**Comparison of Breadth-First, Depth-First and Iterative-Deepening Searches for the 8-Puzzle problem**

The following tables show the outputs that I received for input shuffling factors of 5,10,15,20 and 25. Please note that an ‘F’ indicates that it became too complex for that search paradigm to solve the problem.

The first table is for the **number of goal state checks** made.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Breadth-First | Depth-First | Iterative-Deepening |
| Shuffling Factor - 5 | **39** | **28** | **36** |
| Shuffling Factor – 10 | **543** | **589** | **266** |
| Shuffling Factor – 15 | **F** | **20447** | **16256** |
| Shuffling Factor – 20 | **670** | **F** | **233** |
| Shuffling Factor - 25 | **147375** | **93988** | **4941** |

The second table is for the **max size of the stack (or queue)** used in the search process

|  |  |  |  |
| --- | --- | --- | --- |
|  | Breadth-First | Depth-First | Iterative-Deepening |
| Shuffling Factor - 5 | **69** | **24** | **4** |
| Shuffling Factor – 10 | **999** | **447** | **7** |
| Shuffling Factor – 15 | **F** | **14773** | **13** |
| Shuffling Factor – 20 | **1212** | **F** | **7** |
| Shuffling Factor - 25 | **270589** | **59868** | **12** |

Note: The failing factor that I have set in the previous experiment was as follows:

For BFS: If number of goal state checks exceeds 1000000, then it fails.

For DFS and IDS: If number of goal state checks exceeds 100000, then it fails.

The reason I set a lesser failure limit for DFS and IDS is that they take more time to complete per node, because before inserting any node, the algorithm checks across its entire range of ancestors to determine there is no duplicate (this is to prevent infinite looping). So the machine overhead incurred in my experiment is larger for DFS and IDS.

From the results, it is clear that IDS clearly defeats both BFS and DFS. Especially in terms of space used, the maximum size it ever reached was 13 nodes. This is because IDS uses the best heuristics of BFS and DFS, i.e. the number of goal state checks of BFS and the maximum size factor of DFS.

You can also observe that in my experiments, BFS does not quite live up to its billing of being better than DFS in terms of goal state checks (as we had derived in class). That is because, we derived the formula keeping in mind an average case (where the goal node is in the right extreme at a depth of ‘d’). So possibly in my experiments, it is possible that the nodes were at a much greater depth but more towards the center or left end of that row. So essentially, unless I run an enormous number of cases, I cannot empirically prove the superiority of BFS over DFS for the number of goal state checks.

**Another issue:** I believe that shuffling mechanism introduced in the experiment might not be a very good way to study these search techniques. For example, take BFS. For a shuffling factor of 15, it exceeded 10^6 nodes (and failed), but solved the problem with a shuffling factor of 20 in just 1212 steps.

I think, a better shuffling factor can be defined as follows: F(N), where for a given input N, it takes any search at least N steps to get to the solution. So essentially, in this shuffling, we cannot have a case where the blank space (say) swaps with the piece ‘8’ and then vice versa. The subsequent step cannot backtrack the previous steps.

Out of curiosity, I ran the BFS (without limits) and IDS (retained limit here – 10^5) for large shuffling factor values. The results are shown as follows

The first table shows the number of goal node checks and the second shows the maximum size of the stack (or queue). **Note** that for Shuffling Factor – 100 (II attempt) I turned off the 10^5 limit check for IDS.

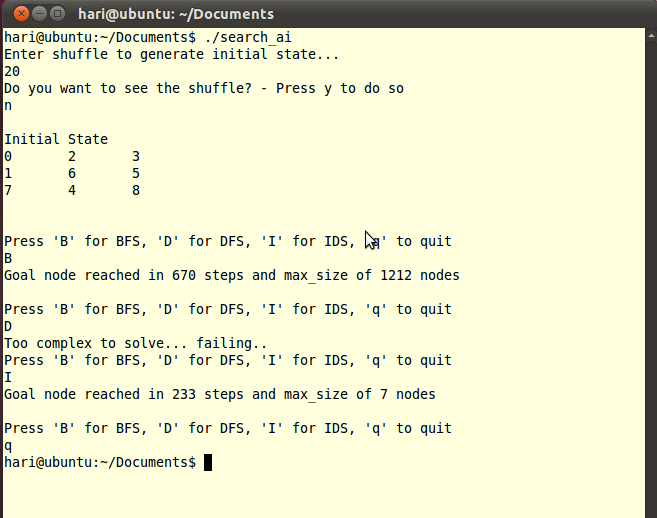
|  |  |  |
| --- | --- | --- |
|  | Breadth-First | Iterative-Deepening |
| Shuffling Factor – 85 | **997788** | **18117** |
| Shuffling Factor - 90 | **334649** | **7105** |
| Shuffling Factor - 100 | **Crashed my Computer** | **F** |
| Shuffling Factor – 100 (II attempt) | **Did not try (after what happened previously)** | **2259823 (no limit)** |

