PROBLEM STATEMENT :

Implementation of any 2 uninformed search methods with some application.

OBJECTIVE :

• To develop problem solving abilities for gamifications.
• To apply algorithmic strategies while solving problems
• To develop time and space efficient algorithms

THEORY :

Uninformed Search :

An uninformed (a.k.a. blind, brute-force) search algorithm generates the search tree without using any domain specific knowledge. The two basic approaches differ as to whether you check for a goal when a node is *generated* or when it is *expanded*.

Breadth-First Search

Strategy: expand the shallowest unexpanded node. Implementation: The fringe is a FIFO queue.

- is complete (if *b* is finite)
- is optimal if all path costs are the same (because it always finds the shallowest node first)

Depth-First Search

Strategy: expand the deepest unexpanded node. Implementation: The fringe is a LIFO queue (stack)

- is not complete (because of infinite depth and loops)
- is not optimal

Depth-Limited Search

This is just a depth-first search with a cutoff depth. Here is the algorithm (for the test-at-expansion-time case)

```
fringe := [make_node(start_state, null, null)]
reached_limit := false
while fringe is not empty
   n := fringe.pop()
   if n.state is a goal state return n.actionListFromRoot()
   if n.depth == limit
      reached_limit := true
   else
      for each action a applicable to n.state
         fringe.push(make_node(succ(n.state, a), a, n))
```

return reached_limit ? cutoff : failure

- Won’t run forever unless *b* is infinite
- is not complete because a goal may be below the cutoff
- is not optimal

Depth-First Iterative Deepening

Algorithm:

```
for i in 1..infinity
   run DLS to level i
   if found a goal at level i, return it immediately
```

Uniform Cost Search

Strategy: Expand the lowest cost node. Implementation: the fringe is a priority queue: lowest cost node has the highest priority. In order to be optimal, must test at expansion, not generation, time.

Backwards Chaining

Run the search backwards from a goal state to a start state. Obiously this works best when the actions are reversible, and the set of goal states is small.

Bidirectional Search

Run a search forward from the start state and simulatneously. Motivation is that *bd*2+*bd*2 is much, much less than *bd*. Works best when the backwards search is feasible. Problem is space complexity: one of the trees has to be kept in memory so we can test membership for a node generated in the other tree.

ALGORITHM

Checking at generation time:

```
if start_state is a goal state return the empty action list
        fringe := [make_node(start_state, null, null)]
                while fringe is not empty
                        n := select and remove some node from the fringe
                        for each action a applicable to n.state
                        s := succ(n.state, a)
                        n' := make_node(s, a, n)
                        if s is a goal state, return n'.actionListFromRoot()
                        add n' to fringe
                return failure
```

Checking at expansion time:

```
fringe := [make_node(start_state, null, null)]
while fringe is not empty
   n := select and remove some node from the fringe
   if n.state is a goal state return n.actionListFromRoot()
   for each action a applicable to n.state
      add make_node(succ(n.state, a), a, n) to fringe
return failure
```

INPUT :

Goal state in (123_45678) format where '_' represents the blank tile .

EXPECTED OUTPUT :

Solution found at particular depth "d".
Solution not found.

MATHEMATICAL MODEL :

Let S be the solution perspective .

S={s, e, i, o, f, DD, NDD, success, failure}

s = { Initial state of the 8-tile puzzel consisting of all tiles at particular places }

