625 Programming Assignment 1

Unnati Eramangalath

UIN: 930001393

PART 1: SEARCH

AIM: Implement six search algorithms to solve 8-puzzle: DFS, BFS, IDS, greedy best-first (hence-forth ‘‘greedy’’), a-star, IDA-star.
- Test and compare time and space complexity for all cases.
- Test and compare the effect of different heuristic functions (for the informed search algorithms).

8-PUZZLE

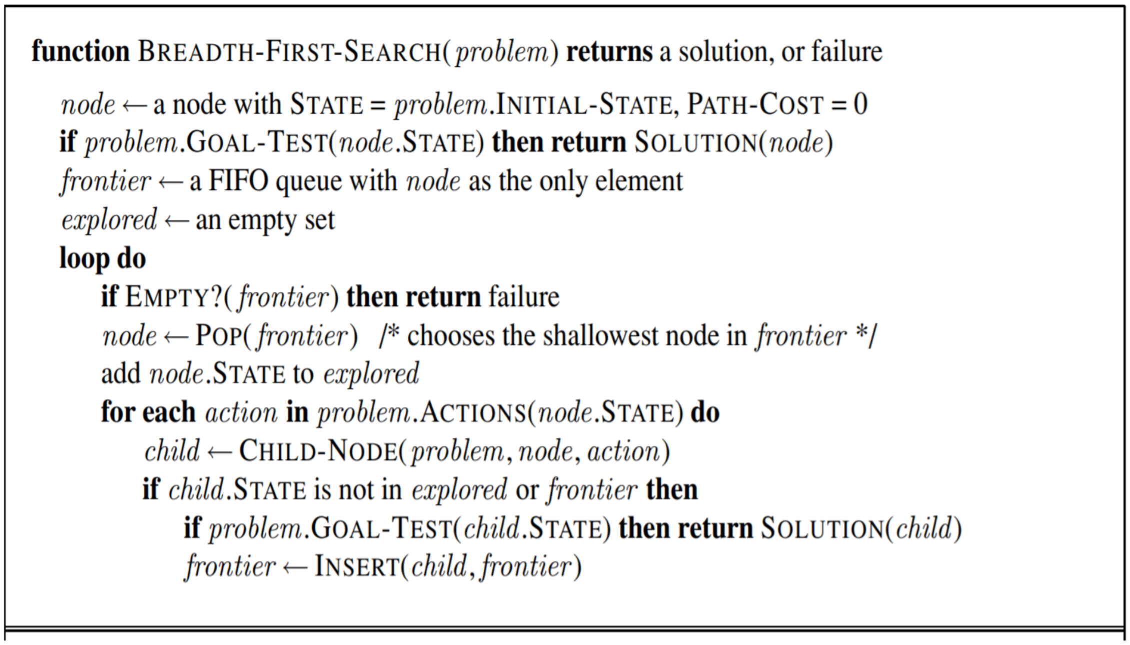
Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

I have implemented 6 search Algorithms to solve 8-Puzzle problem for the given goal state:
1 2 3 8 0 4 7 6 5.

Following are the results obtained post testing the algorithms for 3 initial states.

1. Breadth First Search

BFS is a traversing algorithm where we should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbor nodes (nodes which are directly connected to source node). In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

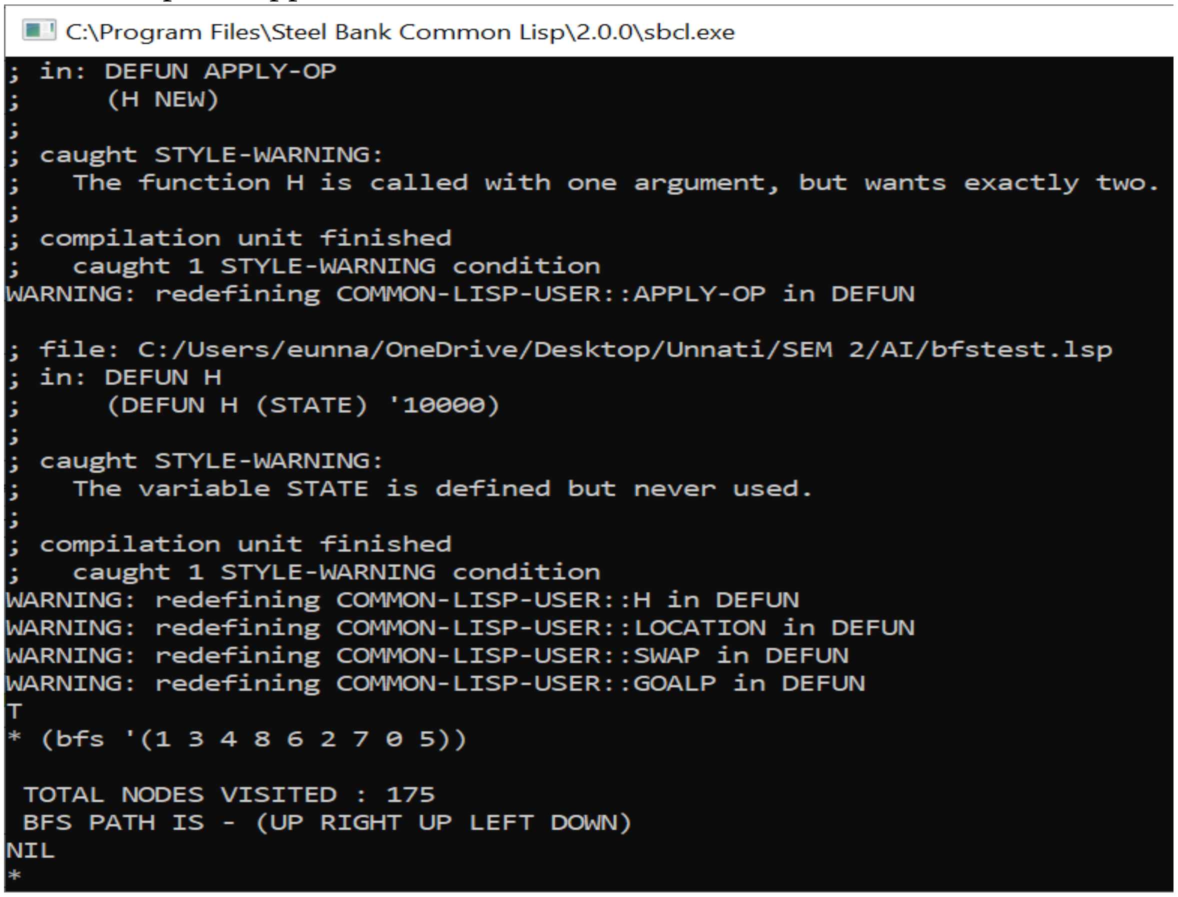


```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Fig.1 ALGORITHM

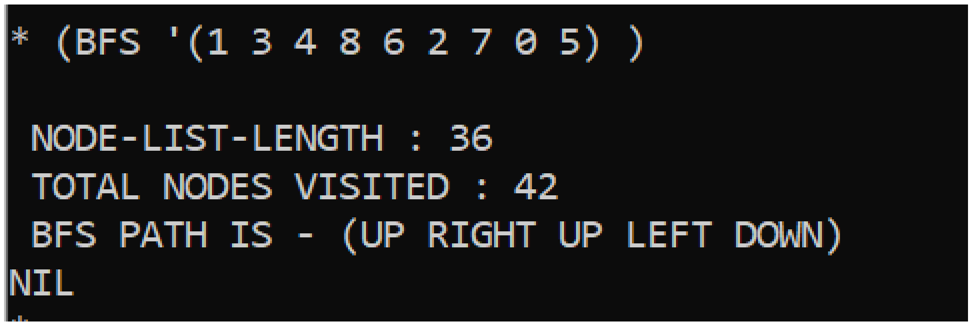
Results:

1. Easy: (1 3 4 8 6 2 7 0 5)
   Output Snippet:



```
C:\Program Files\Steel Bank Common Lisp\2.0.0\sbcl.exe
; in: DEFUN APPLY-OP
;     (H NEW)
;
; caught STYLE-WARNING:
;   The function H is called with one argument, but wants exactly two.
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
WARNING: redefining COMMON-LISP-USER::APPLY-OP in DEFUN

; file: C:/Users/eunna/OneDrive/Desktop/Unnati/SEM 2/AI/bfstest.lsp
; in: DEFUN H
;     (DEFUN H (STATE) '10000)
;
; caught STYLE-WARNING:
;   The variable STATE is defined but never used.
;
; compilation unit finished
;   caught 1 STYLE-WARNING condition
WARNING: redefining COMMON-LISP-USER::H in DEFUN
WARNING: redefining COMMON-LISP-USER::LOCATION in DEFUN
WARNING: redefining COMMON-LISP-USER::SWAP in DEFUN
WARNING: redefining COMMON-LISP-USER::GOALP in DEFUN
T
* (bfs '(1 3 4 8 6 2 7 0 5))