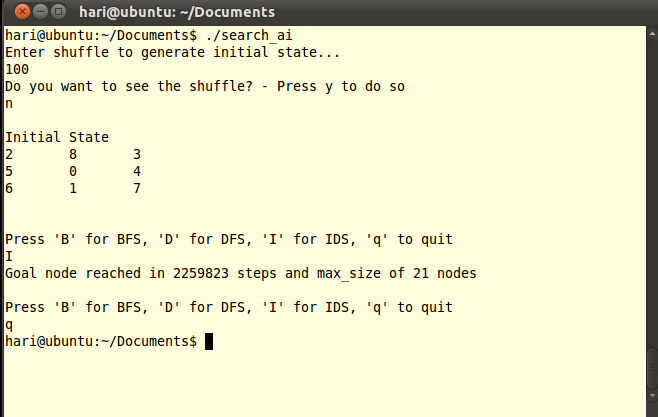
|  |  |  |
| --- | --- | --- |
|  | Breadth-First | Iterative-Deepening |
| Shuffling Factor – 85 | **1862708** | **13** |
| Shuffling Factor - 90 | **607677** | **12** |
| Shuffling Factor - 100 | **Crashed my Computer** | **F** |
| Shuffling Factor – 100 (II attempt) | **Did not try** | **21** |

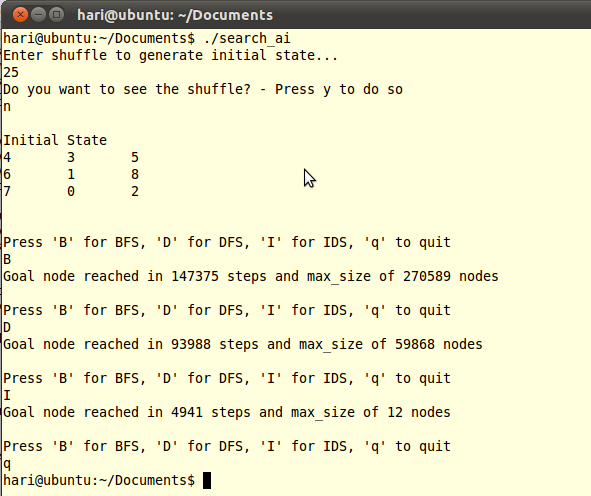
This shows how well IDS scales for complex search instances as well. Also, regarding the computer crash – it could indicate a bad coding style and is mostly because I did not free the memory used by the temporary board object pointers. If I did that, there is a chance the crash can be avoided.

So these experiments have conclusively (rather empirically) proved that IDS is the best search method.

**Screenshots (for shuffling factors – 20, 100 and 25 respectively)**







**Source Code**

The next few pages contain the source code. It is in C++ and was compiled using a g++ compiler on a Linux Ubuntu machine.

# include <stdio.h>

# include <stdlib.h>

# include <time.h>

# include <queue>

# include <stack>

using namespace std;

class board

{

int a[3][3]; //3\*3 matrix to represent the board.

int blank\_space; //position of blank space (0 - 8).

int depth; //depth of the node in the search tree - used for iterative deepening.

board\* parent;

public:

board() //goal state - default constructor

{

int i,j,count=0;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

if (i==1 && j==1)

a[i][j] = 0;

else

a[i][j] = ++count;

}

}

blank\_space = 4;

}

board (board\* b) //Constructor to copy from a previous board.

{

int pos;

for (pos=0;pos<=8;pos++)

a[pos/3][pos%3] = b->a[pos/3][pos%3];

blank\_space = b->blank\_space;

}

int return\_pos (int pos) //function to return value at location pos.

{

return a[pos/3][pos%3];

}

int bs () //function to return blank space location.

