8 Tiles Puzzle Solving Using IDS, and A* Search.

Technical Report · October 2016

DOI: 10.13140/RG.2.2.13971.07206

1 author:

Mohammed Al-Rudaini
Jordan University of Science and Technology
21 PUBLICATIONS **26** CITATIONS

SEE PROFILE

8 Tiles Puzzle Solving Using IDS, and A* Search.

Prepared by: Mohammed Abdullah Al-Rudaini

Tools Used:

1. **Microsoft Visual Studio Community 2015:** Version 14.0.25431.01 Update 3
2. **Microsoft .NET Framework:** Version 4.6.01586
3. **Microsoft Visual C++ 2015**

Modifications: (Yellow Highlighted in the code).

1. **Additional State Member:** "`void Heurs();`" to calculate and assign the heuristic cost for the state.
2. **Additional State:** named "start", to keep start state stored, while manipulating the search using "cur" and "temp" states.
3. **The Goal Array:** named "Goal", for using it in "is goal" member function.
4. **Modified "Main" Function.**
5. **Added "PrintPath" Function:** to print all the the solution states from the Goal state to the Start State.
6. **Added "InClosed" Function:** to search for a State in the Closed List.

Requested Code: (Light-Green Highlighted in the code)

1. **State "is_goal" Member Function:** to return "true" if the state is the goal state.
2. **State "<" Operator:** to compare between two states total costs if informed search is used, or between the general costs if normal search is used.
3. **The "IDS" Function:** to execute the Iterative Deepening Search Algorithm.
4. **The "ASTAR" Function:** to execute the A* Search Algorithm.
5. **The "Expand" Function:** to expand to the current state children, adding them to the fringe list, then removing the expanded state from the fringe list.

Problem Full Code:

```
// 8TilesPuzzle.cpp :
// CS762 Advance Artificial Intelligence
// First Semester 2016 / 2017
// Assignment 2 - 3.
// 8 Tiles Puzzle Solving Using IDS, and A* Search.
// Prepared by : Mohammed Abdullah Al - Rudaini
// Student ID : 20153173001
//

#include "stdafx.h" //nedded in MS VC++ 2015
#include <iostream> //cout.
#include <list> //List Container.
#include <algorithm> // swap(a,b).
#include <ctime> // clock()
using namespace std;

const int n = 3; // rows + columns size.
bool random_start = false, generate_start = true; // State Initializer conditions.
bool informed_search = true;     // true: use A*
                                                    // false: use IDS

// State Class Definition.
```

```
class State {
public:
	int A[n][n], g, h, t;  // State Array, General Cost, Heuristic Cost, and Total Cost.
	State *parent; //Parent Pointer.
	State(); //Constructor
	void reset(); // To set the start, cur, temp States...
	bool is_goal(); // true: state is goal.
				// false: state is not goal.
	bool operator==(const State &) const; //Logical equality Operator.
	bool operator<(const State &) const; //Logical Less-Than Operator.
	void print(); //State Array Printing function.
	void Heurs(); //State Heuristic Cost Calculator.
};

int Goal[n][n] = { { 0,1,2 },{ 3,4,5 },{ 6,7,8 } };  //Goal Array
int space = 1, runtime = 1, timer = 0; // Stored States, Runtime, and initial timer
Counters
list< State > closed, fringe; // Tested States, Active States Lists.
State start, cur, temp; // Start, Current, Temporary States.
void IDS(); // IDS Search function.
void Astar(); //A* Search function.
void Expand(); // State Expander function.
void PrintPath(State *s); // Solution Path Print function.
bool InClosed(State &s); // to search for state in the Closed List.

int main() {
	//initializing the start state.
	start.g = 0; // start cost
	start.Heurs(); // calculating the heuristic cost.
	start.t = start.g + start.h; //total cost.
	start.parent = NULL; // no parent for the start state.
	//reading start time.
	timer = (clock() * 1000) / CLOCKS_PER_SEC;
	//selecting the search algorithm.
	if (informed_search)
		Astar();  // executing the a* algorithm
	else