 TOTAL NODES VISITED : 175
 BFS PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
*
```

WITHOUT DUPE IMPLEMENTATION

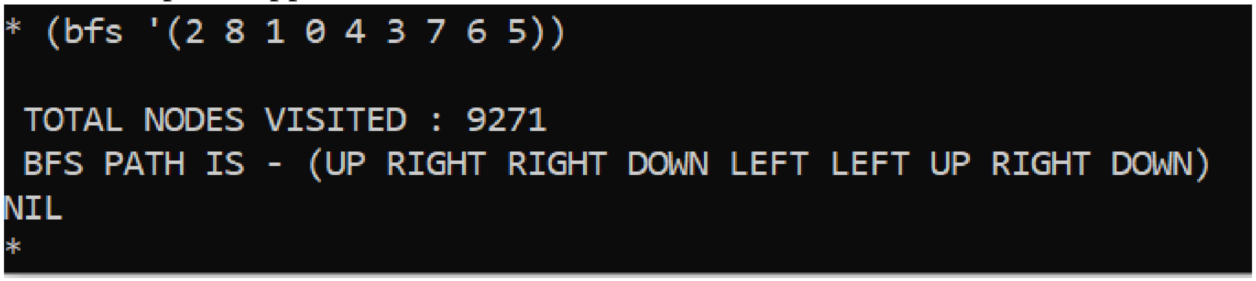


```
* (BFS '(1 3 4 8 6 2 7 0 5) )

 NODE-LIST-LENGTH : 36
 TOTAL NODES VISITED : 42
 BFS PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
```

WITH DUPE IMPLEMENTATION

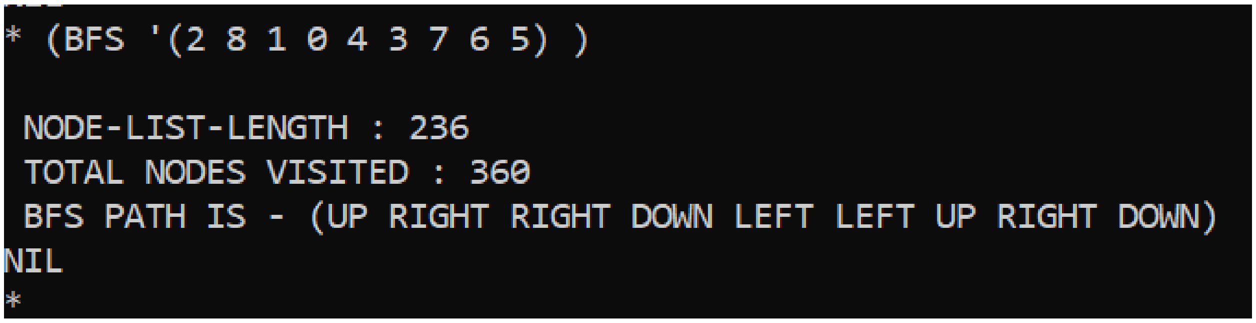
We observe the output sequence of moves to be (UP RIGHT UP LEFT DOWN) and in reaching the goal state it visits a total of 175 nodes before it encounters the node state for without duplicate checking and 42 nodes with duplicate checking. Node list length is 36.

2. Medium: (2 8 1 0 4 3 7 6 5)
   Output Snippet:



```
* (bfs '(2 8 1 0 4 3 7 6 5))

 TOTAL NODES VISITED : 9271
 BFS PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
NIL
*
```

WITHOUT DUPE IMPLEMENTATION

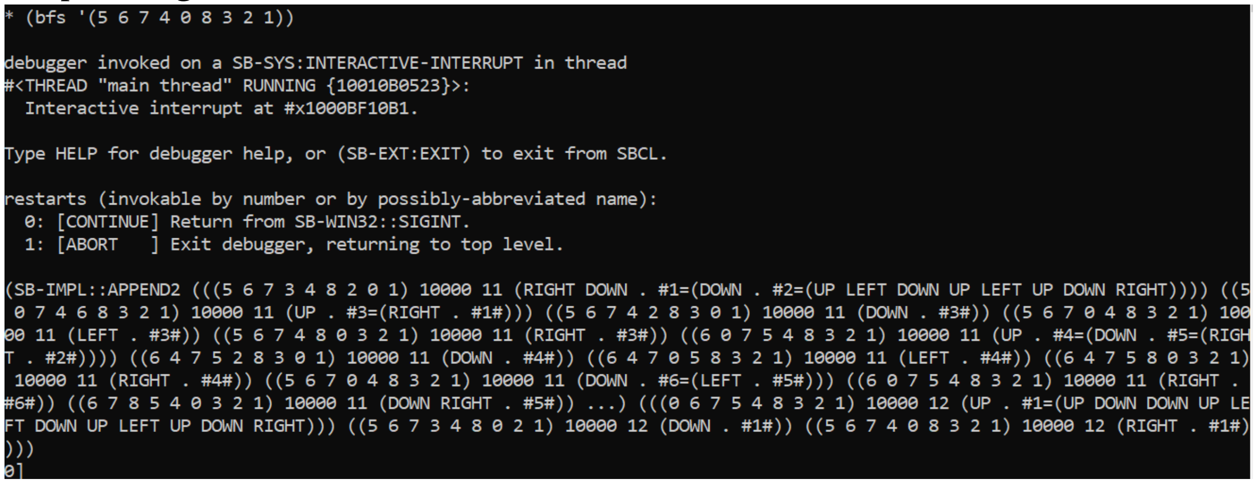


```
* (BFS '(2 8 1 0 4 3 7 6 5) )

 NODE-LIST-LENGTH : 236
 TOTAL NODES VISITED : 360
 BFS PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
NIL
*
```

WITH DUPE IMPLEMENTATION

We observe the output sequence of moves to be (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN) and in reaching the goal state it visits a total of 9271 nodes before it encounters the node state without duplicate checking and 360 with dupe implementation.

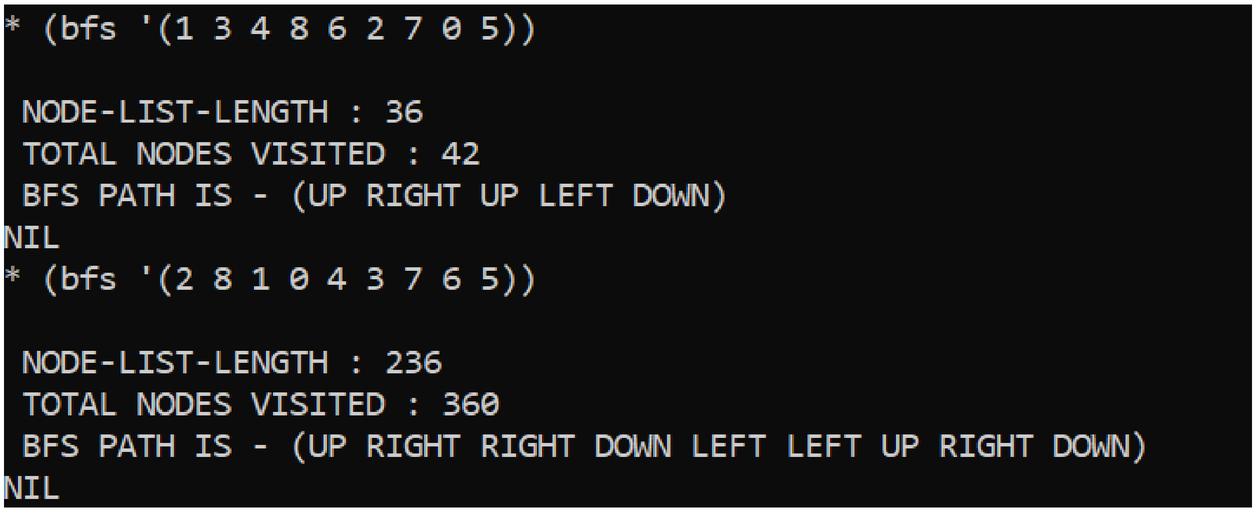
3. Hard: (5 6 7 4 0 8 3 2 1)
   Output: Algorithm ran for more than 10 mins. Had to terminate it.



```
* (bfs '(5 6 7 4 0 8 3 2 1))

debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread
#<THREAD "main thread" RUNNING {10010B0523}>:
  Interactive interrupt at #x1000BF10B1.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE] Return from SB-WIN32::SIGINT.
  1: [ABORT   ] Exit debugger, returning to top level.