{

return this->blank\_space;

}

void set\_parent(board\* par)

{

this->parent = par;

}

board\* get\_parent()

{

return this->parent;

}

void set\_depth(int dep)

{

this->depth = dep;

}

int get\_depth ()

{

return this->depth;

}

void swap(int blank, int pos)

{

if (blank != blank\_space)

{

printf("Fault in generating swap function\nExiting...\n");

exit(0);

}

int temp;

a[blank/3][blank%3] = a[pos/3][pos%3];

a[pos/3][pos%3] = 0;

blank\_space = pos;

}

void print()

{

int i,j;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

printf("%d\t",a[i][j]);

}

printf("\n");

}

printf("\n\n");

}

bool isGoal () //Checks if current state is Goal State.

{

int i,j,count=0;

;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

if (i==1 && j==1)

{

if (a[i][j] != 0)

return false;

}

else

{

count++;

if (a[i][j] != count)

return false;

}

}

}

return true;

}

bool isEqual (board\* b) //Checks if current board is same as board b.

{

int i,j;

for(i=0;i<3;i++)

{

for(j=0;j<3;j++)

{

if (a[i][j] != b->a[i][j])

return false;

}

}

return true;

}

void initial\_state (int shuffle, bool show\_shuffle) //function that generates a start state shuffled 'shuffle' times from goal state.

{

int no\_shuffle;

srand(time(NULL)); //initializing the seed for randomize function.

int random\_no;

for(no\_shuffle=0; no\_shuffle<shuffle; no\_shuffle++)

{

if (blank\_space == 4)

{

random\_no = rand() % 4; //Since there are four places for blank to move.

if (random\_no == 0)

swap (blank\_space,1);

else if (random\_no == 1)

swap (blank\_space,3);

else if (random\_no == 2)

swap (blank\_space,5);

else

swap (blank\_space,7);

}

else if (blank\_space == 0 || blank\_space == 2 || blank\_space == 6 || blank\_space == 8)

{

random\_no = rand() % 2;

if (random\_no == 0)

{

if (blank\_space == 0 || blank\_space == 2)

swap (blank\_space, 1);

else

swap (blank\_space, 7);

}

else

{

if (blank\_space == 0 || blank\_space == 6)

swap (blank\_space, 3);

else

swap (blank\_space, 5);

}

}

else

{

random\_no = rand() % 3;

if(random\_no == 0)

swap (blank\_space,4);

else if (random\_no == 1)

{

if (blank\_space == 1 || blank\_space == 3)

swap(blank\_space,0);

else

swap (blank\_space, 8);

}

else

{

if (blank\_space == 1 || blank\_space == 5)

swap (blank\_space, 2);

else

swap (blank\_space, 6);

}

}

if (show\_shuffle)

this->print();

}

printf("\nInitial State\n");

this->print();

}

};

bool no\_repeat (board\* b) //Compare node b with all its ancestors. If repeat of any ancestor node return false, else return true.

{

board\* temp = b;

while (temp->get\_parent())

{

temp=temp->get\_parent();

if(b->isEqual(temp))

return false;

}

return true;

}

void breadth\_search (board\* b, int\* number\_of\_tests, int \*max\_size) //Breadth First Search Code.

{

board\* temp = b;

\*max\_size = 0;

\*number\_of\_tests = 0;

queue<board\*> q;

q.push(temp);

while(!q.empty())

{

if (q.size()>(\*max\_size))

(\*max\_size) = q.size();

(\*number\_of\_tests)++;

if ((\*number\_of\_tests) > 1000000) //Too many tests performed. Integer becomes negative .Exit.

{

printf("Too complex to solve... failing..\n");

return;

}

if (temp->isGoal()) //Check for Goal status in current node.

{

printf("Goal node reached in %d steps and max\_size of %d nodes\n\n", \*number\_of\_tests, \*max\_size);

return;

}

if ((temp->bs())%3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs()) - 1);

q.push(new\_b);

}

if ((temp->bs())%3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+1);

q.push(new\_b);

}

if ((temp->bs())/3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs())-3);

q.push(new\_b);

}

if ((temp->bs())/3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+3);

q.push(new\_b);

}

temp = q.front();

q.pop();

}

}

void depth\_search (board\* b, int\* number\_of\_tests, int \*max\_size)

{

board\* temp = b;

temp->set\_parent(NULL); //Set parent of goal node to be NULL.

\*max\_size = 0;

\*number\_of\_tests = 0;

stack<board\*> s;

s.push(temp);

while(!s.empty())

{

if (s.size()>(\*max\_size))

(\*max\_size) = s.size();

(\*number\_of\_tests)++;

if ((\*number\_of\_tests) > 100000) //Too many tests performed. Integer becomes negative .Exit.