		IDS(); // executing the ids algorithm.
	// little Pause (^_^).
	getchar();
	return 0;
}
// the state constructor.
State::State() {
	reset();
}
// the state initializer.
void State::reset() {
	int i, j, k;
	g = h = t = 0;
	parent = NULL;
	if (generate_start) {
		if (!random_start) {
			A[0][0] = 1; A[0][1] = 2; A[0][2] = 5;
			A[1][0] = 3; A[1][1] = 7; A[1][2] = 8;
			A[2][0] = 4; A[2][1] = 6; A[2][2] = 0;
		}
		else {
			list< int > l;
```

```
                    list< int >::iterator it;
                    srand(time(0));
                    for (i = 0; i < n * n; i++)
                          l.push_back(i);
                    for (i = 0; i < n; i++) {
                          for (j = 0; j < n; j++) {
                                it = l.begin();
                                for (k = rand() % l.size(); k > 0; k--)
                                      it++;
                                A[i][j] = (*it);
                                l.erase(it);
                          }
                    }
              }
              generate_start = false;
        }
}
// state goal tester.
bool State::is_goal() {
        // Your code goes here
        int i, j;
        for (i = 0; i < n; i++) {
              for (j = 0; j < n; j++) {
                    // if any two same positioned items not equal.
                    if (A[i][j] != Goal[i][j])
                          // this state is not the goal.
                          return false;
              }
        }
        //reaching this point means all items equals the goal items.
        //this state is the goal state.
        return true;
}
// state array printer.
void State::print() {
        int i, j;
        for (i = 0; i < n; i++) {
              for (j = 0; j < n; j++)
                    cout << A[i][j] << ' ';
              cout << endl;
        }
        cout << endl;
}
// state logical equality operator.
bool State::operator==(const State &r) const {
        int i, j;
        for (i = 0; i < n; i++) {
              for (j = 0; j < n; j++) {
                    // if any two same positioned items not equal.
                    if (A[i][j] != r.A[i][j])
                          // states are not equal
                          return false;
              }
        }
        //reaching this point means all items in both states are equal.
        // states are equal
        return true;
}
//
```

```
bool State::operator<(const State &r) const {
	// Your code goes here: Done Sir(^_^)
	if (informed_search) { // for heuristic based algorithms.
		return t < r.t;
	}
	else {
		return g < r.g; // for normal search algorithms.
	}
}
// state heuristic cost calculator.
void State::Heurs() {
	int i, i2, j, j2, Heuristic = 0;
	bool found;
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++) {
			found = false;
			for (i2 = 0; i2 < n; i2++) {
				for (j2 = 0; j2 < n; j2++) {
					// finding similar elements.
					if (Goal[i][j] == A[i2][j2]) {
						//Manhaten Based Heuristic displacement cost.
						Heuristic += abs(i - i2) + abs(j - j2);
						found = true;
					}
					if (found)
						break;
				}
				if (found)
					break;
			}
		}
	}
	// setting the state heurisitc cost.
	h = Heuristic;
}
// Iterative deepning search.
void IDS() {
	// Your code goes here, I did it as best as I could (^_^).
	int depth=0; // Depth cost.
	cout << "Starting IDS Algorithm... \n";
	while(true){

		cur = start;
		fringe.push_front(cur);
		while (!fringe.empty())
		{
			// process the fringe states .
			cur = fringe.front();
			// if the front is goal.
			if (cur.is_goal()) {
				// calcilate the search time in msec.
				runtime = ((clock() * 1000) / CLOCKS_PER_SEC) - timer;
				// print the search costs.
				cout << "Time= " << runtime << "\n";
				cout << "Space= " << space << "\n";
				cout << "Cost= " << cur.g << "\n";
				//print the solution path.
				cout << "Path:\n";
				PrintPath(&cur);
```