(SB-IMPL::APPEND2 (((5 6 7 3 4 8 2 0 1) 10000 11 (RIGHT DOWN . #1=(DOWN . #2=(UP LEFT DOWN UP LEFT UP DOWN RIGHT)))) ((5
 0 7 4 6 8 3 2 1) 10000 11 (UP . #3=(RIGHT . #1#))) ((5 6 7 4 2 8 3 0 1) 10000 11 (DOWN . #3#)) ((5 6 7 0 4 8 3 2 1) 100
00 11 (LEFT . #3#)) ((5 6 7 4 8 0 3 2 1) 10000 11 (RIGHT . #3#)) ((6 0 7 5 4 8 3 2 1) 10000 11 (UP . #4=(DOWN . #5=(RIGH
T . #2#)))) ((6 4 7 5 2 8 3 0 1) 10000 11 (DOWN . #4#)) ((6 4 7 0 5 8 3 2 1) 10000 11 (LEFT . #4#)) ((6 4 7 5 8 0 3 2 1)
 10000 11 (RIGHT . #4#)) ((5 6 7 0 4 8 3 2 1) 10000 11 (DOWN . #6=(LEFT . #5#))) ((6 0 7 5 4 8 3 2 1) 10000 11 (RIGHT .
#6#)) ((6 7 8 5 4 0 3 2 1) 10000 11 (DOWN RIGHT . #5#)) ...) (((0 6 7 5 4 8 3 2 1) 10000 12 (UP . #1=(UP DOWN DOWN UP LE
FT DOWN UP LEFT UP DOWN RIGHT))) ((5 6 7 3 4 8 0 2 1) 10000 12 (DOWN . #1#)) ((5 6 7 4 0 8 3 2 1) 10000 12 (RIGHT . #1#)
)))
0]
```

Terminated Result

<u>Analysis:</u>



```
* (bfs '(1 3 4 8 6 2 7 0 5))

 NODE-LIST-LENGTH : 36
 TOTAL NODES VISITED : 42
 BFS PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
* (bfs '(2 8 1 0 4 3 7 6 5))

 NODE-LIST-LENGTH : 236
 TOTAL NODES VISITED : 360
 BFS PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
NIL
```

We observe that we get an output for first and second case but for the third case, the algorithm takes more time to return output. This is because, BFS is a type of uninformed search techniques i.e. these algorithms do not know anything about what they are searching and where they should search for it. That's why the name "uninformed" search. Uninformed searching takes a lot of time to search as it doesn't know where to head and where the best chances of finding the element are. The breadth-first search finds an optimal solution (one with the fewest number of possible moves) but does so only after examining a great number of intermediate states. As it traverses the tree "shallowest node first", it would always pick the shallower branch until it reaches the solution (or it runs out of nodes and goes to the next branch) and thus takes a lot of time to reach the goal state.

<u>Complexity Analysis</u>:
Time Complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is
b + b^2 + b^3 + ··· + b^d = **O(b^d).**
If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be **O(b^d+1).**

Space Complexity: Equivalent to how large can the fringe get. For any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search, every node generated remains in memory. There will be **O(b^(d−1))** nodes in the explored set and O(b^d) nodes in the frontier, so the space complexity is **O(b^d),** i.e., it is dominated by the size of the frontier.

Strength and Weakness:

Pros: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists. BFS is optimal as long as the costs of all edges are equal.

1. Used to find the shortest path between vertices
2. Always finds optimal solutions.
3. There is nothing like useless path in BFS, since it searches level by level.
4. Finds the closest goal in less time

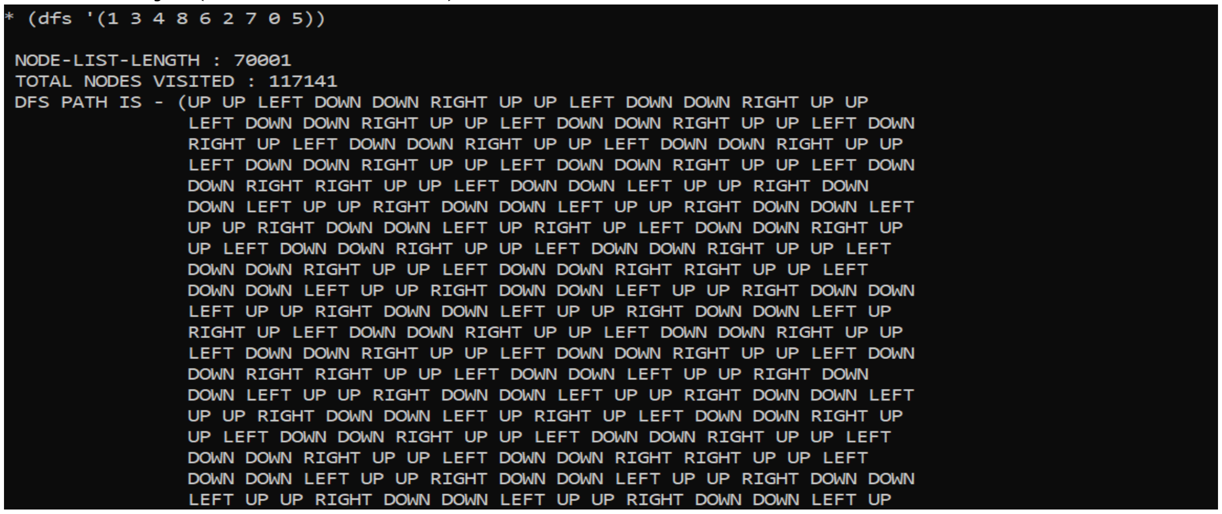
Cons: the memory requirements are a bigger problem for breadth-first search than is the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Also, time taken is a major concern as observed for the hard case in our analysis. If our problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it. In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.
All of the connected vertices must be stored in memory. So consumes more memory

2. DEPTH -FIRST SEARCH

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

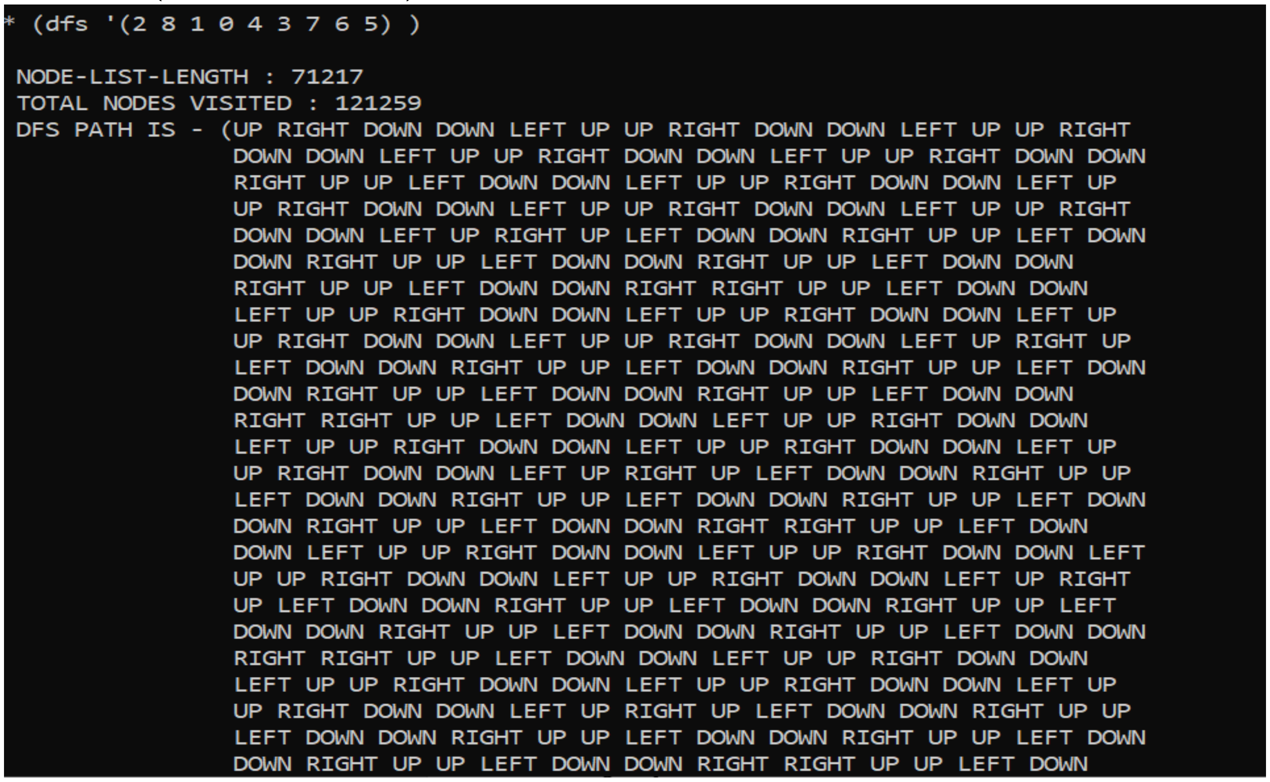
Results:

1. Easy: (1 3 4 8 6 2 7 0 5)



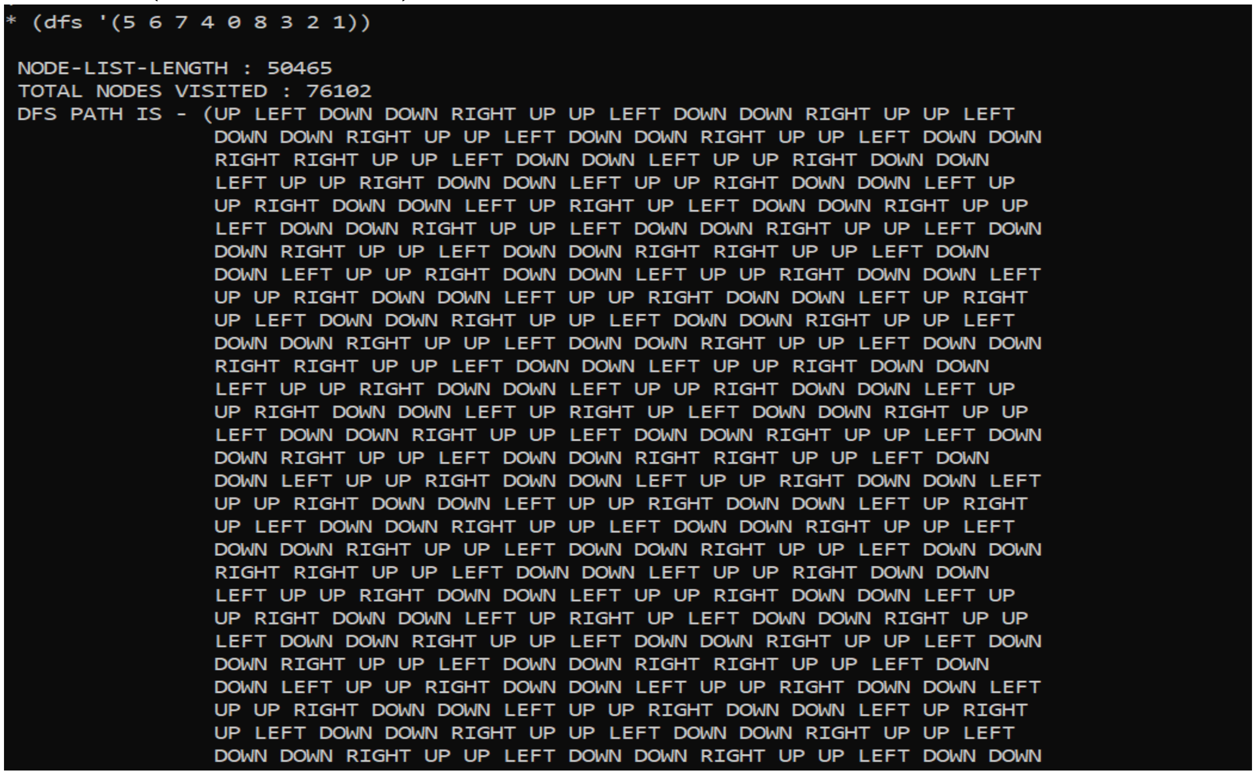
```
* (dfs '(1 3 4 8 6 2 7 0 5))

