COMP2208 Intelligent Systems Assignment: Search Methods—Report

Name: Kuan-Yu Li Student ID: 27440656 Email: kl1u13@soton.ac.uk

- Methods and Approaches
  - Languages:
    I use **C++**, and I wrote my codes in object-oriented style. So I can debug and switch between different searching algorithms easily.
  - Main algorithms:
    Breadth-first search, depth-first search, iterative deepening search, A* search
  - Implementation:

```
39  class tile_moving
40      {
41      public:
42          tile_moving();
43          bool depth_first(int level);
44          bool breadth_first();
45          bool iterative_deepening();
46          bool a_star();
47          void output_backtrace_result();
48          void how_many_nodes();
49
50
51      private:
52          int debug;
53          int num_of_named_tile;
54          int width_of_grid;
55          vector<axis> tiles_position;
56          vector<axis> goal_position;
57          bool used_pattern[16][16][16][16];/** type the number by hand **/
58
59          void move_agent_block(vector<axis> &position_arr, int direction);
60          bool check_goal(vector<axis> now);
61          void recover_tiles_position(vector<axis> &record);
62          int manhattan_distance_calculator(bfs_obj* a);
63      }
```

(0, 0)	(1, 0)	(2, 0)	(3, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)
A (0, 3)	B (1, 3)	C (2, 3)	agent (3, 3)

    I labeled every slot in the grid with x-y coordinate, as shown in the table above. Now take a look at the important functions and variables I used. I will explain how I used them.
    - **vector<axis> tiles_position**: Used to store the coordinates of named tiles, for example, A(0, 3), B(1, 3), C(2, 3), and agent(3, 3).
    - **vector<axis> goal_position:** Used to store the coordinates of goal states, for example, A(1, 0), B(1, 1), C(1, 2), and agent(1, 3).
    - **bool used_pattern[16][16][16][16]:** Used to record the grid pattern that appeared before. Each dimension stands for one named tile. I linearize the x-y coordinate into number 0~15. Take the table above as an example, in this case, used_pattern[12][13][14][15] will be assigned TRUE. This array is used to avoid going to same states appeared before. So we won't get stuck with an infinite loop when searching.
    - **void move_agent_block(vector<axis> &position_arr, int direction):** Used to update the vector of tiles' positions when the agent tile made every move. It will check automatically whether the agent is swapping with a blank tile or a named tile.
    - **bool check_goal(vector<axis> now):** Used to check whether we reach the goal state or not, every time going to a new node.
  - Breadth-first search:
    I use a **FIFO queue** to implement the breadth-first search. Its high level structure is like this:

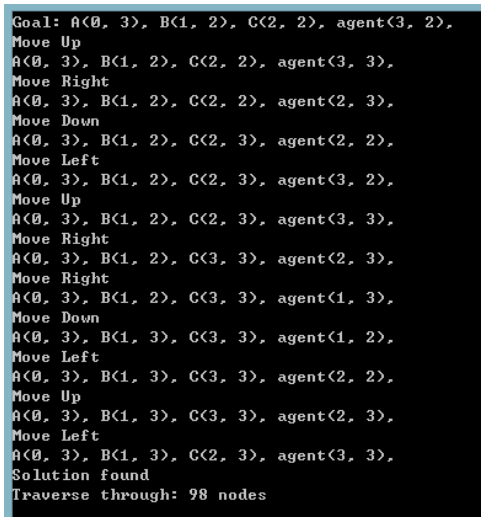
while(queue is no empty)
{
1. Take out the front node.
2. Check if this state appeared before.
If not, check if it is the goal state.
If not, expand the node and push new nodes into the queue.
}

- Depth-first search:
  The structure of DFS is very much like BFS. Just change the FIFO queue I used in BFS to a **LIFO stack**. Every time we expand the top node in the stack, which is the deepest node in the search tree. I also choose the directions the agent go randomly every time.
- Iterative deepening search:
  Iterative deepening search is a little bit like DFS, but it limits the depth of DFS every round. So I use DFS as a sub-function to implement iterative deepening search. I pass a parameter called "level" to DFS, and DFS(level) is limited to that depth. Its high level structure is like this:

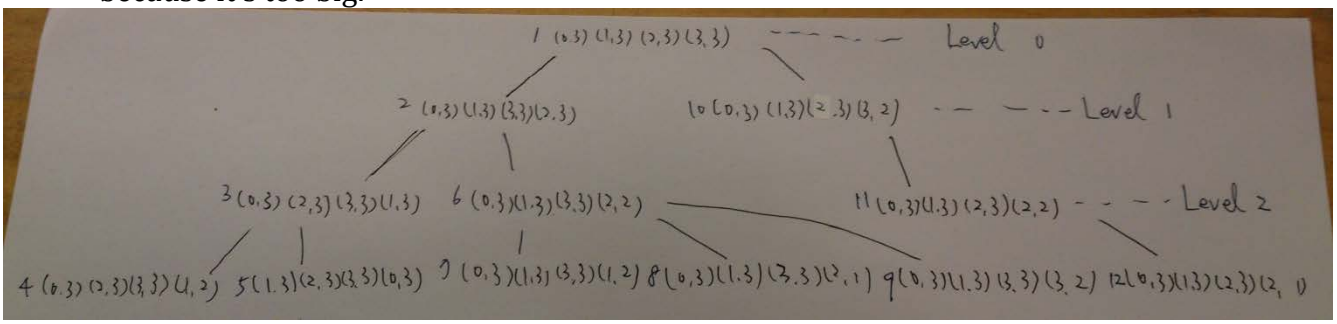
```
level = 1;
while(# of patterns traversed this round != # of all possible patterns)
    {
    if ( DFS(level)==true )
        return true;
    ++level;
    }
return false;
```

- A* search:
  The structure of A* search is very much like BFS. Just change the FIFO queue I used in BFS to **priority queue**. This priority queue uses some heuristic function to determine the priority. I use (# of moves from the initial state + the Manhattan distance to goal state) as my estimated cost function.
- Difficulties scaling
  I use two variables to control the width of grid and the number of named tiles. So it's easy to change these 2 parameters without modifying the program a lot. And the start and goal patterns are written in the class constructor function. So it's also easy to modify patterns. To add some obstacles in the grid, just use the used-pattern array to limit the agent. For example, if the agent is not allowed to go to (3, 0), then I will set used_pattern[a][b][c][3] = TRUE. So the program will never search the patterns with agent in (3, 0).

- Evidence that each searching algorithm works
  - Verification
    The node object remembered the sequence of moves it has done. When the goal is reached, I back trace from the goal pattern according to the sequence of moves. If we could reach the start pattern, then this solution is valid. See the right figure, the lowest node represents the start pattern if the solution is valid.



- Breadth-first search
  I limit the search depth to 3, and output the node's **coordinate** and its **level** in the search tree when expanding a node. As you can see from the output messages in the appendix section, the lower level's nodes are always traversed before the higher level's nodes.
- Depth-first search
  I limit the search depth to 3. Besides the nodes' **coordinates** and their **levels**, this time I also output a message when **back tracing from a wrong search branch**. You can see the output messages in the appendix section. From these messages, I can construct a search tree and label the traversing order on every node. You can see from the picture below that the nodes are traversed in a depth-first way. I only draw half part of the search tree because it's too big.



- Iterative deepening search
  Besides all messages I output in depth-first search, I specify the **start of iteration**. If you take a look at the picture, you can see that the nodes are also traversed in a depth-first order, despite of different depth limits.
- A* search
  I limit the search to 13 nodes. As you can see from the picture in the appendix, the node with the **smallest estimated cost** at the time will be traverse first.

- Comparison of search methods in simple case
  - Start state

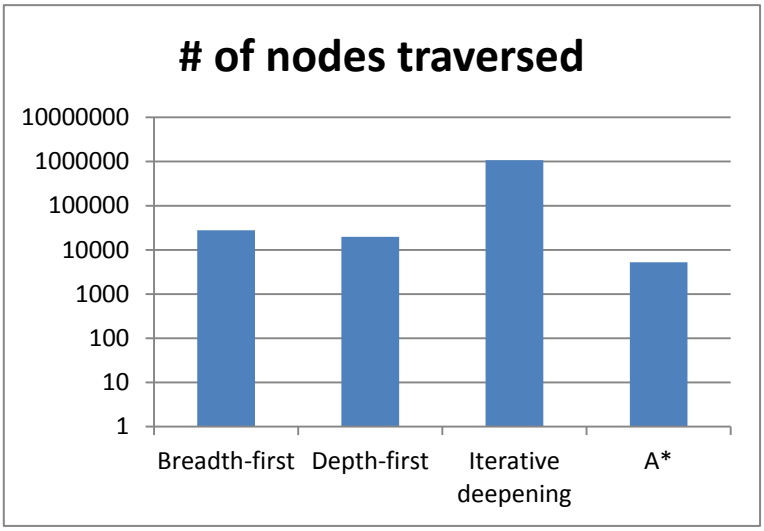
A	B	C	agent

  goal state

	Agent		
	C		
	B		
	A		

  - Total Manhattan distance = 10
  - Results

	# of traversed nodes
Breadth-first	27854
Depth-first	19674
Iterative deepening	1075790
A*	5255

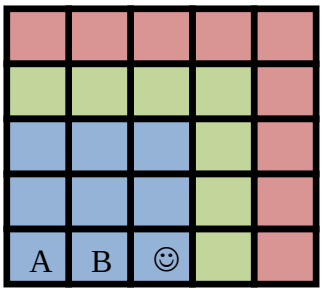


  - Because I **randomly** move the agent when implementing DFS, so there are some variations in DFS and iterative deepening.