I = Input of the system → { I1, I2 }

where I1 = { Initial position of tiles }
I2 = { Final position of tiles }

o = Output of the system → { O1 , O2 }

where O1 = { Goal state reached }
O2 = { Goal state not reached }

f = Functions used → { f1, f2}

where f1 = { Depth first search }
f2 = { Breadth first search }

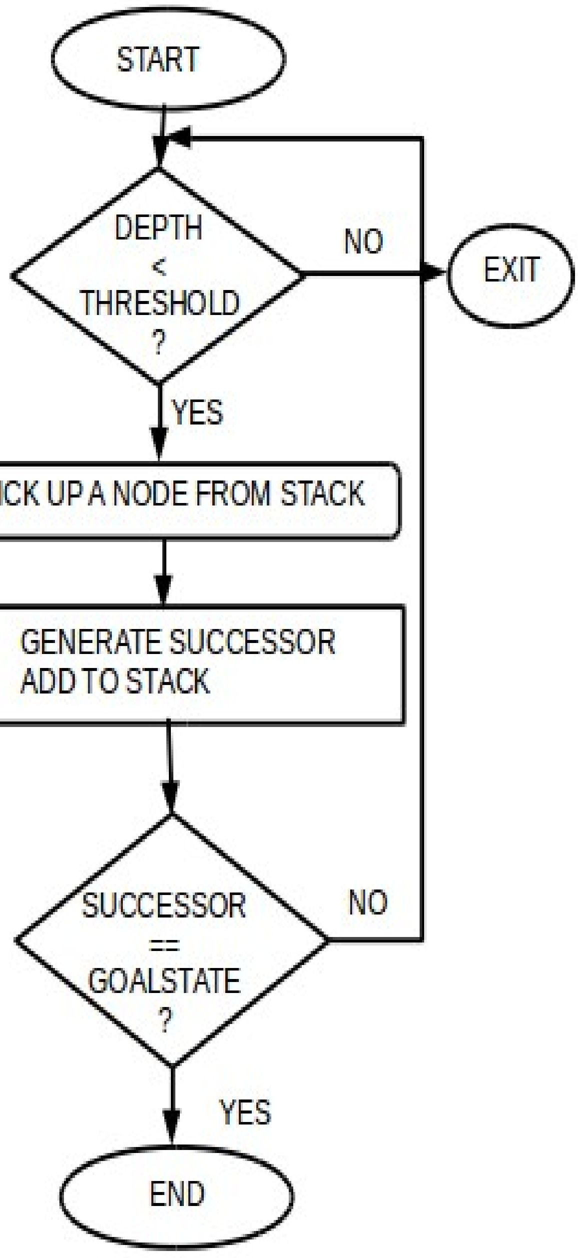
DD - Deterministic data → { Initial position of tiles }

NDD - Non deterministic data → { Number of moves }

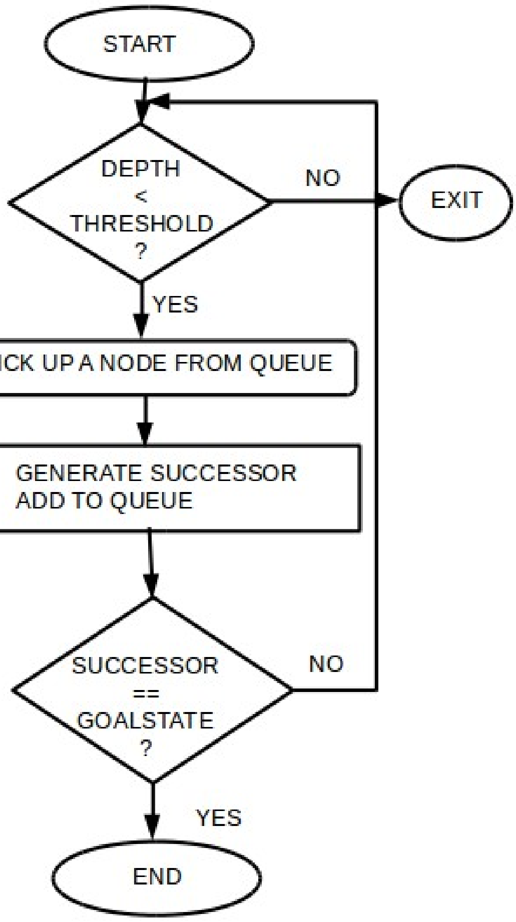
Success - Desired outcome generated → {Goal state reached }

Failure - Desired outcome not generated or forced exit due to system error.

FLOW CHART (DFS):



FLOW CHART (BFS):



TEST CASES : (DFS)

TEST CASE	INPUT	EXPECTED OUTPUT	OUTPUT ACHIEVED	REMARKS
1.	1_3425678	Solution found at depth 2	Solution found at depth 2	Correct
2.	_13425678	Solution found at depth 3	Solution found at depth 3	Correct
3.	1_3456728	Solution not found	Solution not found	Corrrect

TEST CASES : (BFS)

TEST CASE	INPUT	EXPECTED OUTPUT	OUTPUT ACHIEVED	REMARKS
1.	1234756_8	Solution Found At Depth 2	Solution Found At Level 2	Correct
2.	123_45678	Solution Found At Depth 2	Solution Found At Level 2	Correct
3.	1_4567823	No Solution	Solution Not Found Depth Exceeded	Corrrect

TIME AND SPACE COMPLEXITY :

<u>Breadth First Search</u>: Time complexity (worst case, goal at rightmost end of level *d*):

Test at generation: 1+*b*+*b*2+...+*bd*=*O*(*bd*)

Space complexity is same as time complexity because every node has to stay in memory.

<u>Depth First Search</u>: Time complexity (worst case: solution is at m): $O(b^m)$ — regardless whether we test at generation or expansion

Space complexity is *O*(*bm*) — linear space . Only the nodes on the current path are stored .

CONCLUSION :

Hence the implementation of two uninformed search techniques ie BFS and DFS has been successfully implemented.

OUTCOMES ACHIEVED

COURSE OUTCOME	**ACHIEVED**(√)
Problem solving abilities for smart devices.	
Problem solving abilities for gamifications.	√
Problem solving abilities of pervasiveness,embedded security and NLP.	
To solve problems for multicore or distributed,concurrent/Parallel environments	