 NODE-LIST-LENGTH : 70001
 TOTAL NODES VISITED : 117141
 DFS PATH IS - (UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                RIGHT UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT
                DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                RIGHT UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT
                DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
```

2. Medium: (2 8 1 0 4 3 7 6 5)



```
* (dfs '(2 8 1 0 4 3 7 6 5) )

 NODE-LIST-LENGTH : 71217
 TOTAL NODES VISITED : 121259
 DFS PATH IS - (UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT
                DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT
                DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP RIGHT UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP RIGHT
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN
```

3. Hard: (5 6 7 4 0 8 3 2 1)



```
* (dfs '(5 6 7 4 0 8 3 2 1))

 NODE-LIST-LENGTH : 50465
 TOTAL NODES VISITED : 76102
 DFS PATH IS - (UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP RIGHT
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP RIGHT
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
                RIGHT RIGHT UP UP LEFT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN
                LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP
                UP RIGHT DOWN DOWN LEFT UP RIGHT UP LEFT DOWN DOWN RIGHT UP UP
                LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT DOWN
                DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT
                UP UP RIGHT DOWN DOWN LEFT UP UP RIGHT DOWN DOWN LEFT UP RIGHT
                UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT
                DOWN DOWN RIGHT UP UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN DOWN
```

DFS takes more than 10 minutes to return output. The reason why this takes place is because DFS starts exploring each branch in depth starting from root node and explores as far as possible before backtracking. DFS uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected. So, it will start exploring the left subtree first and keeps expanding till the end and thus will take a lot of time to reach the goal state if the goal state is present in the rightmost subtree. It might also go into an infinite loop if duplicate conditions are not handled in the implementation. Thus DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.

Complexity Analysis:

Time Complexity: Equivalent to the number of nodes traversed in DFS. It is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the **O(b^m)** nodes in the search tree, where m is the maximum depth of any node. This can be much greater than the size of the state space. m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

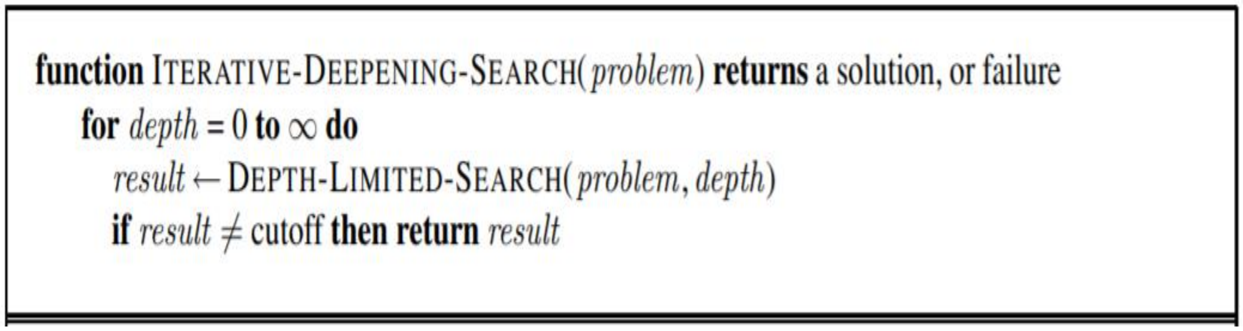
Space Complexity: Equivalent to how large can the fringe get.
DFS will have space complexity advantage over BFS if tree-search is implemented as it will need to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. For a state space with branching factor b and maximum depth m, depth-first search requires storage of only **O(b^m)** nodes.

Pros: DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists. Consumes less memory
Finds the larger distant element (from source vertex) in less time.

Cons: DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high. May not find optimal solution to the problem.
May get trapped in searching useless path.

3. <u>Iterative Deepening Search (IDS)</u>

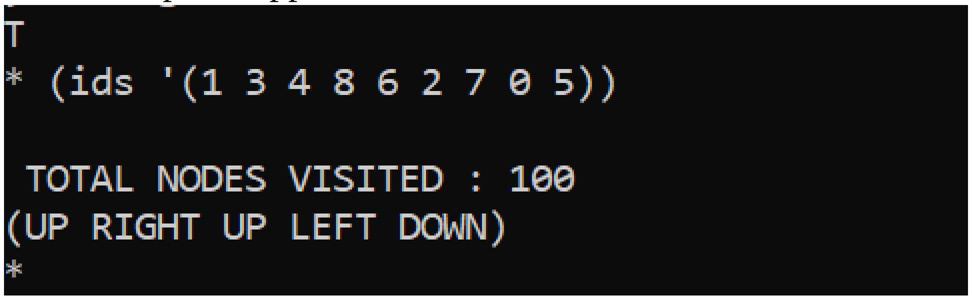
Iterative deepening search or more specifically iterative deepening depth-first search (IDS or IDDFS) is a state space/graph search strategy in which a depth-limited version of depth-first search is run repeatedly with increasing depth limits until the goal is found. IDDFS is optimal like breadth-first search, but uses much less memory; at each iteration, it visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first. It is often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node. Like depth-first search, its memory requirements are modest: O(bd) to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

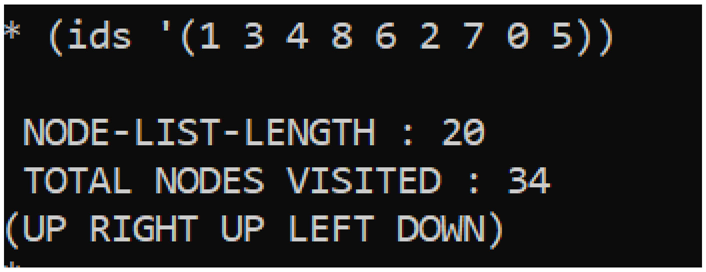
Results:

1. Easy: (1 3 4 8 6 2 7 0 5)
   Output Snippet:



```
T
* (ids '(1 3 4 8 6 2 7 0 5))