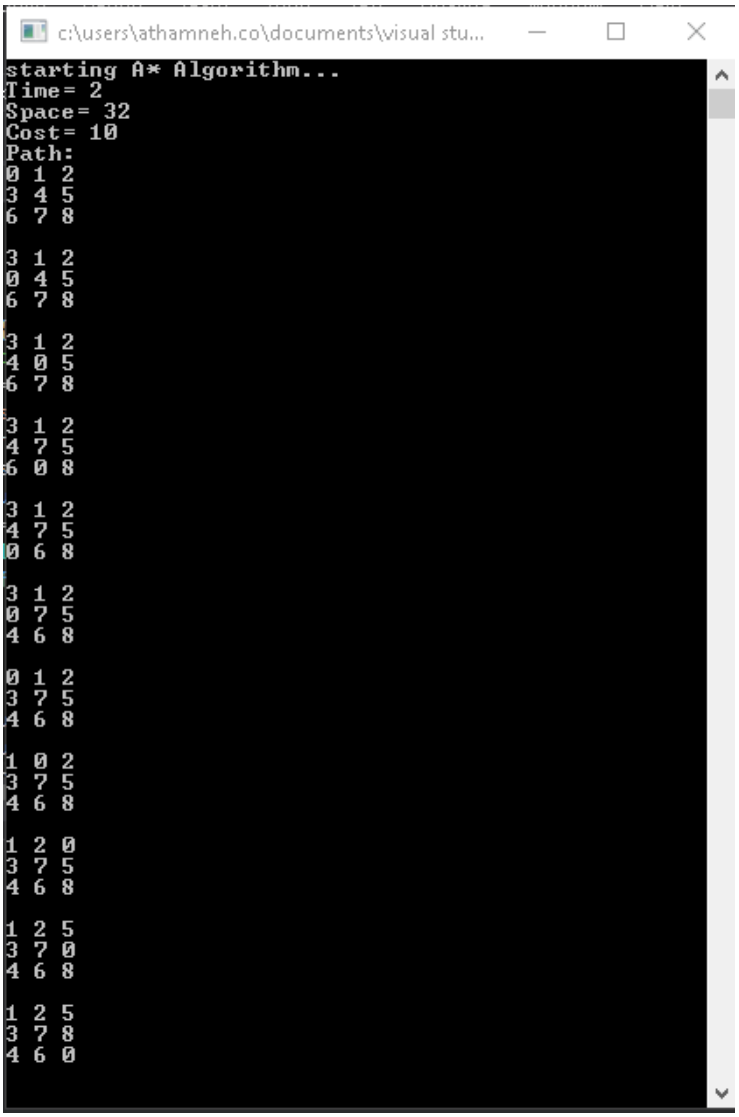
```
                        // exit the function
                        return;
                } //if state not the goal and in the search depth.
                else if (depth > cur.g)
                {
                        //expand the state.
                        Expand();
                }
                else { //not useable state.
                        //pop it out
                        fringe.pop_front();
                }
        }
        // clear both lists for the next round.
        fringe.clear();
        closed.clear();
        // increase the search depth.
        depth++;
    }
}
// A* search algorithm function.
void Astar() {
    // Your code goes here. Roger that (^_^)...
    cout << "starting A* Algorithm... \n";
    cur = start;
    fringe.push_front(cur);
    while (true) {
        // process all states in the fringe.
        cur = fringe.front();
        for (list<State>::iterator it = fringe.begin(); it != fringe.end(); ++it) {
                // find the state with the minimum total cost.
                if ((*it) < cur ) {  //using the state < operator..
                        cur = (*it);
                }
        }
        // if minimal total cost state is the goal.
        if (cur.is_goal()) {
                // calculate and print the search costs.
                runtime = ((clock() * 1000) / CLOCKS_PER_SEC) - timer;
                cout << "Time= " << runtime << "\n";
                cout << "Space= " << space << "\n";
                cout << "Cost= " << cur.g << "\n";
                // print the solution path.
                cout << "Path:\n";
                PrintPath(&cur);
                // exit the function.
                return;
        }// if the minimal total cost state is not the goal.
        else {
                // expand it;
                Expand();
        }
    }
}
```

```
// State Expanding Function.
void Expand() {
	// Your code goes here, Yes, Sir (^_^)..
	//add current state to the closed list.
	closed.push_back(cur);
	int i, j;
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++) {
			//finding the 0 element in the state array.
			if (cur.A[i][j] == 0) {
				// if the 0 not in the first row.
				if (i > 0) {
					// set the child basic elements
					temp = cur;
					temp.parent = &(closed.back());
					// shift the zero element UP..
					swap(temp.A[i][j], temp.A[i - 1][j]);
					// search for the child in the closed list.
					// if the child not found in the closed list.
					if (!InClosed(temp)) {
						//set remaining elements
						temp.g += 1;
						temp.Heurs();
						temp.t = temp.g + temp.h;
						fringe.push_front(temp); //push the shild into the
fringe list
						space++;//increment the space counter
					}
				}
				//if the 0 is not in the last row.
				if (i < n-1) {
					temp = cur;
					temp.parent = &(closed.back());
					//shift the zero element DOWN.
					swap(temp.A[i][j], temp.A[i + 1][j]);
					if (!InClosed(temp)) {
						temp.g += 1;
						temp.Heurs();
						temp.t = temp.g + temp.h;
						fringe.push_front(temp);
						space++;
					}
				}// if the 0 element not in the first column.
				if (j > 0) {
					temp = cur;
					temp.parent = &(closed.back());
					// shift it LEFT.
					swap(temp.A[i][j], temp.A[i][j - 1]);
					if (!InClosed(temp)) {
						temp.g += 1;
						temp.Heurs();
						temp.t = temp.g + temp.h;
						fringe.push_front(temp);
						space++;
					}
				}// if the zero elemnt not in the last column.
				if (j < n-1) {
					temp = cur;
					temp.parent = &(closed.back());
```

```
                                    // shift it RIGHT.
                                    swap(temp.A[i][j], temp.A[i][j + 1]);
                                    if (!InClosed(temp)) {
                                          temp.g += 1;
                                          temp.Heurs();
                                          temp.t = temp.g + temp.h;
                                          fringe.push_front(temp);
                                          space++;
                                    }
                              }
                        }
                  }
            }
      // remove the expanded state from the firinge list.
      fringe.remove(cur);
}
// Recursive Solution Path Printing Function.
void PrintPath(State *s)
{
      // if the start state not reached.
      if (s != NULL) {
            // print current state.
            (*s).print();
            //recursively call printing its parent.
            PrintPath((*s).parent);
      }
}
// Closed List Searching Function.
bool IsInClosed(State &s)
{
      for (list<State>::iterator it = closed.begin(); it != closed.end(); ++it) {
            if ((*it) == s) { //using the State == Operator.
                  return true;
            }
      }
      return false;
}
```

The Output:

1. **A* Search Algorithm:**



2. IDS Search Algorithm: