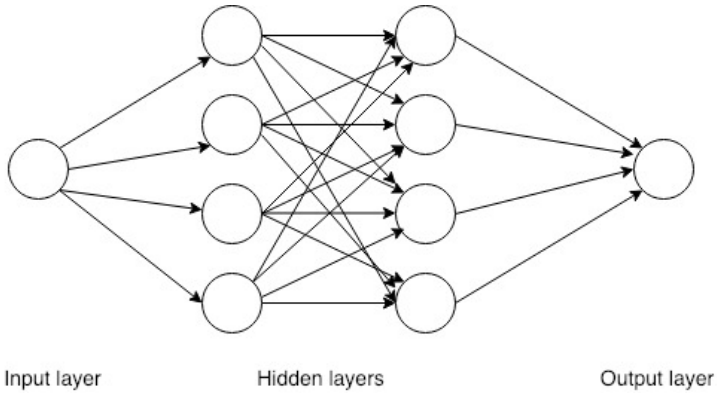
One sample of data consists of the sum of Manhattan distances and the number of misplaced tiles at that point as well as the actual cost. The average cost was computed for every configuration of features.

The following pseudo code demonstrates how the training dataset was built. The parameter of "n_sample" refers to the number of different 8-puzzles to be solved.

```
Function construct_training_set(n_sample) returns a dictionary
    Generate all the permutations of an eight-puzzle and shuffle them
    Initialize a dictionary structure
    Initialize count = 0
    For each permutation do
        If the puzzle is solvable do
            Count += 1
            Use A* to find the optimal solution
            For each state in the solution path do
                Compute the sum of Manhattan distances and the number of misplaced tiles
                and use them as an item key of the dictionary
                Compute the actual cost and append the value to the corresponding key
            If count == n_sample: break
    Compute the average value for every key of the dictionary
```

3.3 The design of neural networks

The neural networks were designed to have the following structure except that the number of neurons in the input layer is flexible, which will be discussed later.



There are two hidden layers and one output layer. The activation functions used in the hidden layers were sigmoid. Since it is a regression problem, no activation function was used in the output layer. All the outputs from the last hidden layer were combined linearly and sent to the output layer. In each layer, a different bias was used to improve the fitting result. The quadratic cost function was used in the output layer to measure the difference between the true cost and the cost predicted by the neural networks.

During the training process, the input layer was fed with features that are relevant to predicting the true cost from the concerned state to the goal while the output layer was fed with the actual cost provided by the optimal solution of A* search. The initial weights and bias in each layer were sampled from a uniform distribution over [-0.5, 0.5] and they were

updated by backpropagation. A matrix-based method was used to compute the outputs of neural networks.

During the testing process, a trained model was used as the heuristic function for A* search algorithm. The new A* search was run with 200 randomly generated eight-puzzles that are solvable. The nodes expanded by the new A* search was compared with the normal A* search which used the sum of Manhattan distances as the heuristic function.

As for the input layer, two different types of inputs were used. One was a single feature of the sum of Manhattan distances. The other was two features which consist of the sum of Manhattan distances and the number of misplaced tiles. Thus, two models were trained in the training process and evaluated in the testing process.

4 Results

There was a training and testing part in building a neural networks model. After that, a model with the best parameters was trained and applied in the testing of the eight-puzzle problem.

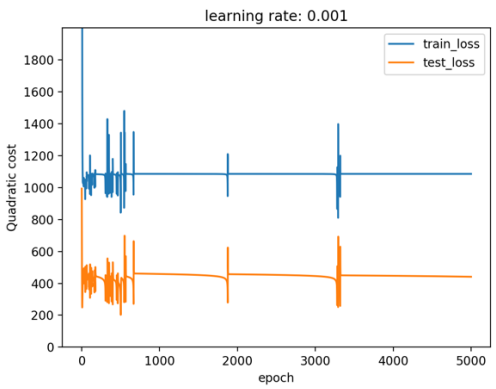
4.1 Training and testing of the neural networks' models

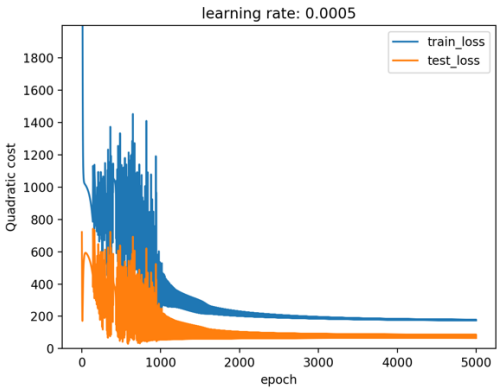
By solving 1000 different 8-puzzles, 76 samples of data for training were collected. The number of samples is a bit small because there are not many combinations of features in all possible cases of eight-puzzle. But using a neural networks model with two hidden layers can still fit the training data well.