 TOTAL NODES VISITED : 100
(UP RIGHT UP LEFT DOWN)
*
```

WITHOUT DUPE IMPLEMENTATION

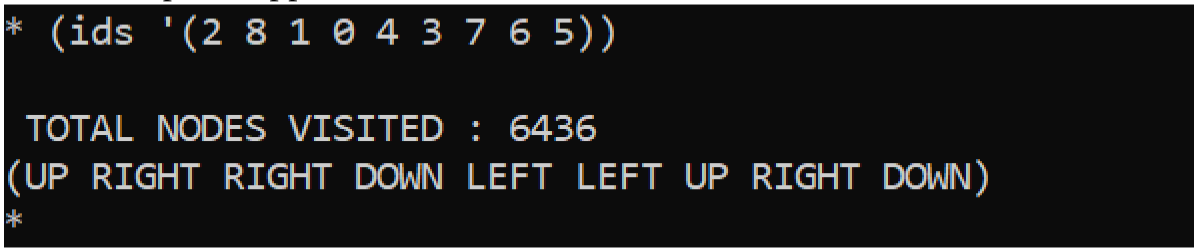


```
* (ids '(1 3 4 8 6 2 7 0 5))

 NODE-LIST-LENGTH : 20
 TOTAL NODES VISITED : 34
(UP RIGHT UP LEFT DOWN)
```

WITH DUPE IMPLEMENTATION

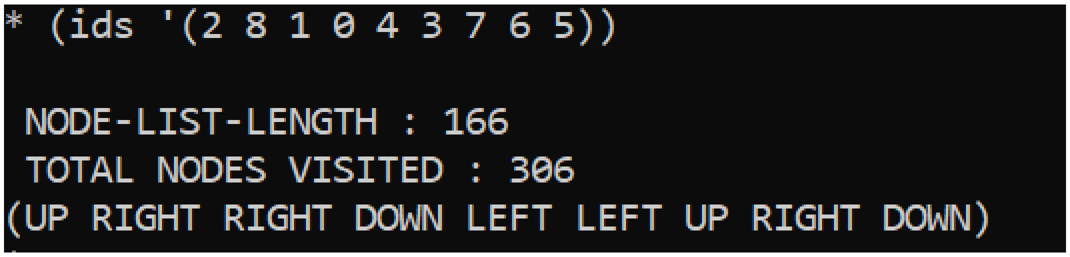
We observe the output sequence of moves to be (UP RIGHT UP LEFT DOWN) and in reaching the goal state it visits a total of 100 nodes before it encounters the node state without checking duplicate and with checking duplicates it is 34. Node list length is 20.

2. Medium: (2 8 1 0 4 3 7 6 5)
   Output Snippet:



```
* (ids '(2 8 1 0 4 3 7 6 5))

 TOTAL NODES VISITED : 6436
(UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
*
```

WITHOUT DUPE IMPLEMENTATION

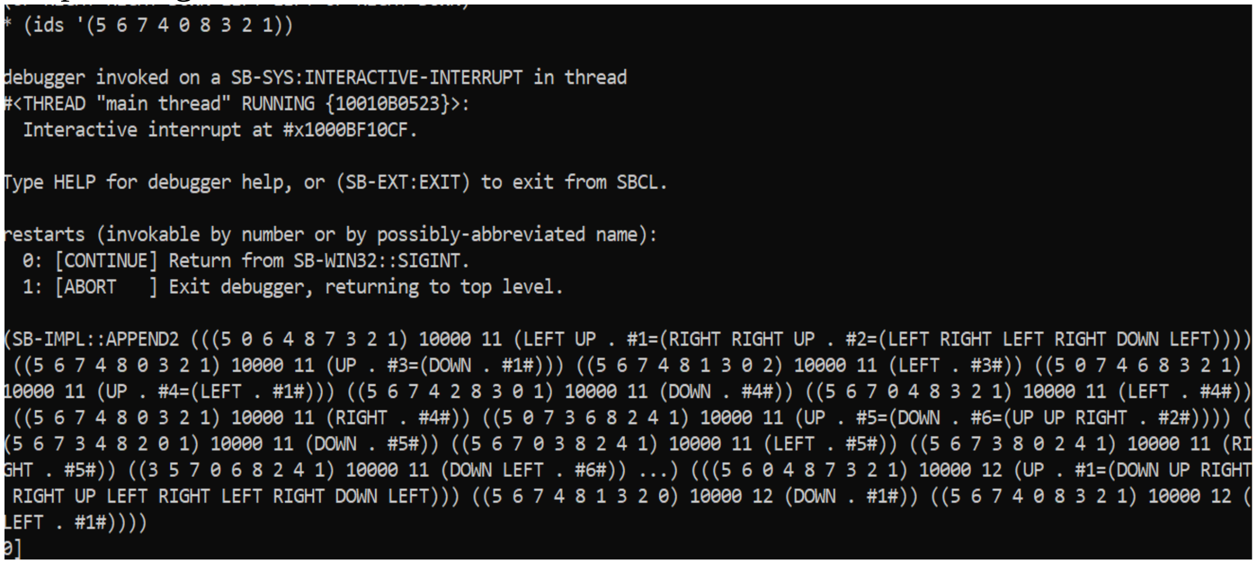


```
* (ids '(2 8 1 0 4 3 7 6 5))

 NODE-LIST-LENGTH : 166
 TOTAL NODES VISITED : 306
(UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
```

WITH DUPE IMPLEMENTATION

We observe the output sequence of moves to be (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN) and in reaching the goal state it visits a total of 6436 nodes before it encounters the node state without checking duplicate and with checking duplicates it is 306. Node list length is 166.

3. Hard: (5 6 7 4 0 8 3 2 1)
   Output: Algorithm ran for more than 10 mins. Had to terminate it.



```
* (ids '(5 6 7 4 0 8 3 2 1))

debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread
#<THREAD "main thread" RUNNING {10010B0523}>:
  Interactive interrupt at #x1000BF10CF.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE] Return from SB-WIN32::SIGINT.
  1: [ABORT   ] Exit debugger, returning to top level.