{

printf("Too complex to solve... failing..\n");

return;

}

if (temp->isGoal())

{

printf("Goal node reached in %d steps and max\_size of %d nodes\n\n", \*number\_of\_tests, \*max\_size);

return;

}

if ((temp->bs())%3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs()) - 1);

new\_b->set\_parent(temp);

if (no\_repeat(new\_b)) //This function checks if temp is not a duplicate board arrangement of any of its ancestors. If we don't check this DFS may go on infinitely.

s.push(new\_b);

}

if ((temp->bs())%3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+1);

new\_b->set\_parent(temp);

if (no\_repeat(new\_b))

s.push(new\_b);

}

if ((temp->bs())/3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs())-3);

new\_b->set\_parent(temp);

if (no\_repeat(new\_b))

s.push(new\_b);

}

if ((temp->bs())/3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+3);

new\_b->set\_parent(temp);

if (no\_repeat(new\_b))

s.push(new\_b);

}

temp = s.top();

s.pop();

}

}

void iterative\_search (board\* b, int\* number\_of\_tests, int \*max\_size)

{

int depth\_iter = 0;

int depth;

\*max\_size = 0;

\*number\_of\_tests = 0;

while (1)

{

depth = 0;

board\* temp = b;

temp->set\_parent(NULL); //Set parent of goal node to be NULL.

temp->set\_depth(0);

stack<board\*> s;

s.push(temp);

while(!s.empty())

{

if ((temp->get\_depth()) > depth\_iter)

{

while ((temp->get\_depth() > depth\_iter) && s.size() > 0)

{

temp = s.top();

s.pop();

}

}

if (s.size() == 0 && temp->get\_depth() > depth\_iter)

break;

if (s.size()>(\*max\_size))

(\*max\_size) = s.size();

(\*number\_of\_tests)++;

if ((\*number\_of\_tests) > 100000) //Too many tests performed. Integer becomes negative .Exit.

{

printf("Too complex to solve... failing..\n");

return;

}

if (temp->isGoal())

{

printf("Goal node reached in %d steps and max\_size of %d nodes\n\n", \*number\_of\_tests, \*max\_size);

return;

}

if ((temp->bs())%3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs()) - 1);

new\_b->set\_parent(temp);

new\_b->set\_depth(temp->get\_depth() + 1);

if (no\_repeat(new\_b)) //This function checks if temp is not a duplicate board arrangement of any of its ancestors. If we don't check this DFS may go on infinitely.

s.push(new\_b);

}

if ((temp->bs())%3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+1);

new\_b->set\_parent(temp);

new\_b->set\_depth(temp->get\_depth() + 1);

if (no\_repeat(new\_b))

s.push(new\_b);

}

if ((temp->bs())/3 != 0)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(), (new\_b->bs())-3);

new\_b->set\_parent(temp);

new\_b->set\_depth(temp->get\_depth() + 1);

if (no\_repeat(new\_b))

s.push(new\_b);

}

if ((temp->bs())/3 != 2)

{

board \*new\_b = new board(temp);

new\_b->swap(new\_b->bs(),(new\_b->bs())+3);

new\_b->set\_parent(temp);

new\_b->set\_depth(temp->get\_depth() + 1);

if (no\_repeat(new\_b))

s.push(new\_b);

}

temp = s.top();

s.pop();

}

depth\_iter++;

if (s.empty())

continue;

}

}

int main()

{

int shuffle; char ch; bool show\_shuffle;

char option;

int number\_of\_tests, max\_size;

printf("Enter shuffle to generate initial state...\n");

scanf("%d",&shuffle);

while(getchar() != '\n'); //flushing the input buffer.

while(shuffle > 100)

{

printf("Shuffle should be <= 100\n");

scanf("%d", &shuffle);

}

printf("Do you want to see the shuffle? - Press y to do so\n");

fflush(stdin);

scanf("%c",&ch);

while(getchar() != '\n');

board \*b = new board();

if (ch == 'y')

show\_shuffle = true;

else

show\_shuffle = false;

b->initial\_state(shuffle, show\_shuffle);

while(1)

{

printf("Press 'B' for BFS, 'D' for DFS, 'I' for IDS, 'q' to quit\n");

scanf("%c",&option);

while(getchar() != '\n');

switch (option)

{

case 'B': breadth\_search (b, &number\_of\_tests,&max\_size);

break;

case 'D': depth\_search (b,&number\_of\_tests,&max\_size);

break;

case 'I': iterative\_search (b, &number\_of\_tests, &max\_size);

break;

case 'q': return 1;

default: printf("Invalid option\n");

break;

}

}

}