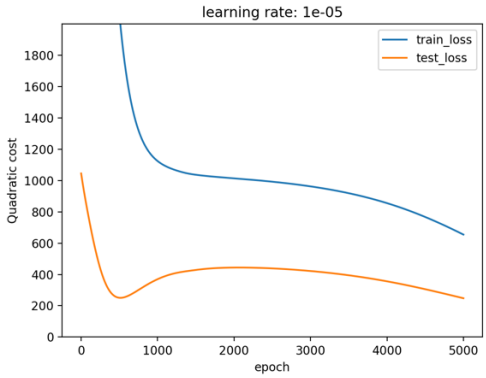
The training data was split into two parts. One fifth of the data were used for testing and the rest were used for training. The number of iterations was set to 5000 and the candidate values for the learning rate were set to 0.001, 0.0005, 0.00001. The quadratic cost was used to record the errors of the training set and testing set.

4.1.2 A neural networks model with one feature

The sum of Manhattan distances was chosen as the feature in the input layer. The results are shown below with different learning rates.



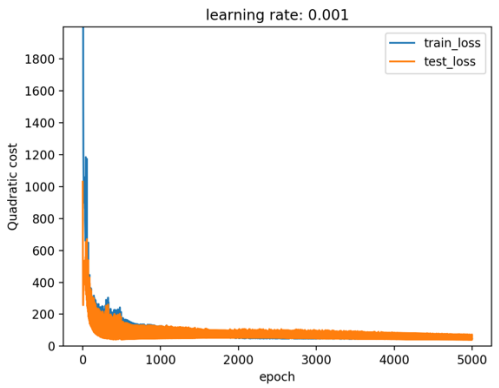


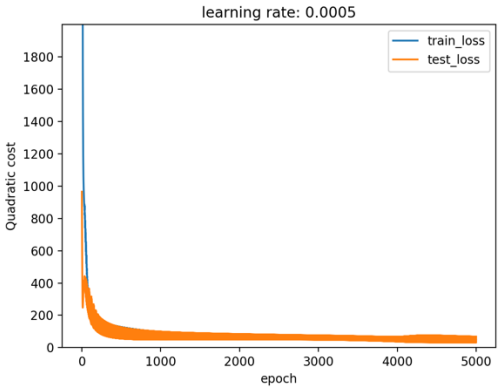


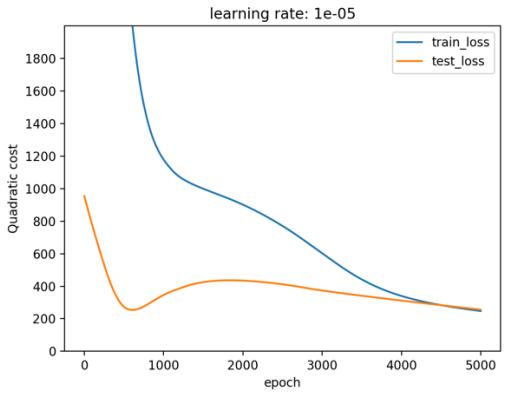
When the learning rate is 0.001, the train loss and test loss were not even converging. When the learning rate is 0.0005, the train loss and test loss decreased gradually and converged at the epoch of around 2000. When the learning rate is 0.00001, the train loss and test loss continued to decrease at the epoch of 5000 and did not converge. Therefore, the learning rate of 0.0005 was chosen to train a model with the whole dataset.

4.1.3 A neural networks model with two features

The sum of Manhattan distances and the number of misplaced tiles at different states were used as features in the input layer. The results are shown below with different learning rates.