(SB-IMPL::APPEND2 (((5 0 6 4 8 7 3 2 1) 10000 11 (LEFT UP . #1=(RIGHT RIGHT UP . #2=(LEFT RIGHT LEFT RIGHT DOWN LEFT))))
 ((5 6 7 4 8 0 3 2 1) 10000 11 (UP . #3=(DOWN . #1#))) ((5 6 7 4 8 1 3 0 2) 10000 11 (LEFT . #3#)) ((5 0 7 4 6 8 3 2 1)
10000 11 (UP . #4=(LEFT . #1#))) ((5 6 7 4 2 8 3 0 1) 10000 11 (DOWN . #4#)) ((5 6 7 0 4 8 3 2 1) 10000 11 (LEFT . #4#))
 ((5 6 7 4 8 0 3 2 1) 10000 11 (RIGHT . #4#)) ((5 0 7 3 6 8 2 4 1) 10000 11 (UP . #5=(DOWN . #6=(UP UP RIGHT . #2#)))) (
(5 6 7 3 4 8 2 0 1) 10000 11 (DOWN . #5#)) ((5 6 7 0 3 8 2 4 1) 10000 11 (LEFT . #5#)) ((5 6 7 3 8 0 2 4 1) 10000 11 (RI
GHT . #5#)) ((3 5 7 0 6 8 2 4 1) 10000 11 (DOWN LEFT . #6#)) ...) (((5 6 0 4 8 7 3 2 1) 10000 12 (UP . #1=(DOWN UP RIGHT
 RIGHT UP LEFT RIGHT LEFT RIGHT DOWN LEFT))) ((5 6 7 4 8 1 3 2 0) 10000 12 (DOWN . #1#)) ((5 6 7 4 0 8 3 2 1) 10000 12 (
LEFT . #1#))))
0]
```

Analysis:

We observe that with increasing difficulty of start state, the total number of nodes visited increases with the increase in node-list length. For the hard case, it took more than 10 mins to execute the result, thus terminated the output. Since for each iteration it has to begin parsing from the beginning, time required to reach goal state in case of hard state is very high. Intuitively, this is a dubious idea because each repetition of depth limited DFS will duplicate uselessly all the work done by previous repetitions. But, this useless duplication is not significant because a branching factor $b > 1$ implies that the number of nodes at depth k exactly is much greater than the total number of nodes at ***all*** depths k-1 and less. Implementation of DUPE function reduced the total number of nodes getting traversed while finding the goal state. Since IDS is similar to in behavior to BFS except that it explores nodes level wise, thus it is taking time to return output for hard case is similar to that of BFS. Iterative deepening simulates breadth-first search, but with only linear space complexity.

Time Complexity:

In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is N(IDS)=(d)b + (d − 1)b^2 + ··· + (1)b^d , which gives a time complexity **of O(b^d)**—asymptotically the same as breadth-first search. There is some extra cost for generating the upper levels multiple times, but it is not large.

As we have noticed from the output above, we visit the nodes at *depth = 0* a lot, the nodes at *depth = 2* a little fewer but we visit them multiple times too, and we visit the nodes at *depth = DEPTH_MAX* only once. This may seem inefficient, but it is actually not. This is because, there are very few nodes at *depth = 0*, but a lot of nodes at *depth = DEPTH_MAX*. If '*d*'is depth, and '*b*' is the branching factor in the search tree (this would be N for an N-ary tree), then mathematically –

$$(d)\cdot b+(d-1)\cdot b^2+\ldots+(2)\cdot b^{d-1}+b^d = \sum_{i=0}^{d}(d+1-i)\cdot b^i = O(b^d)$$

The time complexity remains O(b^d) but the constants are large, so IDDFS is slower than BFS and DFS (which also have time complexity of O(b^d)).

Space Complexity:

The space complexity of IDS is **O(d),** where d is the depth of the goal. Since IDDFS, at any point, is engaged in a depth-first search, it need only store a stack of nodes which represents the branch of the tree it is expanding. Since it finds a solution of optimal length, the maximum depth of this stack is d and hence the maximum of space is O(d).

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

4. <u>Greedy Best First Search</u>

In greedy search, we expand the node closest to the goal node, on the grounds that this is likely to lead to a solution quickly. The “closeness” is estimated by a heuristic h(x). The algorithm is called greedy because at each step it tries to get as close to the goal as it can. This is unlike the uninformed search algorithms which blindly explore paths without considering any cost function and thus it falls under the category of Heuristic Search or Informed Search.

Heuristic: A heuristic h is defined as-
h(x) = Estimate of distance of node x from the goal node.
Lower the value of h(x), closer is the node from the goal.

<u>Algorithm:</u>

Using a greedy algorithm, expand the first successor of the parent. After a successor is generated:

1. If the successor's heuristic is better than its parent, the successor is set at the front of the queue (with the parent reinserted directly behind it), and the loop restarts.
2. Else, the successor is inserted into the queue (in a location determined by its heuristic value). The procedure will evaluate the remaining successors (if any) of the parent.

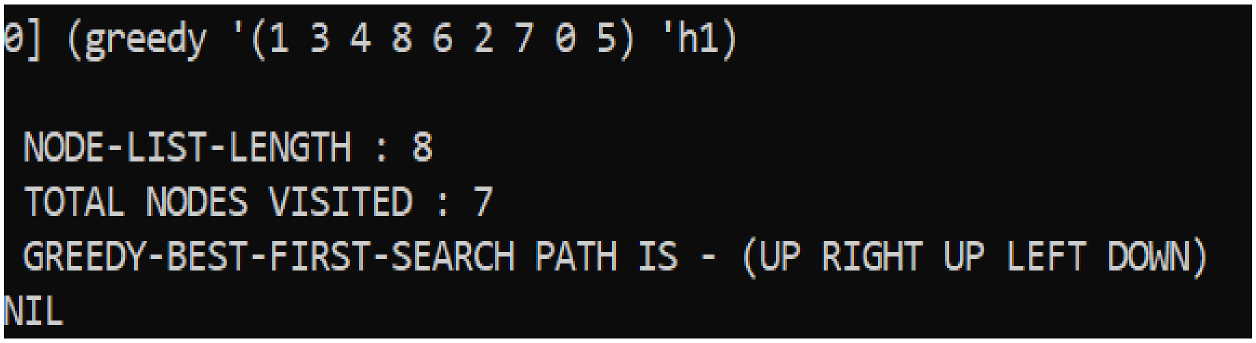
In the current implementation, I have implemented 2 different heuristic functions for informed search algorithms and tested results for all three cases using both heuristics.

In my implementation, H1 heuristic calculates Number of tiles out of place in the 8-puzzle start state.H2 heuristic implementation calculates the Manhattan distance (sum of the distances of each tile from its goal position)

Results:

1. Easy: (1 3 4 8 6 2 7 0 5)
   Output Snippet:

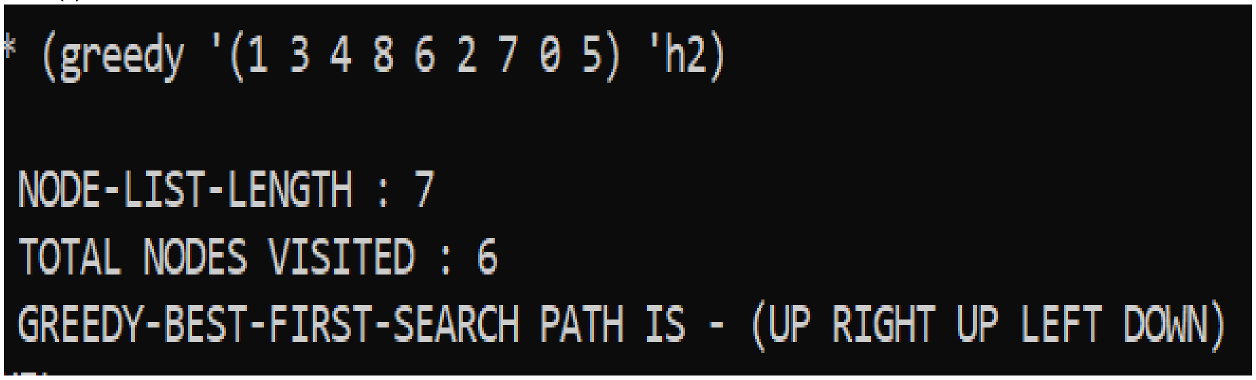
(i) H1 heuristic



```
0] (greedy '(1 3 4 8 6 2 7 0 5) 'h1)

 NODE-LIST-LENGTH : 8
 TOTAL NODES VISITED : 7
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
```

We observe the output sequence of moves to be (UP RIGHT UP LEFT DOWN) and in reaching the goal state it visits a total of 7 nodes before it encounters the node state. Node-list length is 8.

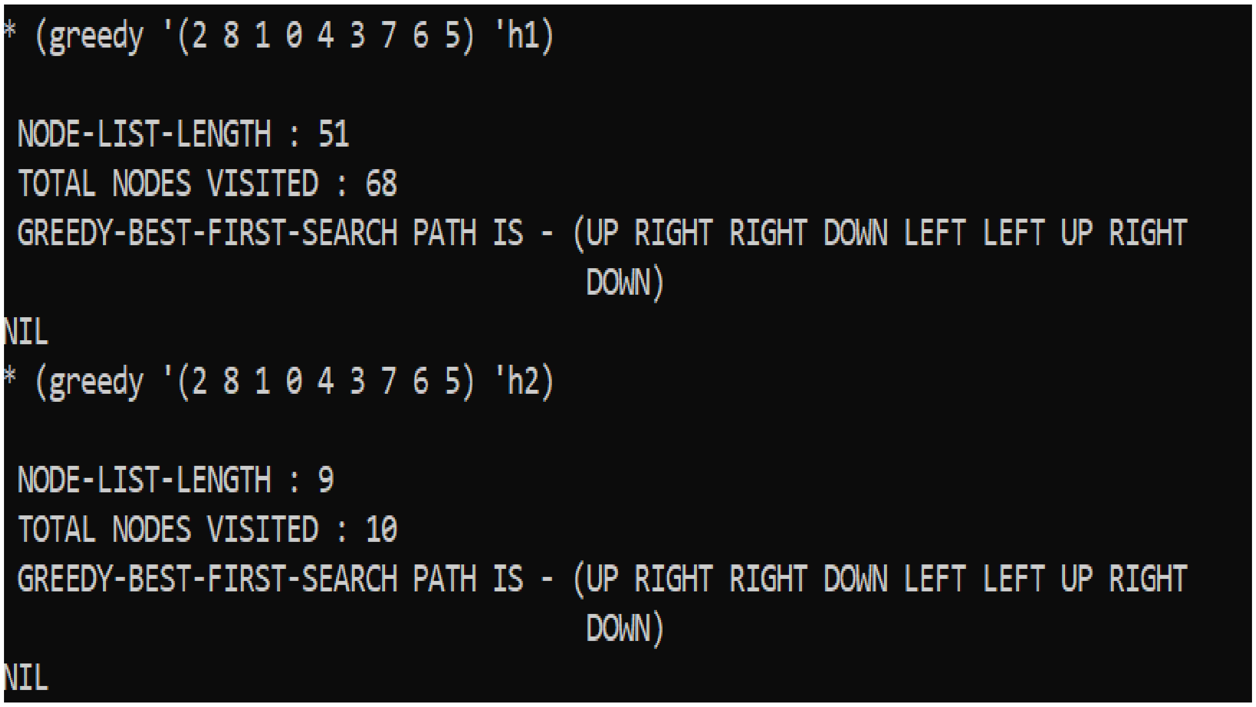
(i) H2 heuristic



```
* (greedy '(1 3 4 8 6 2 7 0 5) 'h2)

 NODE-LIST-LENGTH : 7
 TOTAL NODES VISITED : 6
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP RIGHT UP LEFT DOWN)
```

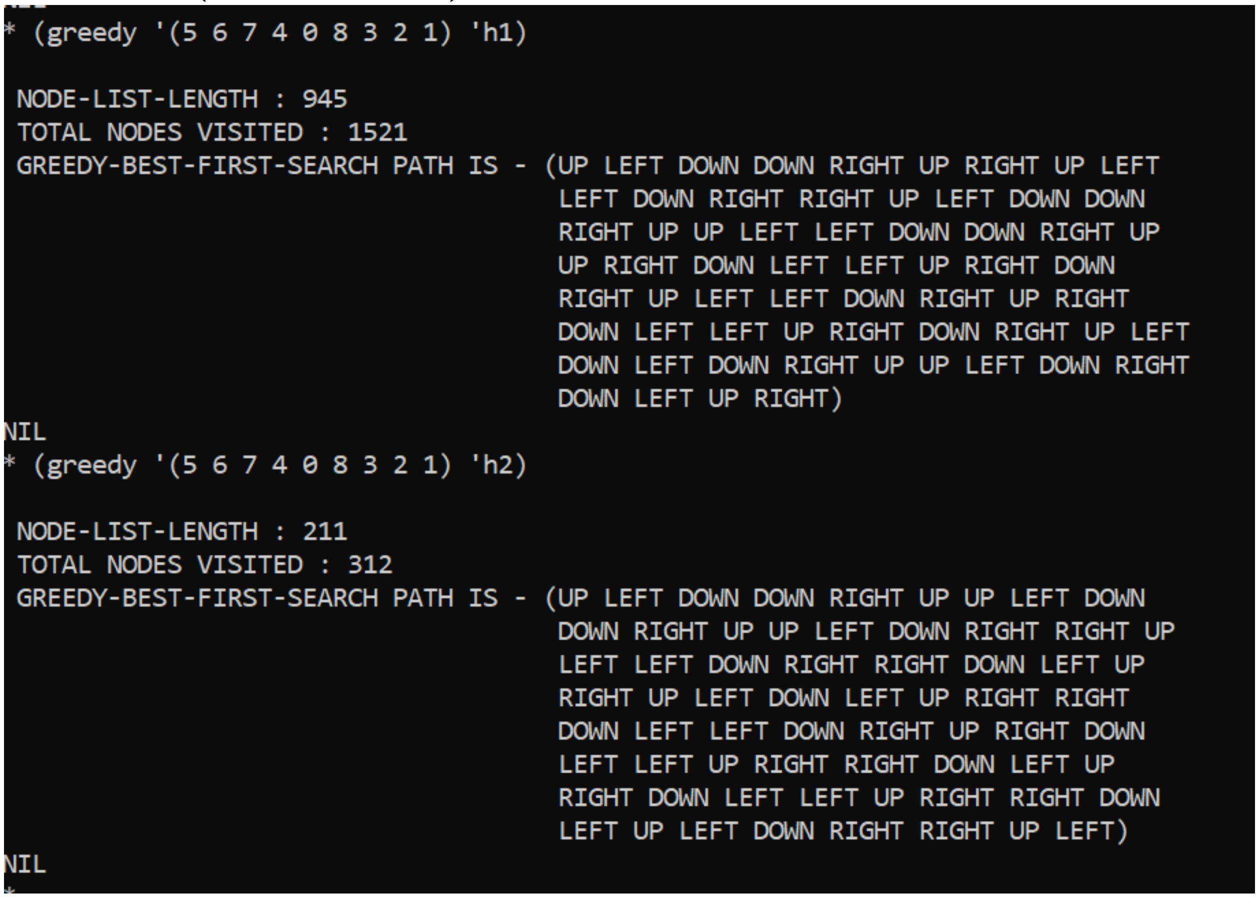
We observe the output sequence of moves to be (UP RIGHT UP LEFT DOWN) and in reaching the goal state it visits a total of 5 nodes before it encounters the node state. Node-list length is 7.

2. Medium: (2 8 1 0 4 3 7 6 5)
   Output Snippet:



```
* (greedy '(2 8 1 0 4 3 7 6 5) 'h1)

 NODE-LIST-LENGTH : 51
 TOTAL NODES VISITED : 68
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT
                                     DOWN)
NIL
* (greedy '(2 8 1 0 4 3 7 6 5) 'h2)

 NODE-LIST-LENGTH : 9
 TOTAL NODES VISITED : 10
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT
                                     DOWN)
NIL
```

3. Hard: (5 6 7 4 0 8 3 2 1)



```
* (greedy '(5 6 7 4 0 8 3 2 1) 'h1)

 NODE-LIST-LENGTH : 945
 TOTAL NODES VISITED : 1521
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP LEFT DOWN DOWN RIGHT UP RIGHT UP LEFT
                                     LEFT DOWN RIGHT RIGHT UP LEFT DOWN DOWN
                                     RIGHT UP UP LEFT LEFT DOWN DOWN RIGHT UP
                                     UP RIGHT DOWN LEFT LEFT UP RIGHT DOWN
                                     RIGHT UP LEFT LEFT DOWN RIGHT UP RIGHT
                                     DOWN LEFT LEFT UP RIGHT DOWN RIGHT UP LEFT
                                     DOWN LEFT DOWN RIGHT UP UP LEFT DOWN RIGHT
                                     DOWN LEFT UP RIGHT)
NIL
* (greedy '(5 6 7 4 0 8 3 2 1) 'h2)