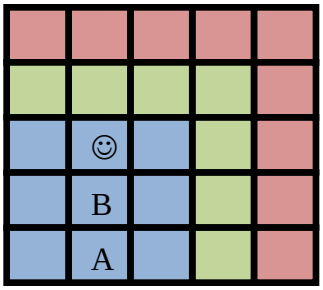
The range of DFS is about 1000~50000. The range of iterative deepening is about 850000~1200000

- Scaling of difficulties in different ways
  - Different size of grid

Start state

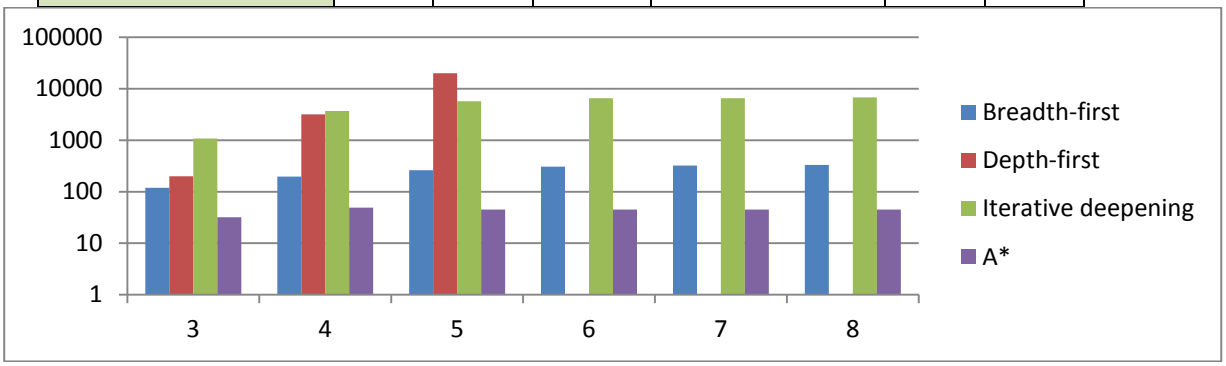


Goal state



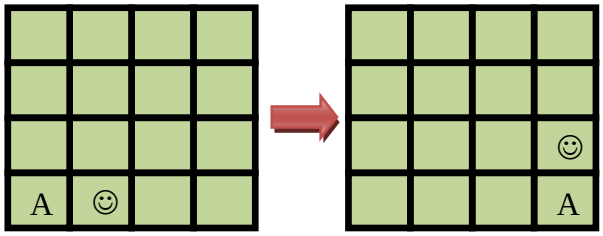
    - fixed number of named tiles = 3; fixed total Manhattan distance = 5;
    - Results (# of traversed nodes)

Width of the grid	3	4	5	6	7	8
Breadth-first	119	196	262	306	323	331
Depth-first	199	3182	20069	Out of memory		
Iterative deepening	1083	3701	5707	6538	6751	6782
A*	32	49	45	45	45	45

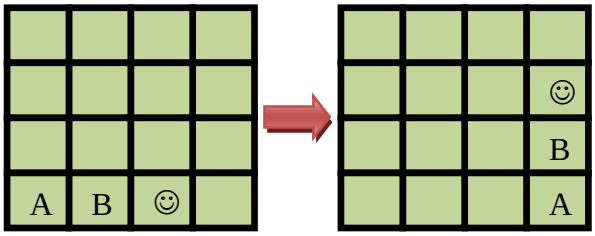


  - Different number of named tiles

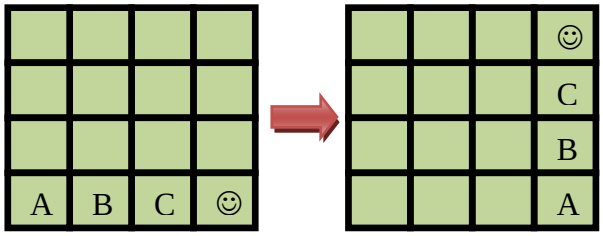
Pattern 1



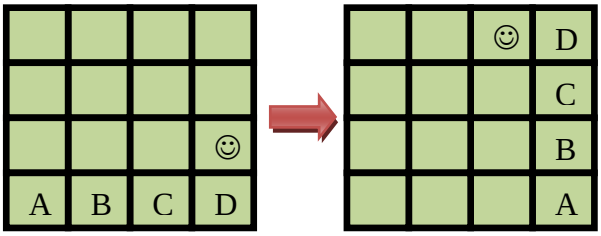
Pattern 2



Pattern 3



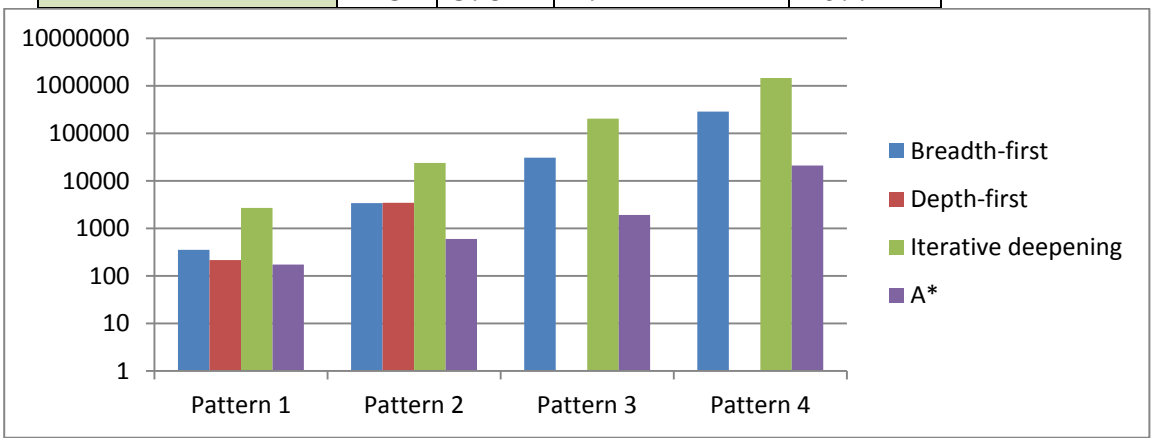
Pattern 4



    - fixed width of grid = 4;

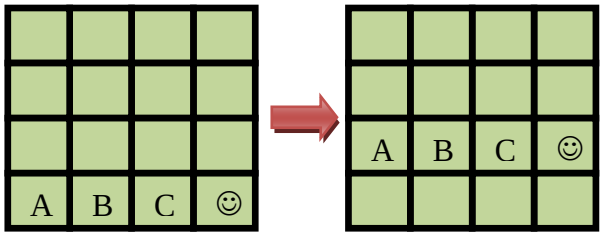
- Results (# of traversed nodes)

Pattern	1	2	3	4
Breadth-first	354	3411	30792	286906
Depth-first	216	3460	Out of memory	
Iterative Deepening	2691	23830	204020	1462333
A*	173	598	1917	20991

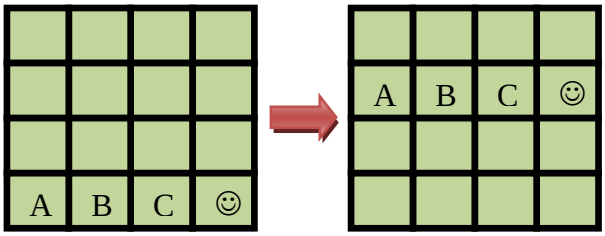


◦ Different goal patterns

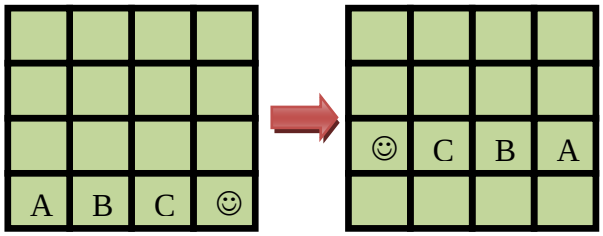
Pattern 1 (Manhattan distance sum = 4)



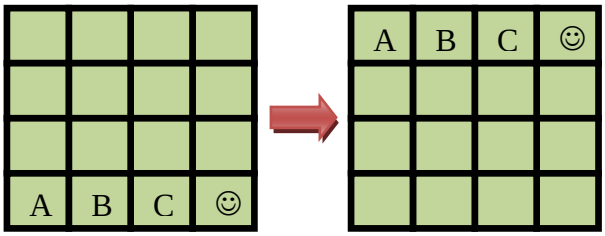
Pattern 2 (Manhattan distance sum = 8)



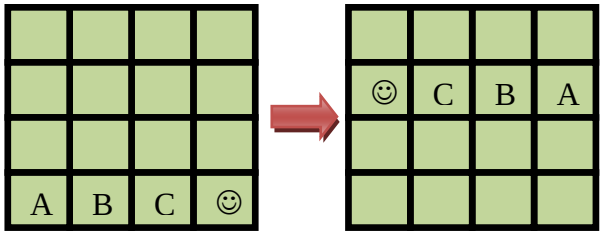
Pattern 3 (Manhattan distance sum = 8)



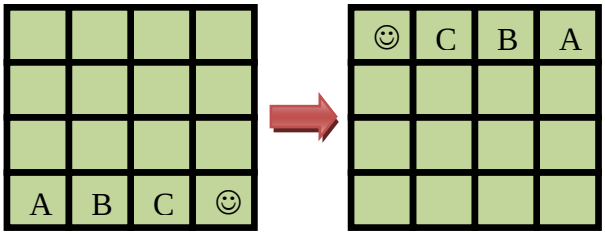
Pattern 4 (Manhattan distance sum = 12)



Pattern 5 (Manhattan distance sum = 16)



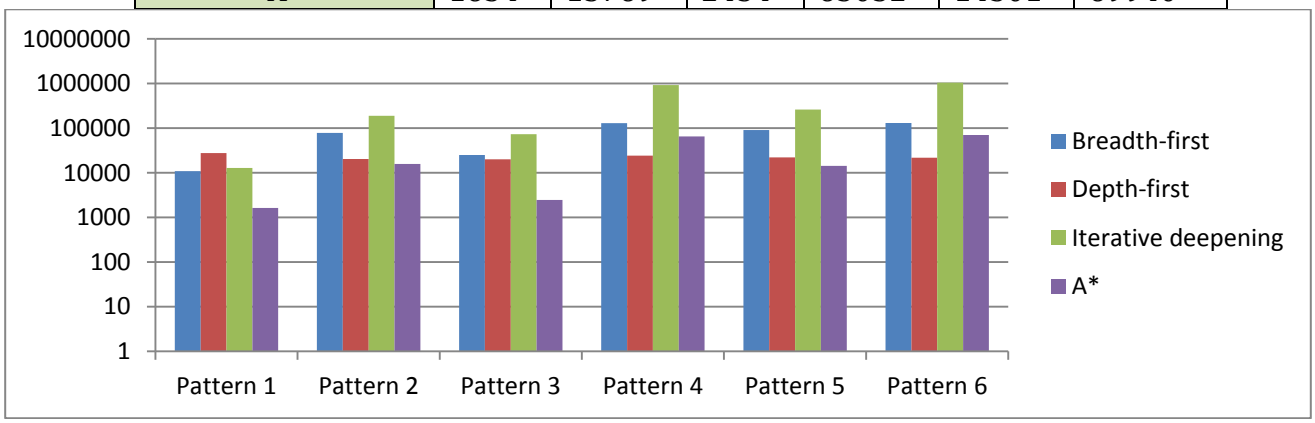
Pattern 6 (Manhattan distance sum = 20)



- Fixed width of grid = 4; fixed number of tiles = 4;
- Results (# of traversed nodes)

Pattern	1	2	3	4	5	6
Breadth-first	10891	78133	25105	129207	90805	129993
Depth-first	27686	20279	20013	24231	22052	21714

Iterative Deepening	12784	188605	73271	919935	261071	1045623
A*	1634	15769	2454	65052	14301	69940



- Put some obstacles in the grid

Start pattern

	1	2	
	3	4	
A	B	C	☺

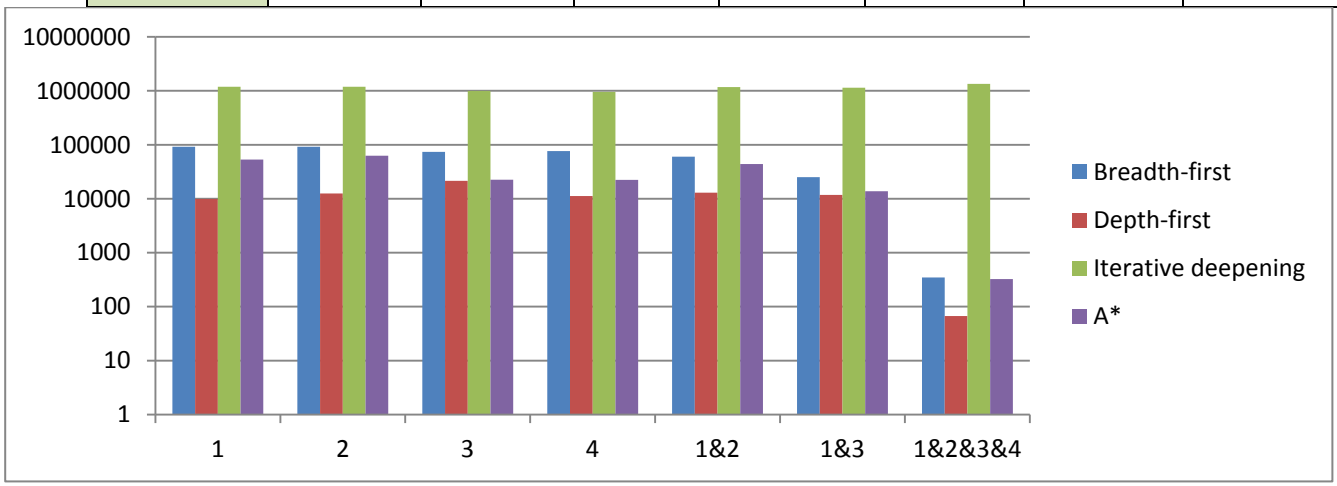
Goal pattern

☺	C	B	A
	1	2	
	3	4	

After experiment, there are only **4 slots** which are possible to add obstacles to. If I add an obstacle to any of other slots, there will be **no solution**. It is a reasonable result. For example, if there is an obstacle above A in the start pattern, then there is no way for the agent to move A. Because if the agent swaps with A, **the only move the agent can take next is swapping with A again**. This will make A go back to the initial pattern. So it's impossible to move A to the goal.

- Fixed width of grid =4; fixed start and goal pattern
- Results

Position of obstacles	1	2	3	4	1&2	1&3	1&2&3&4
Breadth-first	91614	91537	74068	76588	59958	25166	347
Depth-first	9890	12488	21418	11235	12906	11763	67
Iterative deepening	1184530	1188491	998247	953614	1169432	1143398	1340149
A*	53352	62685	22534	22417	43894	13731	324



- Discussion and conclusions
  - Comparison between 4 methods:
    - From those tests results, we can see that basically, BFS and A* are the most efficient and stable algorithms.
    - Since choosing directions matters a lot to DFS, it is a very unstable algorithm. When we are lucky, DFS can find a solution in short time. But sometimes it goes deep into the search tree and ends up running out of memory.
    - Iterative deepening is a stable algorithm. However, it is always slower than BFS and A* by several orders. Because it repeats traversing same nodes for many times.
  - The case of **changing size of grid:** Because A* always chooses the most promising nodes, it wasn't affected by the change of grid size. It still headed to the right direction no matter how big the grid become. For BFS and iterative deepening, the change of grid size also doesn't affect them much. As long as the solution is the same, BFS and iterative deepening won't spend too much time searching in the wrong direction.
  - The case of **different number of named tiles:** This case's complexity grows dramatically with the increasing of number of named tiles. Every method needs more and more time to solve the problem. In such complex problem, the superiority of A* method becomes very apparent.
  - The case of **different goal patterns:** We can see from the case that the difficulty of such problem is not strongly depending on total Manhattan distance. Pattern 5 has bigger Manhattan distance than pattern 4 between start and goal pattern. But obviously, pattern 4 requires more time to solve. So I guess that the difficulty of such problem might strongly depend on the placement of every tile, not only the distance between start and goal.
  - The case of **adding obstacles:** We can see from the results that "the more the obstacles are, the easier the problem will be." This is a reasonable result, because obstacles can limit the selections of moves. The search tree thus becomes smaller. However, some slots in the grid couldn't be placed obstacles, or there will be no solution. The reason is described in the previous passage.
  - Weakness of my work:
    - My heuristic function for A* is actually not totally admissible. There is one case it will fail. When agent and a named tile are beside each other and the goal pattern is they swapping, the total Manhattan distance (the estimated lowest cost to goal) will be 2. However, the actual cost is only 1, since only 1 swap is needed. This is only a small error, so it still gives right solutions for most of the cases.
    - My method to remember all traversed nodes is not very good. Because it needs an array of size = (# of positions in the grid)^(# of named tiles), this way is not scalable. For example, if the grid size is 4*4, and we have 5 named tiles, then we need an array of size = 16*16*16*16*16 = 1048576. Perhaps I should use map container in C++ instead. It will need O(log n) time to look up, but it won't waste memory.
    - My implementation of DFS is not very good. Because every node remembers the whole move sequence, it will thus need a lot of memory when DFS goes deep into the search tree. That's why in some complex cases DFS went out of memory. Perhaps using recursion to implement DFS could alleviate the problem, since the recursion version of DFS won't store so many nodes. It only keeps track of one branch a time.

- Appendix (all of my code is in the end)
  - Debugging (verification) messages mentioned before

BFS

```
----Level 1: (0, 3)  (1, 3)  (3, 3)  (2, 3)
----Level 1: (0, 3)  (1, 3)  (2, 3)  (3, 2)
----Level 2: (0, 3)  (2, 3)  (3, 3)  (1, 3)
----Level 2: (0, 3)  (1, 3)  (3, 3)  (2, 2)
----Level 2: (0, 3)  (1, 3)  (2, 3)  (2, 2)
----Level 2: (0, 3)  (1, 3)  (2, 3)  (3, 1)
----Level 3: (1, 3)  (2, 3)  (3, 3)  (0, 3)
----Level 3: (0, 3)  (2, 3)  (3, 3)  (1, 2)
----Level 3: (0, 3)  (1, 3)  (3, 3)  (1, 2)
----Level 3: (0, 3)  (1, 3)  (3, 3)  (3, 2)
----Level 3: (0, 3)  (1, 3)  (3, 3)  (2, 1)
----Level 3: (0, 3)  (1, 3)  (2, 3)  (1, 2)
----Level 3: (0, 3)  (1, 3)  (2, 3)  (2, 1)
----Level 3: (0, 3)  (1, 3)  (2, 2)  (2, 3)
----Level 3: (0, 3)  (1, 3)  (2, 3)  (3, 0)
No Solution
Traverse through: 22 nodes

Process returned 0 (0x0)   execution time : 0.036 s
Press any key to continue.
```

DFS

```
Level 0: (0,3)  (1,3)  (2,3)  (3,3)
Level 1: (0,3)  (1,3)  (3,3)  (2,3)
Level 2: (0,3)  (2,3)  (3,3)  (1,3)
Level 3: (0,3)  (2,3)  (3,3)  (1,2)
Back trace from level 3
Level 3: (1,3)  (2,3)  (3,3)  (0,3)
Back trace from level 3
Back trace from level 2
Level 2: (0,3)  (1,3)  (3,3)  (2,2)
Level 3: (0,3)  (1,3)  (3,3)  (1,2)
Back trace from level 3
Level 3: (0,3)  (1,3)  (3,3)  (2,1)
Back trace from level 3
Level 3: (0,3)  (1,3)  (3,3)  (3,2)
Back trace from level 3
Back trace from level 2
Back trace from level 1
Level 1: (0,3)  (1,3)  (2,3)  (3,2)
Level 2: (0,3)  (1,3)  (2,3)  (2,2)
Level 3: (0,3)  (1,3)  (2,3)  (2,1)
Back trace from level 3
Level 3: (0,3)  (1,3)  (2,2)  (2,3)
Back trace from level 3
Level 3: (0,3)  (1,3)  (2,3)  (1,2)
Back trace from level 3
Back trace from level 2
Level 2: (0,3)  (1,3)  (2,3)  (3,1)
Level 3: (0,3)  (1,3)  (2,3)  (2,1)
Back trace from level 3
Level 3: (0,3)  (1,3)  (2,3)  (3,0)
Back trace from level 3
Back trace from level 2
Back trace from level 1
Back trace from level 0
No Solution
Traverse through: 17 nodes
```

Iterative deepening

```
<<<<<<Round start! Maximum depth = 0
Level 2: (0,3)  (1,3)  (2,3)  (3,3)
Back trace from level 2
<<<<<<Round start! Maximum depth = 1
Level 1: (0,3)  (1,3)  (2,3)  (3,3)
Level 2: (0,3)  (1,3)  (3,3)  (2,3)
Back trace from level 2
Level 2: (0,3)  (1,3)  (2,3)  (3,2)
Back trace from level 2
Back trace from level 1
<<<<<<Round start! Maximum depth = 2
Level 0: (0,3)  (1,3)  (2,3)  (3,3)
Level 1: (0,3)  (1,3)  (3,3)  (2,3)
Level 2: (0,3)  (2,3)  (3,3)  (1,3)
Back trace from level 2
Level 2: (0,3)  (1,3)  (3,3)  (2,2)
Back trace from level 2
Back trace from level 1
Level 1: (0,3)  (1,3)  (2,3)  (3,2)
Level 2: (0,3)  (1,3)  (2,3)  (2,2)
Back trace from level 2
Level 2: (0,3)  (1,3)  (2,3)  (3,1)
Back trace from level 2
Back trace from level 1
Back trace from level 0
No Solution
Traverse through: 11 nodes
```

A*

```
(0, 3)   (1, 3)   (2, 3)   (3, 3)
created time: 0, traversed time: 1, estimated cost = 10
(0, 3)   (1, 3)   (3, 3)   (2, 3)
created time: 2, traversed time: 3, estimated cost = 10
(0, 3)   (1, 3)   (2, 3)   (3, 2)
created time: 2, traversed time: 5, estimated cost = 11
(0, 3)   (1, 3)   (2, 3)   (3, 3)
created time: 4, traversed time: 7, estimated cost = 11
(0, 3)   (1, 3)   (2, 3)   (3, 3)
created time: 6, traversed time: 9, estimated cost = 11
(0, 3)   (2, 3)   (3, 3)   (1, 3)
created time: 4, traversed time: 11, estimated cost = 11
(0, 3)   (1, 3)   (2, 3)   (2, 2)
created time: 6, traversed time: 13, estimated cost = 11
(0, 3)   (1, 3)   (2, 2)   (2, 3)
created time: 14, traversed time: 15, estimated cost = 10
(0, 3)   (1, 3)   (2, 3)   (1, 2)
created time: 14, traversed time: 17, estimated cost = 11
(0, 3)   (1, 2)   (2, 3)   (1, 3)
created time: 18, traversed time: 19, estimated cost = 10
(0, 3)   (2, 3)   (2, 2)   (1, 3)
created time: 16, traversed time: 21, estimated cost = 11
(1, 3)   (1, 2)   (2, 3)   (0, 3)
created time: 20, traversed time: 23, estimated cost = 11
(0, 3)   (1, 2)   (1, 3)   (2, 3)
created time: 20, traversed time: 25, estimated cost = 11
No Solution
Traverse through: 13 nodes

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

- My source code (in C++)

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <vector>
#include <queue>
#include <time.h>
#include <iomanip>
#include <math.h>
#include <algorithm>
#include <stack>

/** index 0 start from top left corner **/
/** 0123 **/
/** 4567 **/
/** 89ab **/
/** cdef **/

using namespace std;

class axis
   {
   public:
      int x, y;
   };

class bfs_obj
   {
   public:
      bfs_obj();
      int created_time;
      int m_distance;
      int level;
      int next_direction;
      vector<int> direction_list;
      vector<axis> recent_position;
   };

bfs_obj::bfs_obj()
   {
   m_distance = 0;
   }

class tile_moving
   {
   public:
      tile_moving();
```

```
    bool depth_first_typical();
    bool depth_first_stack_for_iterative_deepening(int level);
    bool breadth_first_new();
    bool iterative_deepening();
    bool a_star();
    void output_backtrace_result();
    void how_many_nodes();


  private:
    int debug;
    int num_of_named_tile;
    int width_of_grid;
    vector<axis> tiles_position;
    vector<axis> goal_position;
    vector<vector<axis> > traversed_nodes;
    bool used_pattern[16][16][16][16];
    //bool used_pattern[9][9][9][9];/** type the number by hand **/

    void move_agent_block(vector<axis> &position_arr, int direction);
    bool check_goal(vector<axis> now);
    void recover_tiles_position(vector<axis> &record);
    int manhattan_distance_calculator(bfs_obj* a);

    /** for debugging purpose **/
    void output_tiles_position();
  };

tile_moving::tile_moving()
  {
  debug = 0;
  int i;
  int a, b, c, d, e;
  axis temp_axis;
  /** type following arguments by hand **/
  num_of_named_tile = 4;
  width_of_grid = 4;

  /** construct initial position **/
  i = 0;
  while(i < num_of_named_tile)
    {
    temp_axis.x = i;
    temp_axis.y = width_of_grid - 1;
    tiles_position.push_back(temp_axis);
    ++i;
    }
```

```
/** construct goal position **/
i = 0;
while(i < num_of_named_tile)
   {
   temp_axis.x = 1;
   temp_axis.y = width_of_grid - i - 1;
   //temp_axis.x = i;
   //temp_axis.y = width_of_grid - 1;
   goal_position.push_back(temp_axis);
   ++i;
   }
//goal_position[0].y = 2;
/** enter start and goal **/
tiles_position[0].x = 0;
tiles_position[0].y = 3;
tiles_position[1].x = 1;
tiles_position[1].y = 3;
tiles_position[2].x = 2;
tiles_position[2].y = 3;
tiles_position[3].x = 3;
tiles_position[3].y = 3;
//tiles_position[4].x = 2;
//tiles_position[4].y = 3;

goal_position[0].x = 3;
goal_position[0].y = 0;
goal_position[1].x = 2;
goal_position[1].y = 0;
goal_position[2].x = 1;
goal_position[2].y = 0;
goal_position[3].x = 0;
goal_position[3].y = 0;
//goal_position[4].x = 2;
//goal_position[4].y = 3;
/** used pattern record initialization **/
a = width_of_grid * width_of_grid - 1;
while(a >= 0)
   {
   b = width_of_grid * width_of_grid - 1;
   while(b >= 0)
      {
      //used_pattern[a][b] = false;
      c = width_of_grid * width_of_grid - 1;
      while(c >= 0)
         {
         //used_pattern[a][b][c] = false;
         d = width_of_grid * width_of_grid - 1;
         while(d >= 0)
```

```
            {
            /*e = width_of_grid * width_of_grid - 1;
            while( e >= 0)
              {
              used_pattern[a][b][c][d][e] = false;
              --e;
              }*/
            if(d==5 || d==6 || d==9 || d==10)
              used_pattern[a][b][c][d] = true;
            else
              used_pattern[a][b][c][d] = false;
            --d;
            }
          --c;
          }
        --b;
        }
      --a;
      }
   a = tiles_position[0].x + tiles_position[0].y * width_of_grid;
   b = tiles_position[1].x + tiles_position[1].y * width_of_grid;
   c = tiles_position[2].x + tiles_position[2].y * width_of_grid;
   d = tiles_position[3].x + tiles_position[3].y * width_of_grid;
   //e = tiles_position[4].x + tiles_position[4].y * width_of_grid;
   used_pattern[a][b][c][d] = true;
   }

void tile_moving::how_many_nodes()
   {
   cout << "Traverse through: " << debug << " nodes" << endl;
   return;
   }

void tile_moving::output_tiles_position()
   {
   cout << "A(" << tiles_position[0].x << ", " << tiles_position[0].y << "), "
       << "B(" << tiles_position[1].x << ", " << tiles_position[1].y << "), "
       << "C(" << tiles_position[2].x << ", " << tiles_position[2].y << "), "
       << "agent(" << tiles_position[3].x << ", " << tiles_position[3].y << "), " << endl;
   }

void tile_moving::output_backtrace_result()
   {
   cout << "A(" << goal_position[0].x << ", " << goal_position[0].y << "), "
       << "B(" << goal_position[1].x << ", " << goal_position[1].y << "), "
       << "C(" << goal_position[2].x << ", " << goal_position[2].y << "), "
       << "agent(" << goal_position[3].x << ", " << goal_position[3].y << "), " << endl;
   }
```

```
struct OrderByDistance
   {
   bool operator()(bfs_obj* a, bfs_obj* b) {return a->m_distance > b->m_distance;}
   };

bool tile_moving::a_star()
   {
   priority_queue<bfs_obj*, vector<bfs_obj*>, OrderByDistance> processing_pq;
   //vector<bfs_obj*> sorted_vector;
   int time_stamp = 0;
   int man_dist;
   int i, temp;
   int a, b, c, d, e;
   axis temp_axis;
   bfs_obj* this_move = NULL;
   bfs_obj* next_move = NULL;

   /** push the start node into PQ **/
   next_move = new bfs_obj;
   i = 0;
   while(i<num_of_named_tile)
      {
      temp_axis.x = tiles_position[i].x;
      temp_axis.y = tiles_position[i].y;
      next_move->recent_position.push_back(temp_axis);
      ++i;
      }
   //next_move->created_time = time_stamp;
   next_move->m_distance = 0 + manhattan_distance_calculator(next_move);
   //sorted_vector.push_back(next_move);
   processing_pq.push(next_move);

   a = next_move->recent_position[0].x + next_move->recent_position[0].y * width_of_grid;
   b = next_move->recent_position[1].x + next_move->recent_position[1].y * width_of_grid;
   c = next_move->recent_position[2].x + next_move->recent_position[2].y * width_of_grid;
   d = next_move->recent_position[3].x + next_move->recent_position[3].y * width_of_grid;
   //e = next_move->recent_position[4].x + next_move->recent_position[4].y * width_of_grid;
   used_pattern[a][b][c][d] = false;

   //++time_stamp;
   //while(debug!=13)
   while(processing_pq.empty()!=true)
      {
      ++debug;
      this_move = processing_pq.top();
      processing_pq.pop();
```

```
      /*cout << "(" << this_move->recent_position[0].x << ", " << this_move->recent_position[0].y
<< ")   ";
      cout << "(" << this_move->recent_position[1].x << ", " << this_move->recent_position[1].y <<
")   ";
      cout << "(" << this_move->recent_position[2].x << ", " << this_move->recent_position[2].y <<
")   ";
      cout << "(" << this_move->recent_position[3].x << ", " << this_move->recent_position[3].y <<
")" << endl;
      cout << "created time: " << this_move->created_time
           << ", traversed time: " << time_stamp
           <<", estimated cost = " << this_move->m_distance << endl;*/
      //++time_stamp;
      /** check if this pattern have already appeared **/
      a = this_move->recent_position[0].x + this_move->recent_position[0].y * width_of_grid;
      b = this_move->recent_position[1].x + this_move->recent_position[1].y * width_of_grid;
      c = this_move->recent_position[2].x + this_move->recent_position[2].y * width_of_grid;
      d = this_move->recent_position[3].x + this_move->recent_position[3].y * width_of_grid;
      //e = this_move->recent_position[4].x + this_move->recent_position[4].y * width_of_grid;

      if(used_pattern[a][b][c][d] != true)
        {
        /*cout << "(" << this_move->recent_position[0].x << ", " << this_move->recent_position[0].y
<< ")   ";
        cout << "(" << this_move->recent_position[1].x << ", " << this_move->recent_position[1].y
<< ")   ";
        cout << "(" << this_move->recent_position[2].x << ", " << this_move->recent_position[2].y
<< ")   ";
        cout << "(" << this_move->recent_position[3].x << ", " << this_move->recent_position[3].y
<< ")";
        cout << ", distance = " << this_move->m_distance << endl;*/
        /** the pattern(after move) is new **/
        used_pattern[a][b][c][d] = true;

        /** check if we reach the goal **/
        i = num_of_named_tile - 1;
        while(i>=0)
          {
          //cout << "start " << goal_position[i].x << ", end " << endl;
          //cout << "Recent position: " << this_move->recent_position[i].x << ", " << this_move-
>recent_position[i].y;
          //cout << ", goal is: " << goal_position[i].x << ", " << goal_position[i].y << endl;
          if(this_move->recent_position[i].x!=goal_position[i].x || this_move-
>recent_position[i].y!=goal_position[i].y)
            break;
          else
            --i;
          }
        /*cout << this_move->recent_position[0].x << ", " << this_move->recent_position[0].y <<
```

```
endl;
        cout << this_move->recent_position[1].x << ", " << this_move->recent_position[1].y << endl;
        cout << this_move->recent_position[2].x << ", " << this_move->recent_position[2].y << endl;
        cout << this_move->recent_position[3].x << ", " << this_move->recent_position[3].y <<
endl;*/
        if(i==-1)/** we found the solution **/
           {
           i = this_move->direction_list.size() - 1;
           cout << "Goal: ";
           output_backtrace_result();
           while(i >= 0)
              {
              if(this_move->direction_list[i]==0)
                 cout << "Move Right" << endl;
              else if(this_move->direction_list[i]==1)
                 cout << "Move Down" << endl;
              else if(this_move->direction_list[i]==2)
                 cout << "Move Left" << endl;
              else if(this_move->direction_list[i]==3)
                 cout << "Move Up" << endl;
              else
                 cout << "Fucking ERROR" << endl;
              temp = (this_move->direction_list[i]+2) % 4;
              move_agent_block(goal_position, temp);/** move in reverse direction **/
              output_backtrace_result();
              --i;
              }
           return true;
           }

        if(this_move->recent_position[num_of_named_tile-1].x > 0)
           {
           //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< 0 << endl;
           /** can move left **/
           next_move = new bfs_obj;
           i = 0;
           //cout << "Move Left" << endl;
           while(i<num_of_named_tile)
              {
              temp_axis.x = this_move->recent_position[i].x;
              temp_axis.y = this_move->recent_position[i].y;
              next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
              ++i;
              }
           /** make the move **/
```

```
            move_agent_block(next_move->recent_position, 2);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(2);
            next_move->m_distance = this_move->direction_list.size() +
manhattan_distance_calculator(next_move);

            processing_pq.push(next_move);
            }
          if(this_move->recent_position[num_of_named_tile-1].x < (width_of_grid-1))
            {
            //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< (width_of_grid-1) << endl;
            /** can move right **/

            next_move = new bfs_obj;
            i = 0;
            //cout << "Move Right" << endl;
            while(i<num_of_named_tile)
               {
               temp_axis.x = this_move->recent_position[i].x;
               temp_axis.y = this_move->recent_position[i].y;
               next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
               ++i;
               }
            /** make the move **/
            move_agent_block(next_move->recent_position, 0);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(0);
            next_move->m_distance = this_move->direction_list.size() +
manhattan_distance_calculator(next_move);

            processing_pq.push(next_move);
            }
          if(this_move->recent_position[num_of_named_tile-1].y > 0)
            {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< 0 << endl;
            /** can move up **/

            next_move = new bfs_obj;
            i = 0;
            //cout << "Move Up" << endl;
            while(i<num_of_named_tile)
               {
```

```
            temp_axis.x = this_move->recent_position[i].x;
            temp_axis.y = this_move->recent_position[i].y;
            next_move->recent_position.push_back(temp_axis);
            //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
            ++i;
            }
          /** make the move **/
          move_agent_block(next_move->recent_position, 3);

          next_move->direction_list = this_move->direction_list;
          next_move->direction_list.push_back(3);
          next_move->m_distance = this_move->direction_list.size() +
manhattan_distance_calculator(next_move);

          processing_pq.push(next_move);
          }
        if(this_move->recent_position[num_of_named_tile-1].y < (width_of_grid-1))
          {
          //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< (width_of_grid-1) << endl;
          /** can move down **/

          next_move = new bfs_obj;
          next_move->next_direction = 1;
          i = 0;
          //cout << "Move Down" << endl;
          while(i<num_of_named_tile)
            {
            temp_axis.x = this_move->recent_position[i].x;
            temp_axis.y = this_move->recent_position[i].y;
            next_move->recent_position.push_back(temp_axis);
            //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
            ++i;
            }
          /** make the move **/
          move_agent_block(next_move->recent_position, 1);

          next_move->direction_list = this_move->direction_list;
          next_move->direction_list.push_back(1);
          next_move->m_distance = this_move->direction_list.size() +
manhattan_distance_calculator(next_move);

          processing_pq.push(next_move);
          }
        }
      delete this_move;
```

```
    }
  return false;
  }

bool tile_moving::breadth_first_new()
  {
  queue<bfs_obj*> processing_Q;
  int time_stamp = 0;
  int man_dist;
  int i, temp;
  int a, b, c, d, e;
  axis temp_axis;
  bfs_obj* this_move = NULL;
  bfs_obj* next_move = NULL;

  /** push the start node into Queue **/
  next_move = new bfs_obj;
  i = 0;
  while(i<num_of_named_tile)
    {
    temp_axis.x = tiles_position[i].x;
    temp_axis.y = tiles_position[i].y;
    next_move->recent_position.push_back(temp_axis);
    ++i;
    }
  processing_Q.push(next_move);

  a = next_move->recent_position[0].x + next_move->recent_position[0].y * width_of_grid;
  b = next_move->recent_position[1].x + next_move->recent_position[1].y * width_of_grid;
  c = next_move->recent_position[2].x + next_move->recent_position[2].y * width_of_grid;
  d = next_move->recent_position[3].x + next_move->recent_position[3].y * width_of_grid;
  //e = next_move->recent_position[4].x + next_move->recent_position[4].y * width_of_grid;
  used_pattern[a][b][c][d] = false;

  //while(debug!=13)
  while(processing_Q.empty()!=true)
    {
    ++debug;
    this_move = processing_Q.front();
    processing_Q.pop();

    /** check if this pattern have already appeared **/
    a = this_move->recent_position[0].x + this_move->recent_position[0].y * width_of_grid;
    b = this_move->recent_position[1].x + this_move->recent_position[1].y * width_of_grid;
    c = this_move->recent_position[2].x + this_move->recent_position[2].y * width_of_grid;
    d = this_move->recent_position[3].x + this_move->recent_position[3].y * width_of_grid;
    // = this_move->recent_position[4].x + this_move->recent_position[4].y * width_of_grid;
```

```
    if(used_pattern[a][b][c][d] != true)
      {
      /** the pattern(after move) is new **/
      used_pattern[a][b][c][d] = true;

      /** check if we reach the goal **/
      i = num_of_named_tile - 1;
      while(i>=0)
        {
        if(this_move->recent_position[i].x!=goal_position[i].x || this_move-
>recent_position[i].y!=goal_position[i].y)
          break;
        else
          --i;
        }
      if(i==-1)/** we found the solution **/
        {
        /*i = this_move->direction_list.size() - 1;
        output_backtrace_result();
        while(i >= 0)
          {
          if(this_move->direction_list[i]==0)
            cout << "Move Right" << endl;
          else if(this_move->direction_list[i]==1)
            cout << "Move Down" << endl;
          else if(this_move->direction_list[i]==2)
            cout << "Move Left" << endl;
          else if(this_move->direction_list[i]==3)
            cout << "Move Up" << endl;
          else
            cout << "Fucking ERROR" << endl;
          temp = (this_move->direction_list[i]+2) % 4;
          move_agent_block(goal_position, temp);*/
          /** move in reverse direction **/
          //output_backtrace_result();
          //--i;
          //}
        return true;
        }

      if(this_move->recent_position[num_of_named_tile-1].x > 0)
        {
        //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< 0 << endl;
        /** can move left **/
        next_move = new bfs_obj;
        i = 0;
        //cout << "Move Left" << endl;
```

```
            while(i<num_of_named_tile)
               {
               temp_axis.x = this_move->recent_position[i].x;
               temp_axis.y = this_move->recent_position[i].y;
               next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
               ++i;
               }
            /** make the move **/
            move_agent_block(next_move->recent_position, 2);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(2);
            processing_Q.push(next_move);
            }
         if(this_move->recent_position[num_of_named_tile-1].x < (width_of_grid-1))
            {
            //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< (width_of_grid-1) << endl;
            /** can move right **/

            next_move = new bfs_obj;
            i = 0;
            //cout << "Move Right" << endl;
            while(i<num_of_named_tile)
               {
               temp_axis.x = this_move->recent_position[i].x;
               temp_axis.y = this_move->recent_position[i].y;
               next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
               ++i;
               }
            /** make the move **/
            move_agent_block(next_move->recent_position, 0);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(0);
            processing_Q.push(next_move);
            }
         if(this_move->recent_position[num_of_named_tile-1].y > 0)
            {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< 0 << endl;
            /** can move up **/

            next_move = new bfs_obj;
```

```
            i = 0;
            //cout << "Move Up" << endl;
            while(i<num_of_named_tile)
               {
               temp_axis.x = this_move->recent_position[i].x;
               temp_axis.y = this_move->recent_position[i].y;
               next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
               ++i;
               }
            /** make the move **/
            move_agent_block(next_move->recent_position, 3);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(3);
            processing_Q.push(next_move);
            }
         if(this_move->recent_position[num_of_named_tile-1].y < (width_of_grid-1))
            {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< (width_of_grid-1) << endl;
            /** can move down **/

            next_move = new bfs_obj;
            next_move->next_direction = 1;
            i = 0;
            //cout << "Move Down" << endl;
            while(i<num_of_named_tile)
               {
               temp_axis.x = this_move->recent_position[i].x;
               temp_axis.y = this_move->recent_position[i].y;
               next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
               ++i;
               }
            /** make the move **/
            move_agent_block(next_move->recent_position, 1);

            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(1);
            processing_Q.push(next_move);
            }
         }
      delete this_move;
      }
   return false;
```

```
    }

bool tile_moving::iterative_deepening()
    {
    int i, j;
    int a, b, c, d, w;
    int e, f, g, h, z;
    int total_pattern_number;
    int pattern_count;

    e = tiles_position[0].x + tiles_position[0].y * width_of_grid;
    f = tiles_position[1].x + tiles_position[1].y * width_of_grid;
    g = tiles_position[2].x + tiles_position[2].y * width_of_grid;
    h = tiles_position[3].x + tiles_position[3].y * width_of_grid;
    //z = tiles_position[4].x + tiles_position[4].y * width_of_grid;

    i = 1;
    total_pattern_number = width_of_grid * width_of_grid;
    total_pattern_number = total_pattern_number * total_pattern_number;
    total_pattern_number = total_pattern_number * total_pattern_number;
    pattern_count = 1;
    //while(i<=3)
    while(pattern_count!=total_pattern_number)
        {
        //cout << "<<<<<<Round start! Maximum depth = " << i-1 << endl;
        //cout << i << endl;
        //cout << pattern_count << endl;
        if(depth_first_stack_for_iterative_deepening(i))
            return true;
        pattern_count = 1;
        a = width_of_grid * width_of_grid - 1;
        while(a >= 0)
            {
            b = width_of_grid * width_of_grid - 1;
            while(b >= 0)
                {
                c = width_of_grid * width_of_grid - 1;
                while(c >= 0)
                    {
                    d = width_of_grid * width_of_grid - 1;
                    while(d >= 0)
                        {
                        if(used_pattern[a][b][c][d] ==true)
                            {
                            ++pattern_count;
                            used_pattern[a][b][c][d] = false;
                            }
                        --d;
```

```
                }
              --c;
              }
            --b;
            }
          --a;
          }
        /** initialization for the next round **/
        /*j = 0;
        while(j < num_of_named_tile)
          {
          tiles_position[j].x = j;
          tiles_position[j].y = width_of_grid - j - 1;
          ++j;
          }*/
        used_pattern[e][f][g][h] = true;
        tiles_position[0].x = 0;
        tiles_position[0].y = 3;
        tiles_position[1].x = 1;
        tiles_position[1].y = 3;
        tiles_position[2].x = 2;
        tiles_position[2].y = 3;
        tiles_position[3].x = 3;
        tiles_position[3].y = 3;

        ++i;
        //cout << "Depth: " << i << endl;
        //cout << "# of pattern: " << pattern_count << endl;
        }
    return false;
    }

bool tile_moving::depth_first_stack_for_iterative_deepening(int level)
    {
    stack<bfs_obj*> processing_stack;
    int i, j, temp;
    int a, b, c, d, e;
    axis temp_axis;
    bfs_obj* this_move = NULL;
    bfs_obj* next_move = NULL;

    /** push the start node into Queue **/
    next_move = new bfs_obj;
    i = 0;
    while(i<num_of_named_tile)
        {
        temp_axis.x = tiles_position[i].x;
        temp_axis.y = tiles_position[i].y;
```

```
    next_move->recent_position.push_back(temp_axis);
    ++i;
    }
  next_move->level = 0;
  processing_stack.push(next_move);

  a = next_move->recent_position[0].x + next_move->recent_position[0].y * width_of_grid;
  b = next_move->recent_position[1].x + next_move->recent_position[1].y * width_of_grid;
  c = next_move->recent_position[2].x + next_move->recent_position[2].y * width_of_grid;
  d = next_move->recent_position[3].x + next_move->recent_position[3].y * width_of_grid;
  //e = next_move->recent_position[4].x + next_move->recent_position[4].y * width_of_grid;
  used_pattern[a][b][c][d] = false;

  //while(debug!=13)
  while(processing_stack.empty()!=true)
    {
    this_move = processing_stack.top();
    processing_stack.pop();
    if(this_move->level != level)
      {
      ++debug;
      /** check if this pattern have already appeared **/
      a = this_move->recent_position[0].x + this_move->recent_position[0].y * width_of_grid;
      b = this_move->recent_position[1].x + this_move->recent_position[1].y * width_of_grid;
      c = this_move->recent_position[2].x + this_move->recent_position[2].y * width_of_grid;
      d = this_move->recent_position[3].x + this_move->recent_position[3].y * width_of_grid;
      //e = this_move->recent_position[4].x + this_move->recent_position[4].y * width_of_grid;
      if(used_pattern[a][b][c][d] !=true)
      {
      /** the pattern(after move) is new **/
      used_pattern[a][b][c][d] = true;
      /** check if we reach the goal **/
      i = num_of_named_tile - 1;
      while(i>=0)
        {
        if(this_move->recent_position[i].x!=goal_position[i].x || this_move-
>recent_position[i].y!=goal_position[i].y)
          break;
        else
          --i;
        }
      if(i==-1)/** we found the solution **/
        {
        /*i = this_move->direction_list.size() - 1;
        output_backtrace_result();
        while(i >= 0)
          {
          if(this_move->direction_list[i]==0)
```

```
                cout << "Move Right" << endl;
              else if(this_move->direction_list[i]==1)
                cout << "Move Down" << endl;
              else if(this_move->direction_list[i]==2)
                cout << "Move Left" << endl;
              else if(this_move->direction_list[i]==3)
                cout << "Move Up" << endl;
              else
                cout << "Fucking ERROR" << endl;
              temp = (this_move->direction_list[i]+2) % 4;
              move_agent_block(goal_position, temp);*//** move in reverse direction **/
              //output_backtrace_result();
              //--i;
              //}
            return true;
            }
          j = 4;
          temp = abs(rand());
          while(j>0)
            {
            temp = temp % 4;
            switch(temp)
            {
            case 0:
            if(this_move->recent_position[num_of_named_tile-1].x > 0)
              {
            //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< 0 << endl;
              /** can move left **/
              next_move = new bfs_obj;
              i = 0;
            //cout << "Move Left" << endl;
              while(i<num_of_named_tile)
                {
                temp_axis.x = this_move->recent_position[i].x;
                temp_axis.y = this_move->recent_position[i].y;
                next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
                ++i;
                }
              /** make the move **/
              move_agent_block(next_move->recent_position, 2);

              next_move->level = this_move->level + 1;
              next_move->direction_list = this_move->direction_list;
              next_move->direction_list.push_back(2);
              processing_stack.push(next_move);
```

```
            }
          break;

          case 1:
          if(this_move->recent_position[num_of_named_tile-1].x < (width_of_grid-1))
            {
          //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< (width_of_grid-1) << endl;
            /** can move right **/

            next_move = new bfs_obj;
            i = 0;
          //cout << "Move Right" << endl;
            while(i<num_of_named_tile)
              {
              temp_axis.x = this_move->recent_position[i].x;
              temp_axis.y = this_move->recent_position[i].y;
              next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
              ++i;
              }
            /** make the move **/
            move_agent_block(next_move->recent_position, 0);

            next_move->level = this_move->level + 1;
            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(0);
            processing_stack.push(next_move);
            }
          break;

          case 2:
          if(this_move->recent_position[num_of_named_tile-1].y > 0)
            {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge:
" << 0 << endl;
            /** can move up **/

            next_move = new bfs_obj;
            i = 0;
          //cout << "Move Up" << endl;
            while(i<num_of_named_tile)
              {
              temp_axis.x = this_move->recent_position[i].x;
              temp_axis.y = this_move->recent_position[i].y;
              next_move->recent_position.push_back(temp_axis);
            //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
```

```
>recent_position[i].y << ")  ";
                ++i;
                }
            /** make the move **/
            move_agent_block(next_move->recent_position, 3);

            next_move->level = this_move->level + 1;
            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(3);
            processing_stack.push(next_move);
            }
          break;

          case 3:
          if(this_move->recent_position[num_of_named_tile-1].y < (width_of_grid-1))
            {
          //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< (width_of_grid-1) << endl;
            /** can move down **/

            next_move = new bfs_obj;
            next_move->next_direction = 1;
            i = 0;
          //cout << "Move Down" << endl;
            while(i<num_of_named_tile)
              {
              temp_axis.x = this_move->recent_position[i].x;
              temp_axis.y = this_move->recent_position[i].y;
              next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
                ++i;
                }
            /** make the move **/
            move_agent_block(next_move->recent_position, 1);

            next_move->level = this_move->level + 1;
            next_move->direction_list = this_move->direction_list;
            next_move->direction_list.push_back(1);
            processing_stack.push(next_move);
            }
          break;

          default :
            cout << "Bug" << endl;
            break;
          }
          ++temp;
```

```
            --j;
            }
          }
          }
      delete this_move;
      }
   return false;
   }

bool tile_moving::depth_first_typical()
   {
   stack<bfs_obj*> processing_stack;
   int i, j, temp;
   int a, b, c, d, e;
   axis temp_axis;
   bfs_obj* this_move = NULL;
   bfs_obj* next_move = NULL;

   /** push the start node into Queue **/
   next_move = new bfs_obj;
   i = 0;
   while(i<num_of_named_tile)
      {
      temp_axis.x = tiles_position[i].x;
      temp_axis.y = tiles_position[i].y;
      next_move->recent_position.push_back(temp_axis);
      ++i;
      }
   next_move->level = 0;
   processing_stack.push(next_move);

   a = next_move->recent_position[0].x + next_move->recent_position[0].y * width_of_grid;
   b = next_move->recent_position[1].x + next_move->recent_position[1].y * width_of_grid;
   c = next_move->recent_position[2].x + next_move->recent_position[2].y * width_of_grid;
   d = next_move->recent_position[3].x + next_move->recent_position[3].y * width_of_grid;
   //e = next_move->recent_position[4].x + next_move->recent_position[4].y * width_of_grid;
   used_pattern[a][b][c][d] = true;

   //while(debug!=13)
   while(processing_stack.empty()!=true)
      {
      this_move = processing_stack.top();
      processing_stack.pop();
         ++debug;
         //cout << debug << endl;
         /** check if this pattern have already appeared **/
         //a = this_move->recent_position[0].x + this_move->recent_position[0].y * width_of_grid;
         //b = this_move->recent_position[1].x + this_move->recent_position[1].y * width_of_grid;
```

```
        //c = this_move->recent_position[2].x + this_move->recent_position[2].y * width_of_grid;
        //d = this_move->recent_position[3].x + this_move->recent_position[3].y * width_of_grid;
        //e = this_move->recent_position[4].x + this_move->recent_position[4].y * width_of_grid;
        //if(used_pattern[a][b][c][d] == false)
           //{
           /** the pattern(after move) is new **/
           //used_pattern[a][b][c][d] = true;

           /** check if we reach the goal **/
           i = num_of_named_tile - 1;
           while(i>=0)
              {
              if(this_move->recent_position[i].x!=goal_position[i].x || this_move-
>recent_position[i].y!=goal_position[i].y)