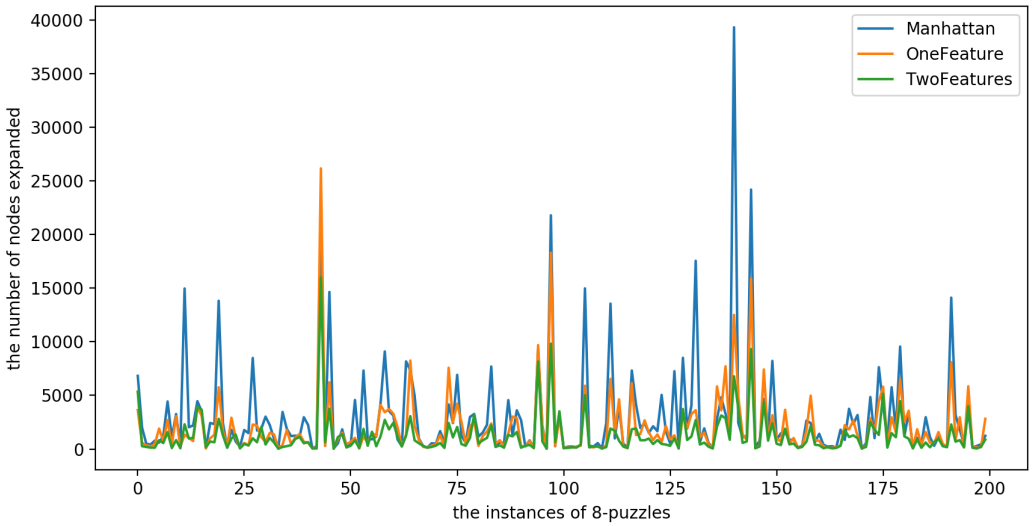




When the learning rate is 0.001 or 0.0005, the errors on the training set and testing set decreased continuously and then converged but the overall errors seem to be lower when the rate is 0.0005. When the learning rate is 0.00001, both errors did not converge before the epoch of 5000. Therefore, the learning rate of 0.0005 was chosen as the best parameter. A new model trained with the whole dataset was built.

4.2 Testing of eight-puzzles

Two hundred randomly generated 8-puzzles were used to compare the performance of a new A* search that has a heuristic function learned by the neural networks with the performance of a normal A* search that used the sum of Manhattan distances as the heuristic value. The performance was measured by the number of nodes expanded when a solution was found. The result is shown as below.



The x axis refers to different instances of the 8-puzzle problem. The y axis is the number of nodes expanded for every instance. The “Manhattan” label refers to the A* search using the sum of Manhattan distances as the heuristic value. The “OneFeature” label is the result of A* search using the heuristic function learned by the one-feature neural networks mentioned in 4.1.2. The “TwoFeatures” label is used when a heuristic learned by the two-features neural networks mentioned in 4.1.3 was applied to A* search.

From the diagram above, it is easy to find that in most cases, the A* search with a heuristic learned by the two-features neural networks has the least number of nodes expanded when a solution is found, although its performance is close to the A* search with the one-feature model. Without the use of neural networks, in some cases the number of nodes expanded was much bigger.

5 Limitations

The number of samples collected for training neural networks was a bit small but this problem is limited to the maximum feature space. More features could be used to learn a better heuristic function. To collect a larger number of training data, a possible method is to use a search problem which is more sophisticated than the eight-puzzle.

In addition, a graph method could be used to implement A* search which may make it faster.

6 Conclusion

From the experiments, we can find that by using neural networks to learn a heuristic function, the A* search can be more efficient. And it is possible that if we choose appropriate features for feeding the neural networks, a model with a better performance can be built when features are combined.

7 References

[1] Russell, Stuart J., and Peter Norvig. Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.
[2] Škutová, Jolana. "Weights initialization methods for mlp neural networks." (2008).
[3] Chen, Hung-Che, and Jyh-Da Wei. "Using neural networks for evaluation in heuristic search algorithm." Twenty-Fifth AAAI Conference on Artificial Intelligence. 2011.
[4] Walczak, Steven, and Narciso Cerpa. "Heuristic principles for the design of artificial neural networks." Information and software technology 41.2 (1999): 107-117.

8 Code

```
"""
Astar.py

The implementation of A* search for the problem of 8-puzzle.
"""

import numpy as np
import pickle
from queue import PriorityQueue

'''
goal:
    [[0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]]
'''
goal_state = {1:[0, 1], 2:[0, 2], 3:[1, 0],
              4:[1, 1], 5:[1, 2], 6:[2, 0],
              7:[2, 1], 8:[2, 2]}
goal = np.arange(9).reshape([3, 3])


class Node:

    # the neural networks model used to predict the heuristic value
    model = None
    # the flag that implied which heuristic function will be used
    # heuristic = 1 --- use the one-feature neural networks model
    # heuristic = 2 --- use the two-features neural networks model
    # heuristic = None --- use the sum of Manhattan distances
    heuristic = None

    def __init__(self, puzzle):
        '''
        goal:
        [[0, 1, 2],
         [3, 4, 5],
         [6, 7, 8]]
        '''
        self.state = puzzle
        self.action = 0
```

```
        self.parent = None
        if Node.heuristic == 1:
            self.h = Node.model.predict(self.get_H1())
        elif Node.heuristic == 2:
            self.h = Node.model.predict([self.get_H1(), self.get_H2()])
        else:
            self.h = self.get_H1()
        self.g = 0

    def __lt__(self, other):
        '''
        Used for sorts the nodes in the priority queue
        '''
        return (self.g + self.h) < (other.g + other.h)

    def get_H1(self):
        '''
        compute the sum of manhattan distances
        '''
        distances_sum = 0
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != 0:
                    x, y = goal_state[self.state[i][j]]
                    distances_sum += abs(x-i) + abs(y-j)
        return distances_sum

    def get_H2(self):
        '''
        compute the number of misplaced tiles
        '''
        count = 0
        for tile, loc in goal_state.items():
            if self.state[loc[0]][loc[1]] != tile:
                count += 1
        return count

    def is_goal(self):
        return np.array_equal(self.state, goal)

    def get_blank_tile_loc(self):
        '''
        get the location of the blank tile
        '''
        for i in range(3):
            for j in range(3):
                if self.state[i][j] == 0:
                    return i, j


def is_valid_move(node, direction):
```

```
    '''
    1 up, 2 down, 3 left, 4 right

    The move is valid when the blank tile will not cross the boundaries
    and it will not go back to the position it last visited.
    '''
    x, y = node.get_blank_tile_loc()
    if_last_visited = direction + node.action
    if direction == 1:
        change = x - 1
    elif direction == 2:
        change = x + 1
    elif direction == 3:
        change = y - 1
    elif direction == 4:
        change = y + 1
    if node.action == 0:  #  action = 0 means it is the root node
        return change >=0 and change < 3
    else:
        return if_last_visited != 3 and if_last_visited != 7 and change >=0 and
change < 3

def move(node, direction):
    '''
    Move the node to the specified direction.
    Return a new node.
    '''
    state = node.state.copy()  # make a copy of the state
    x, y = node.get_blank_tile_loc()
    if direction == 1:
        state[x][y], state[x - 1][y] = state[x - 1][y], state[x][y]
    elif direction == 2:
        state[x][y], state[x + 1][y] = state[x + 1][y], state[x][y]
    elif direction == 3:
        state[x][y], state[x][y - 1] = state[x][y - 1], state[x][y]
    elif direction == 4:
        state[x][y], state[x][y + 1] = state[x][y + 1], state[x][y]
    new_node = Node(state)
    new_node.parent = node
    new_node.action = direction
    new_node.g = node.g + 1
    return new_node

def Astar(root):
    '''
    The A* search.

    :return: the node with the goal state and the number of nodes expanded.
    '''
    queue = PriorityQueue()
    nodes_expanded = 1
```

```
    queue.put(root)
    while(not queue.empty()):
        min_f_node = queue.get()
        if min_f_node.is_goal():
            # print(nodes_expanded)
            return min_f_node, nodes_expanded
        for i in range(1, 5):
            if is_valid_move(min_f_node, i):
                nodes_expanded += 1
                child = move(min_f_node, i)
                queue.put(child)
    return None

def load_nn_model(choice):
    '''
    Load a neural networks model that has been trained.
