**16-puzzle game** (Heuristic, Backtracking &amp; Branch and bound algorithms)

**Problem statement:**

This program is a game in which numbers are spread randomly and the player is supposed to arrange them in order with the given keys.

Problem Explanation:

* In this problem there are 15 tiles, which are numbered from 0 – 15.
* The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.
* The initial and goal arrangement is shown by the following figure.

**Steps / Algorithm to solve the problem:**

* Move the empty space in the grid in order from 1 to 15 where A=10 to F=15
* To move the tile you can use your keywords on the keyboard.

W or w - up,

S or s- Down,

A or a- Left,

D or d - Right

* If any other character is used, then it shows a dialogue of the Wrong character being used.

In the puzzle above A=10, B=11, C=12, D=13, E=14, F is 15.

**Sample input and output:**

====================

| F | E |  1 |  6  |

|  9 | B |  4 | C  |

|    | A |  7 |  3  |

| D |  8 |  5 |  2  |

 ====================

RRRULDDLUUULDRURDDDRULLULURRRDDLDLUURDDLULURRULDRDRD 52 moves

Found the shortest path is 1130063 steps and 17.61 seconds

 ====================

|  1 |  2 |  3 |  4  |

|  5 |  6 |  7 |  8  |

|  9 |  A | B  |  C  |

|  D  | E | F  |     |

 ====================

**Program code in C++:**

#include <iostream>

using std::cout;

using std::cin;

using std::endl;;

#include <cstdlib>

using std::rand;

using std::srand;

#include <ctime>

using std::time;

const int arraySize = 4;

char elements[ arraySize ] [ arraySize ] = {{'1','2','3','4'},

{'5','6','7','8'},

{'9','A','B','C'},

{'D','E',' ','F'}};

char check [ arraySize ] [ arraySize ] = {{'1','2','3','4'},

{'5','6','7','8'},

{'9','A','B','C'},

{'D','E','F',' '}};

int vSP = 3; // vertical space Position

int hSP = 2; // horizontal space Position

void moveUp();

void moveDown();

void moveRight();

void moveLeft();

void randomise(); // randomize the array

int winer(); // checks if player have solved the puzzle

int main()

{

srand(time(0));

randomise();

bool quite(false);

do

{

for(int i = 0; i < arraySize; i++){

for(int j = 0; j < arraySize; j++)

cout << " " << elements[i][j];

cout << endl << endl;

}

char a;

cout << "w - Up, z - Down, a - Left, s - Right" << endl;

cin >> a;

switch(a)

{

case 'W':

case 'w':

moveUp();

break;

case 'Z':

case 'z':

moveDown();

break;

case 's':

case 'S':

moveRight();

break;

case 'a':

case 'A':

moveLeft();

break;

default:

cout << "Wrong character, please type again!" << endl;

break;

}

int c = winer();

if(c == 1){

cout << "Bravo! You solved the puzzle!" << endl;

quite = true;

}

system("cls");

}while(quite == false);

return 0;

}

void moveUp()

{

int vP = vSP;

if(vP + 1 < 4 && vP >= 0){

elements[vSP][hSP] = elements[vSP + 1][hSP];

elements[vSP + 1][hSP] = ' ';

vSP += 1;

}

}

void moveDown()

{

int vP = vSP;

if(vP + 1 <= 4 && vP > 0){

elements[vSP][hSP] = elements[vSP - 1][hSP];

elements[vSP - 1][hSP] = ' ';

vSP -= 1;

}

}

void moveRight()

{

int hP = hSP;

if(hP + 1 <= 4 && hP > 0){

elements[vSP][hSP] = elements[vSP][hSP - 1];

elements[vSP][hSP - 1] = ' ';

hSP -= 1;

}

}

void moveLeft()

{

int hP = hSP;

if(hP + 1 < 4 && hP >= 0){

elements[vSP][hSP] = elements[vSP][hSP + 1];

elements[vSP][hSP + 1] = ' ';

hSP += 1;

}

}

void randomise()

{

for(int i = 0; i < 20000; i++)

{

int a = 1 + rand() % 4;

switch(a)

{

case 1:

moveUp();

break;

case 2:

moveDown();

break;

case 3:

moveRight();

break;

case 4:

moveLeft();

break;

}

}

}

int winer()

{

int ans;

for(int i = 0; i < arraySize; i++){

for(int j = 0; j < arraySize; j++){

if(elements[i][j] == check[i][j])

ans = 1;

else

return -1;

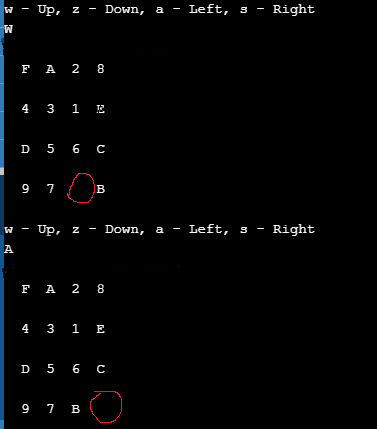
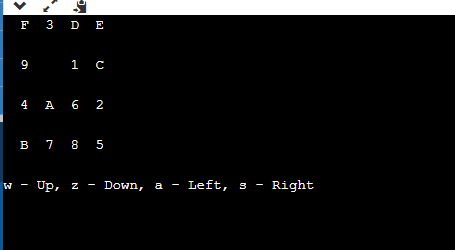
}

}

return 1;

}

**Program compilation:**



**Branch and Bound:**

The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS-like search.

There are basically three types of nodes involved in Branch and Bound

1. **Live node** is a node that has been generated but whose children have not yet been generated.

2. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

3. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

**Cost function:**

Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where

   g(X) = cost of reaching the current node

          from the root

   h(X) = cost of reaching an answer node from X.

In the above program, heuristics for all tiles on the board are as follows.

1. Σ(AP-GP)^2:

(Actual Position- Goal Position)^2:

(7-1)^2+(12-2)^2+(2-3)^2+(9-4)^2+(16-5)^2+(11-6)^2+(14-7)^2+(15-8)^2+(5-9)^2+(10-10)^2+(13-11)^2+ (8-12)^2+ (3-13)^2+(4-14)^2+(1-15)^2

=36 + 100 +1 + 25 + 121 + 25 + 49 + 49 +16 + 0 + 4 + 16 + 100 + 100 + 196

= 838

1. Absolute Value:Σ|AP-GP|

= 7-1 + 12-2 + 2-3 + 9-4 + 16-5 +11-6 +14-7 + 15-8 + 5-9 + 10-10 + 13-11 +8-12 + 3-13 +4-14 + 1-15

= 6 + 10 + 1 + 5 + 11 + 5 + 7 + 7 + 4 + 0 + 2 + 4 + 10 + 10 +14

= 96

1. ΣManhattan Distance:

= 3+4+1+5+5+2+3+3+1+0+3+1+5+5+6

=47

**Time Complexity:**

O(n^n) is definitely an upper bound on solving n-puzzle using backtracking when you are solving this by assigning an empty tile column-wise. However, consider this - when we assign a location of the empty tile in the first column, you have n options, after that, you only have n-1 options.

If we add all this up and define the run time as T(N). Then T(N) = O(N2) + N\*T(N-1). If you draw a recursion tree using this recurrence, the final term will be something like n3+ n!O(1). By the definition of Big O, this can be reduced to O(n!) running time.

Thus, the worst-case complexity is still upper bounded by O(n!).