                 break;
              else
                 --i;
              }
           if(i==-1)/** we found the solution **/
              {
              /*i = this_move->direction_list.size() - 1;
              output_backtrace_result();
              while(i >= 0)
                 {
                 if(this_move->direction_list[i]==0)
                    cout << "Move Right" << endl;
                 else if(this_move->direction_list[i]==1)
                    cout << "Move Down" << endl;
                 else if(this_move->direction_list[i]==2)
                    cout << "Move Left" << endl;
                 else if(this_move->direction_list[i]==3)
                    cout << "Move Up" << endl;
                 else
                    cout << "Fucking ERROR" << endl;
                 temp = (this_move->direction_list[i]+2) % 4;*/
                 //move_agent_block(goal_position, temp);/** move in reverse direction **/
                 //output_backtrace_result();
                 //--i;
                 //}
              return true;
              }
           j = 4;
           temp = abs(rand());
        while(j>0)
           {
           temp = temp % 4;
           switch(temp)
           {
```

```
            case 0:
            if(this_move->recent_position[num_of_named_tile-1].x > 0)
               {
            //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< 0 << endl;
               /** can move left **/
               next_move = new bfs_obj;
               i = 0;
            //cout << "Move Left" << endl;
               while(i<num_of_named_tile)
                  {
                  temp_axis.x = this_move->recent_position[i].x;
                  temp_axis.y = this_move->recent_position[i].y;
                  next_move->recent_position.push_back(temp_axis);
               //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
                  ++i;
                  }
               /** make the move **/
               move_agent_block(next_move->recent_position, 2);
               a = next_move->recent_position[0].x + next_move->recent_position[0].y *
width_of_grid;
               b = next_move->recent_position[1].x + next_move->recent_position[1].y *
width_of_grid;
               c = next_move->recent_position[2].x + next_move->recent_position[2].y *
width_of_grid;
               d = next_move->recent_position[3].x + next_move->recent_position[3].y *
width_of_grid;
               if(used_pattern[a][b][c][d] != true)
                  {
                  used_pattern[a][b][c][d] = true;
                  next_move->level = this_move->level + 1;
                  next_move->direction_list = this_move->direction_list;
                  next_move->direction_list.push_back(2);
                  processing_stack.push(next_move);
                  }
               else
                  delete next_move;
               }
            break;

            case 1:
            if(this_move->recent_position[num_of_named_tile-1].x < (width_of_grid-1))
               {
            //cout << "Agent x: " << this_move->recent_position[num_of_named_tile-1].x << ", edge: "
<< (width_of_grid-1) << endl;
               /** can move right **/
```

```
            next_move = new bfs_obj;
            i = 0;
          //cout << "Move Right" << endl;
            while(i<num_of_named_tile)
              {
              temp_axis.x = this_move->recent_position[i].x;
              temp_axis.y = this_move->recent_position[i].y;
              next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
              ++i;
              }
            /** make the move **/
            move_agent_block(next_move->recent_position, 0);

            a = next_move->recent_position[0].x + next_move->recent_position[0].y *
width_of_grid;
            b = next_move->recent_position[1].x + next_move->recent_position[1].y *
width_of_grid;
            c = next_move->recent_position[2].x + next_move->recent_position[2].y *
width_of_grid;
            d = next_move->recent_position[3].x + next_move->recent_position[3].y *
width_of_grid;
            if(used_pattern[a][b][c][d] != true)
              {
              used_pattern[a][b][c][d] = true;
              next_move->level = this_move->level + 1;
              next_move->direction_list = this_move->direction_list;
              next_move->direction_list.push_back(0);
              processing_stack.push(next_move);
              }
            else
              delete next_move;
            }
          break;

          case 2:
          if(this_move->recent_position[num_of_named_tile-1].y > 0)
            {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge:
" << 0 << endl;
            /** can move up **/

            next_move = new bfs_obj;
            i = 0;
          //cout << "Move Up" << endl;
            while(i<num_of_named_tile)
              {
```

```
                temp_axis.x = this_move->recent_position[i].x;
                temp_axis.y = this_move->recent_position[i].y;
                next_move->recent_position.push_back(temp_axis);
              //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
>recent_position[i].y << ")  ";
                ++i;
                }
              /** make the move **/
              move_agent_block(next_move->recent_position, 3);

              a = next_move->recent_position[0].x + next_move->recent_position[0].y *
width_of_grid;
              b = next_move->recent_position[1].x + next_move->recent_position[1].y *
width_of_grid;
              c = next_move->recent_position[2].x + next_move->recent_position[2].y *
width_of_grid;
              d = next_move->recent_position[3].x + next_move->recent_position[3].y *
width_of_grid;
              if(used_pattern[a][b][c][d] != true)
                {
                used_pattern[a][b][c][d] = true;
                next_move->level = this_move->level + 1;
                next_move->direction_list = this_move->direction_list;
                next_move->direction_list.push_back(3);
                processing_stack.push(next_move);
                }
              else
                delete next_move;
              }
            break;

            case 3:
            if(this_move->recent_position[num_of_named_tile-1].y < (width_of_grid-1))
              {
            //cout << "Agent y: " << this_move->recent_position[num_of_named_tile-1].y << ", edge: "
<< (width_of_grid-1) << endl;
              /** can move down **/

              next_move = new bfs_obj;
              next_move->next_direction = 1;
              i = 0;
            //cout << "Move Down" << endl;
              while(i<num_of_named_tile)
                {
                temp_axis.x = this_move->recent_position[i].x;
                temp_axis.y = this_move->recent_position[i].y;
                next_move->recent_position.push_back(temp_axis);
                //cout << "(" << next_move->recent_position[i].x << ", " << next_move-
```

```
>recent_position[i].y << ")  ";
                ++i;
                }
            /** make the move **/
            move_agent_block(next_move->recent_position, 1);

            a = next_move->recent_position[0].x + next_move->recent_position[0].y *
width_of_grid;
            b = next_move->recent_position[1].x + next_move->recent_position[1].y *
width_of_grid;
            c = next_move->recent_position[2].x + next_move->recent_position[2].y *
width_of_grid;
            d = next_move->recent_position[3].x + next_move->recent_position[3].y *
width_of_grid;
            if(used_pattern[a][b][c][d] != true)
               {
               used_pattern[a][b][c][d] = true;
               next_move->level = this_move->level + 1;
               next_move->direction_list = this_move->direction_list;
               next_move->direction_list.push_back(1);
               processing_stack.push(next_move);
               }
            else
               delete next_move;
            }
         break;

         default:
            cout << "Bug" << endl;
         }
         ++temp;
         --j;
      }
    delete this_move;
    }
  return false;
  }

int tile_moving::manhattan_distance_calculator(bfs_obj* a)
  {
  int m_distance = 0;
  int i;
  i = num_of_named_tile - 1;
  while(i >= 0)
    {
    m_distance = m_distance + abs(goal_position[i].x-a->recent_position[i].x) +
abs(goal_position[i].y-a->recent_position[i].y);
    --i;
```

```
    }
  return m_distance;
  }

bool tile_moving::check_goal(vector<axis> now)
  {
  int i;
  i = width_of_grid - 1;
  while(i >= 0)
    {
    //cout << "i: " << i << endl;
    //cout << "(" << now[i].x << ", " << now[i].y << ")" << ",  " << endl;
    //cout << "(" << goal_position[i].x << ", " << goal_position[i].y << ")" << ",  ";
    if((now[i].x!=goal_position[i].x) || (now[i].y!=goal_position[i].y))
      {
      //cout << "Conflict!" << endl;
      return false;
      }
    --i;
    }
  //cout << endl;
  return true;
  }

void tile_moving::move_agent_block(vector<axis> &position_arr, int direction)
  {
  int i;
  axis next_posit;

  if(direction==0)/** move right **/
    {
    //next_posit
    next_posit.x = position_arr[num_of_named_tile-1].x + 1;
    next_posit.y = position_arr[num_of_named_tile-1].y;
    }
  else if(direction==1)/** move down **/
    {
    next_posit.x = position_arr[num_of_named_tile-1].x;
    next_posit.y = position_arr[num_of_named_tile-1].y + 1;
    }
  else if(direction==2)/** move left **/
    {
    next_posit.x = position_arr[num_of_named_tile-1].x - 1;
    next_posit.y = position_arr[num_of_named_tile-1].y;
    }
  else if(direction==3)/** move up **/
    {
    next_posit.x = position_arr[num_of_named_tile-1].x;
```

```
      next_posit.y = position_arr[num_of_named_tile-1].y - 1;
      }
   else
      {
      cout << "ERROR in function---move_agent_block" << endl;
      }

   i = num_of_named_tile - 2;
   while(i >= 0)
      {
      /** if the position of agent block is same as A or B or C **/
      /** These two blocks have to swap their axis **/
      if(next_posit.x==position_arr[i].x && next_posit.y==position_arr[i].y)
         {
         position_arr[i].x = position_arr[num_of_named_tile-1].x;
         position_arr[i].y = position_arr[num_of_named_tile-1].y;
         break;
         }
      --i;
      }
   position_arr[num_of_named_tile-1].x = next_posit.x;
   position_arr[num_of_named_tile-1].y = next_posit.y;
   i = 0;
   /*cout << "Move to: " << endl;
   while(i < num_of_named_tile)
      {
      cout << "(" << position_arr[i].x << ", " << position_arr[i].y << ")  ";
      ++i;
      }
   cout << endl;*/
   return;
   }

void tile_moving::recover_tiles_position(vector<axis> &record)
   {
   int i = width_of_grid - 1;
   while(i>=0)
      {
      //cout << i << ": ";
      //cout << "(" << record[i].x << ", " << record[i].y << ")" << endl;
      tiles_position[i].x = record[i].x;
      tiles_position[i].y = record[i].y;
      --i;
      }
   }

int main(int argc, char *argv[])
   {
```

```
tile_moving obj;

srand(time(NULL));

//obj.initialization();
if(!obj.depth_first_typical())
   cout << "No Solution" << endl;
else
   {
   //obj.output_backtrace_result();
   cout << "Solution found" << endl;
   }
obj.how_many_nodes();
return 0;
}
```