    The training process is in the file of "train.py".
    '''
    if choice == 1:
        Node.heuristic = 1
        with open('model_1', 'rb') as f:
            model = pickle.load(f)
    elif choice == 2:
        Node.heuristic = 2
        with open('model_2', 'rb') as f:
            model = pickle.load(f)
    Node.model = model


"""
construct_training_set.py

Construct a training set for neural networks.
"""

import numpy as np
import pickle
import time
from itertools import permutations
from random import shuffle
from Astar import Astar, Node

def check_solvable(puzzle):
    '''
    The 8-puzzle is solvable when the number of inversions is not odd.
    '''
    puzzle = list(puzzle)
    puzzle.remove(0)  # remove the blank tile
    sum_inversion = 0
    for i in np.arange(8):
        for j in np.arange(i + 1, 8):
```

```
            if puzzle[i] > puzzle[j]:
                sum_inversion += 1
    return sum_inversion % 2 == 0

def construct_training_set(n_try):
    '''
    Build a tranining set which is sampled from the instances of 8-puzzle.

    :param int n_try: the number of 8-puzzle instances to be solved.
    '''
    start = time.time()
    print('Total:', n_try)
    puzzle_l = np.arange(9)
    # get all the possible states of 8-puzzle
    perms = list(permutations(puzzle_l))
    # shuffel the permutations
    shuffle(perms)
    data = dict()
    count = 0  # count the number of tries when the puzzle is solvable
    for _ in range(len(perms)):
        if count % 100 == 0:
            print('Current process:', count)
        if count == n_try: break
        element = perms.pop()  # get an initial state of 8 puzzle
        if check_solvable(element):
            count += 1
            puzzle = np.array(element).reshape([3, 3])
            root = Node(puzzle)
            solution = Astar(root)[0]
            if solution != None:
                # record the path cost of the solution
                full_cost = solution.g
                while(solution.parent != None):
                    # record the features of the states of the solution path
                    solution = solution.parent
                    sum_of_mattan_dis = solution.get_H1()
                    number_of_misplaced = solution.get_H2()
                    if data.get((sum_of_mattan_dis, number_of_misplaced)):
                        if len(data[(sum_of_mattan_dis, number_of_misplaced)]) >
20:
                            continue
                    cost = full_cost - solution.g
                    try:
                        data[(sum_of_mattan_dis, number_of_misplaced)].append(cost)
                    except:
                        data[(sum_of_mattan_dis, number_of_misplaced)] = [cost]
    # get the average cost of same initial heuristic values
    for key, value in data.items():
        data[key] = np.average(value)
    print(data)
    # save the dataset as a dictionary
```

```
    with open('dataset', 'wb') as f:
        pickle.dump(data, f)
    print(len(data.keys()))
    print('Dataset completed! Time elapsed: %.2f s' % (time.time() - start))


if __name__ == '__main__':
    construct_training_set(1000)




"""
Filename: MLP.py

An implementation of Multi-layer Perceptrons.

"""

import numpy as np
import pickle


def neuron_layer(number_of_neurons, number_of_inputs):
    '''
    Generate initial weights which are sampled
    from a uniform distribution in the range of [-0.5, 0.5].
    '''
    return 1 * np.random.rand(number_of_neurons, number_of_inputs) - 0.5


class MLP:

    def __init__(self, hidden1, hidden2, output, learning_rate=0.0005):
        self.hidden1_w = hidden1
        self.hidden2_w = hidden2
        self.output_w =  output
        self.bias_1, self.bias_2, self.bias_3 = 1 * np.random.rand(3) - 0.5
        self.lr = learning_rate

    def feed_forward(self, inputs):
        self.layer_1 = self.sigmoid(np.dot(inputs, self.hidden1_w.T) + self.bias_1)
        self.layer_2 = self.sigmoid(np.dot(self.layer_1, self.hidden2_w.T) +
self.bias_2)
        self.output = np.dot(self.layer_2, self.output_w.T) + self.bias_3

    def back_propagate(self, inputs, targets):
        output_err = -(targets - self.output)
        layer_2_err = output_err.dot(self.output_w)
        layer_2_delta = layer_2_err * self.sigmoid_derivative(self.layer_2)
        layer_1_err = layer_2_delta.dot(self.hidden2_w)
        layer_1_delta = layer_1_err * self.sigmoid_derivative(self.layer_1)
```

```
        # compute the gradients of weights
        w_output_grad = output_err.T.dot(self.layer_2)
        w_2_grad = layer_2_delta.T.dot(self.layer_1)
        w_1_grad = layer_1_delta.T.dot(inputs)
        # compute the gradients of bias
        b3_g = np.sum(output_err, axis=0)
        b2_g = np.sum(layer_2_delta, axis=0)
        b1_g = np.sum(layer_1_delta, axis=0)
        return w_1_grad, w_2_grad, w_output_grad, b1_g, b2_g, b3_g

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def train(self, train_X, train_y, test_X, test_y, epoch=5000):
        '''
        Train with a part of the dataset and test with the rest.
        Record the train loss and test loss.
        '''
        alpha = self.lr
        # record the train errors and test errors
        train_loss_history = []
        test_loss_history = []
        for _ in range(epoch):
            self.feed_forward(train_X)
            # compute the quadratic cost
            train_loss_history.append(0.5 * np.square(train_y - self.output).sum())
            test_loss_history.append(0.5 * np.square(test_y -
self.predict(test_X)).sum())
            w_1_grad, w_2_grad, w_output_grad, b1_g, b2_g, b3_g =
self.back_propagate(train_X, train_y)
            # update the weights and bias
            self.hidden1_w -= alpha * w_1_grad
            self.hidden2_w -= alpha * w_2_grad
            self.output_w -= alpha * w_output_grad
            self.bias_1 -= alpha * b1_g
            self.bias_2 -= alpha * b2_g
            self.bias_3 -= alpha * b3_g
        return train_loss_history, test_loss_history

    def train_all(self, train_X, train_y, file_name, epoch=5000):
        '''
        Train with the whole dataset
        and store the model as a local file for further use.
        '''
        alpha = self.lr
        train_loss_history = []
        for _ in range(epoch):
            self.feed_forward(train_X)
```

```
            # compute the quadratic cost
            train_loss_history.append(0.5 * np.square(train_y - self.output).sum())
            w_1_grad, w_2_grad, w_output_grad, b1_g, b2_g, b3_g =
self.back_propagate(train_X, train_y)
            self.hidden1_w -= alpha * w_1_grad
            self.hidden2_w -= alpha * w_2_grad
            self.output_w -= alpha * w_output_grad
            self.bias_1 -= alpha * b1_g
            self.bias_2 -= alpha * b2_g
            self.bias_3 -= alpha * b3_g
        with open(file_name, 'wb') as f:
            pickle.dump(self, f)
        return train_loss_history

    def predict(self, x):
        '''
        Predict the output using the trained model.
        '''
        layer_1 = self.sigmoid(np.dot(x, self.hidden1_w.T) + self.bias_1)
        layer_2 = self.sigmoid(np.dot(layer_1, self.hidden2_w.T) + self.bias_2)
        output = np.dot(layer_2, self.output_w.T) + self.bias_3
        # print(output.shape)
        return output


"""
Filename: train.py

Train the neural networks models
"""

from MLP import *
import pickle
import matplotlib.pyplot as plt
import numpy as np

# The dataset will be built
# after running the file of "construct_training_set.py".
with open('dataset', 'rb') as f:
    dataset = pickle.load(f)

def train_test_split(features, targets):
    '''
    split 1/5 of dataset for testing and the rest for training
    '''
    split = int(1/5 * len(features))
    # print(split)
    train_X = features[split:]
    train_y = targets[split:]
    test_X = features[:split]
    test_y = targets[:split]
```

```
    return train_X, train_y, test_X, test_y

def train(num_of_features, alpha, refit=False, filename=None):
    '''
    Train neural networks.

    :param int num_of_features:
        num_of_features = 1: use the sum of Manhattan distances as features in the
input layer;
        num_of_features = 2: use the sum of Manhattan distances and the number of
misplaced tiles
                             as features in the input layer;
    :param float alpha: learning rate
    :param boolean refit: if refit = True, the model will be trained with the whole
dataset.
    :param str filename: if refit = True, a filename is needed to store the trained
model.
    '''
    hidden_layer1 = neuron_layer(number_of_neurons=4,
number_of_inputs=num_of_features)
    hidden_layer2 = neuron_layer(number_of_neurons=4, number_of_inputs=4)
    output_layer =  neuron_layer(number_of_neurons=1, number_of_inputs=4)
    features = []
    targets = []
    if num_of_features == 1:
        for key, value in dataset.items():
            features.append(key[0])
            targets.append(value)
        features = np.atleast_2d(features).T
        targets = np.atleast_2d(targets).T
    elif num_of_features == 2:
        for key, value in dataset.items():
            features.append(np.array(key))
            targets.append(value)
        features = np.array(features)
        targets = np.atleast_2d(targets).T
    if refit:
        nn_model = MLP(hidden_layer1, hidden_layer2, output_layer,
learning_rate=alpha)
        loss_hist = nn_model.train_all(features, targets, filename)
        print(loss_hist[0], loss_hist[-1])
    else:
        # split the dataset into two parts
        # for training the model and evaluating the model performance
        train_X, train_y, test_X, test_y = train_test_split(features, targets)
        print(train_X.shape, train_y.shape)
        print(test_X.shape, test_y.shape)
        mlp = MLP(hidden_layer1, hidden_layer2, output_layer, learning_rate=alpha)
        train_loss, test_loss = mlp.train(train_X, train_y, test_X, test_y)
        plot_train_test_loss(train_loss, test_loss, alpha)
```

```
def plot_train_test_loss(train_loss, test_loss, alpha):
    '''
    Plot the train loss and test loss as the epoch increases
    with the given alpha(learning rate).
    '''
    print(train_loss[0], train_loss[-1])
    print(test_loss[0], test_loss[-1])
    plt.plot(train_loss, label='train_loss')
    plt.plot(test_loss, label='test_loss')
    plt.xlabel('epoch')
    plt.ylabel('Quadratic cost')
    plt.title('learning rate: ' + str(alpha))
    plt.yticks(np.arange(0,2000, 200))
    plt.ylim([0, 2000])
    plt.legend()
    plt.show()


if __name__ == '__main__':
    # Compare train loss with test loss given different learning rates
    # learning_rates = [1e-3, 5e-4, 1e-5]
    # for lr in learning_rates:
    #     print(lr)
    #     train(num_of_features=2, alpha=lr)
    #     train(num_of_features=1, alpha=lr)

    # train and save the model with the best learning rate
    lr = 5e-4
    # print(lr)
    train(num_of_features=1, alpha=lr, refit=True, filename='model_1')
    train(num_of_features=2, alpha=lr, refit=True, filename='model_2')



"""
Filename: test.py

Run tests with the trained neural networks models which will be used in A* search.
"""

import pickle
from Astar import *
import numpy as np
from itertools import permutations
from random import shuffle
import matplotlib.pyplot as plt
from construct_training_set import check_solvable


def test(number_of_test):
    '''
```

```
    Generate a number of random 8-puzzles that are solvable
    and apply them to the new A* search with heuristic functions learned by neural
networks
    and the normal A* search which uses the sum of Manhattan distances as the
heuristic value.

    :param int number_of_test: the number of randomly generated 8-puzzle
    '''
    # generate some 8-puzzles and store them in a list
    puzzle_l = np.arange(9)
    perms = list(permutations(puzzle_l))
    shuffle(perms)
    test_puzzles = []
    while(len(perms) > 0):
        eg = perms.pop()
        if check_solvable(eg):
            puzzle = np.array(eg).reshape([3, 3])
            test_puzzles.append(puzzle)
            if len(test_puzzles) == number_of_test:
                break
    # test the normal A* search
    nodes_list1 = []
    for puzzle in test_puzzles:
        root = Node(puzzle)
        num_of_nodes = Astar(root)[1]
        nodes_list1.append(num_of_nodes)
    # test the A* search which used one-feature neural networks model
    nodes_list2 = []
    load_nn_model(choice=1)
    for puzzle in test_puzzles:
        root = Node(puzzle)
        num_of_nodes = Astar(root)[1]
        nodes_list2.append(num_of_nodes)
    # test the A* search which used two-features neural networks model
    nodes_list3 = []
    load_nn_model(choice=2)
    for puzzle in test_puzzles:
        root = Node(puzzle)
        num_of_nodes = Astar(root)[1]
        nodes_list3.append(num_of_nodes)
    # plot and compare the results
    plt.figure(figsize=[10,5])
    plt.plot(nodes_list1, label='Manhattan')
    plt.plot(nodes_list2, label='OneFeature')
    plt.plot(nodes_list3, label='TwoFeatures')
    plt.xlabel('the instances of 8-puzzles')
    plt.ylabel('the number of nodes expanded')
    plt.legend()
    plt.show()

test(200)
```