 NODE-LIST-LENGTH : 211
 TOTAL NODES VISITED : 312
 GREEDY-BEST-FIRST-SEARCH PATH IS - (UP LEFT DOWN DOWN RIGHT UP UP LEFT DOWN
                                     DOWN RIGHT UP UP LEFT DOWN RIGHT RIGHT UP
                                     LEFT LEFT DOWN RIGHT RIGHT DOWN LEFT UP
                                     RIGHT UP LEFT DOWN LEFT UP RIGHT RIGHT
                                     DOWN LEFT LEFT DOWN RIGHT UP RIGHT DOWN
                                     LEFT LEFT UP RIGHT RIGHT DOWN LEFT UP
                                     RIGHT DOWN LEFT LEFT UP RIGHT RIGHT DOWN
                                     LEFT UP LEFT DOWN RIGHT RIGHT UP LEFT)
NIL
```

Analysis:

We observe that the total nodes visited increases in number with the difficulty of the start state. Also increases the node-list length with that. We observe that search path iterated through for all 3 cases are similar for both heuristics, but the total number of nodes visited varies with Manhattan distance h2 heuristic having lesser number of nodes visited than h1 number of tiles misplaced heuristic. This is because h1 only takes into account whether a tile is misplaced or not, but it doesn't take into account how far away that tile is from being correct: a tile that is 1 square away from its ultimate destination is treated the same as a tile that is far away from where it belongs.
In contrast, h2 does take this information into account. Instead of treating each tile as either "correct" or "incorrect" (a binary decision), h2 introduces shades of grey that take into account how far the tile is from where it belongs.

Time Complexity:

In general, the time complexity is **O(b^m)**, where b is the (maximum) branching factor and m is the maximum depth of the search tree.

Space Complexity: The space complexity is proportional to the number of nodes in the fringe and to the length of the found path .i.e can be polynomial.

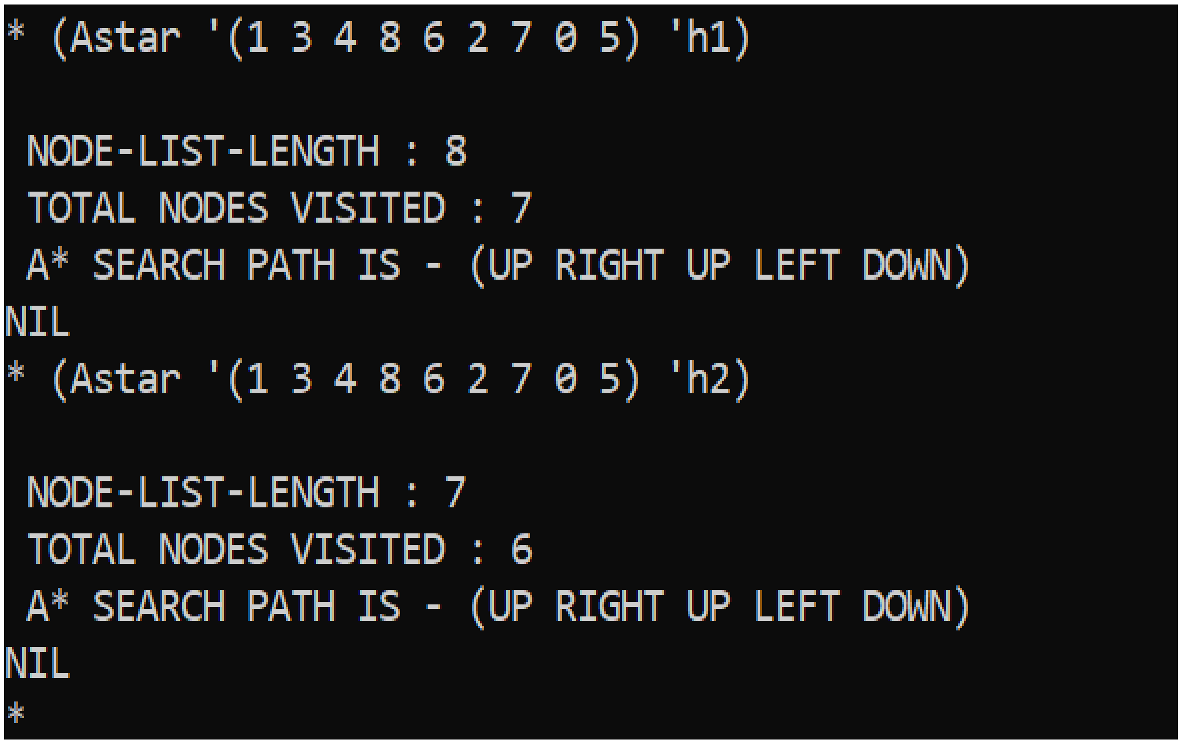
In the case of *the greedy BFS algorithm*, the evaluation function is f(n)=h(n)f(n)=h(n), that is, the greedy BFS algorithm first expands the node whose estimated distance to the goal is the smallest. So, greedy BFS does not use the "past knowledge", i.e. g(n)g(n). Hence its connotation "greedy". In general, the greedy BST algorithm is **not complete**, that is, there is always the risk to take a path that does not bring to the goal. In the greedy BFS algorithm, all nodes on the *border* (or fringe or frontier) are kept in memory, and nodes that have already been expanded do not need to be stored in memory and can therefore be discarded. In general, the greedy BFS is also **not optimal**, that is, the path found may not be the optimal one.

5. A* Algorithm

In the case of *the A* algorithm*, the evaluation function is f(n)=g(n)+h(n) where h is an admissible heuristic function. The "star", often denoted by an asterisk, *, refers to the fact that A* uses an admissible heuristic function, which essentially means that A* is **optimal**, that is, it always finds the optimal path between the starting node and the goal node. A* is also **complete** (unless there are infinitely many nodes to explore in the search space).

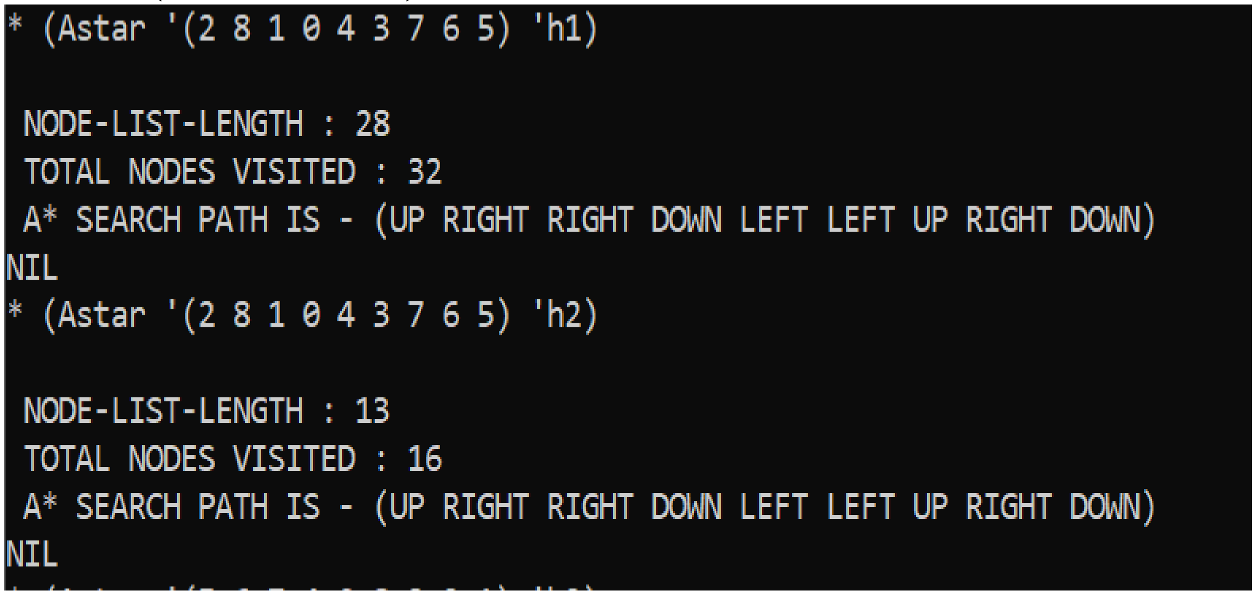
Results

1. Easy: (1 3 4 8 6 2 7 0 5)



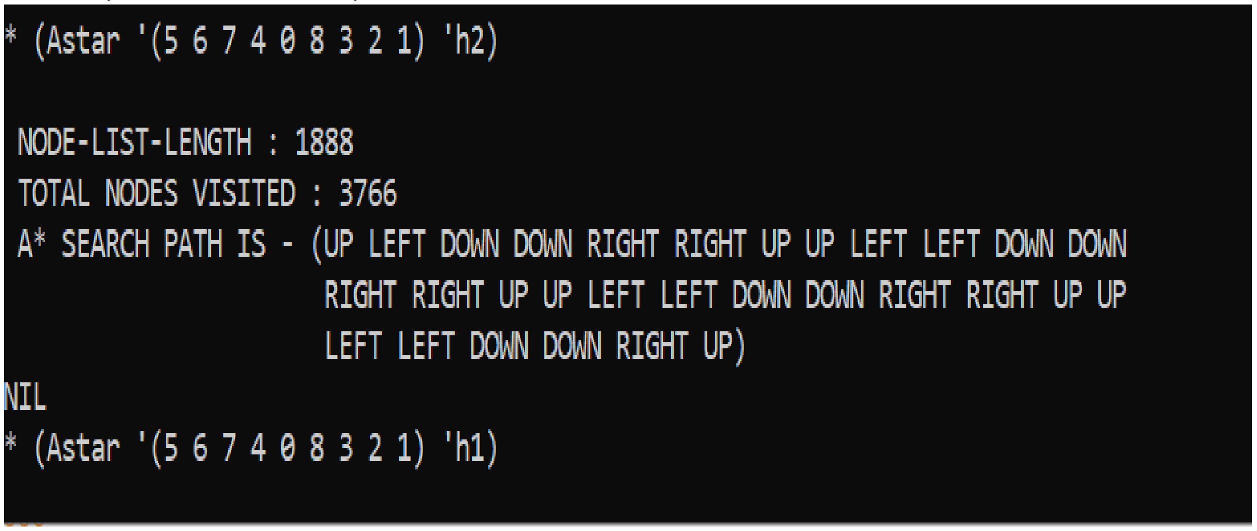
```
* (Astar '(1 3 4 8 6 2 7 0 5) 'h1)

 NODE-LIST-LENGTH : 8
 TOTAL NODES VISITED : 7
 A* SEARCH PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
* (Astar '(1 3 4 8 6 2 7 0 5) 'h2)

 NODE-LIST-LENGTH : 7
 TOTAL NODES VISITED : 6
 A* SEARCH PATH IS - (UP RIGHT UP LEFT DOWN)
NIL
*
```

2. Medium: (2 8 1 0 4 3 7 6 5)



```
* (Astar '(2 8 1 0 4 3 7 6 5) 'h1)

 NODE-LIST-LENGTH : 28
 TOTAL NODES VISITED : 32
 A* SEARCH PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
NIL
* (Astar '(2 8 1 0 4 3 7 6 5) 'h2)

 NODE-LIST-LENGTH : 13
 TOTAL NODES VISITED : 16
 A* SEARCH PATH IS - (UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
NIL
```

3. Hard: (5 6 7 4 0 8 3 2 1)



```
* (Astar '(5 6 7 4 0 8 3 2 1) 'h2)

 NODE-LIST-LENGTH : 1888
 TOTAL NODES VISITED : 3766
 A* SEARCH PATH IS - (UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT LEFT DOWN DOWN
                      RIGHT RIGHT UP UP LEFT LEFT DOWN DOWN RIGHT RIGHT UP UP
                      LEFT LEFT DOWN DOWN RIGHT UP)
NIL
* (Astar '(5 6 7 4 0 8 3 2 1) 'h1)
```

2

Analysis:

We observe that search path iterated through for all 3 cases are similar for both heuristics, but the total number of nodes visited varies with Manhattan distance h2 heuristic having lesser number of nodes visited than h1 number of tiles misplaced heuristic. This is because h1 only takes into account whether a tile is misplaced or not, but it doesn't take into account how far away that tile is from being correct: a tile that is 1 square away from its ultimate destination is treated the same as a tile that is far away from where it belongs.
In contrast, h2 does take this information into account. Instead of treating each tile as either "correct" or "incorrect" (a binary decision), h2 introduces shades of grey that consider how far the tile is from where it belongs. A* Algorithm for hard case calculating heuristic function using count of misplaced tiles take very long to execute and thus terminated the output after 10 mins

Time Complexity:
The time complexity is **O(b^m).** However, A* needs to keep all nodes in memory while searching, not just the ones in the fringe, because A*, essentially, performs an "exhaustive search" (which is "informed", in the sense that it uses a heuristic function)

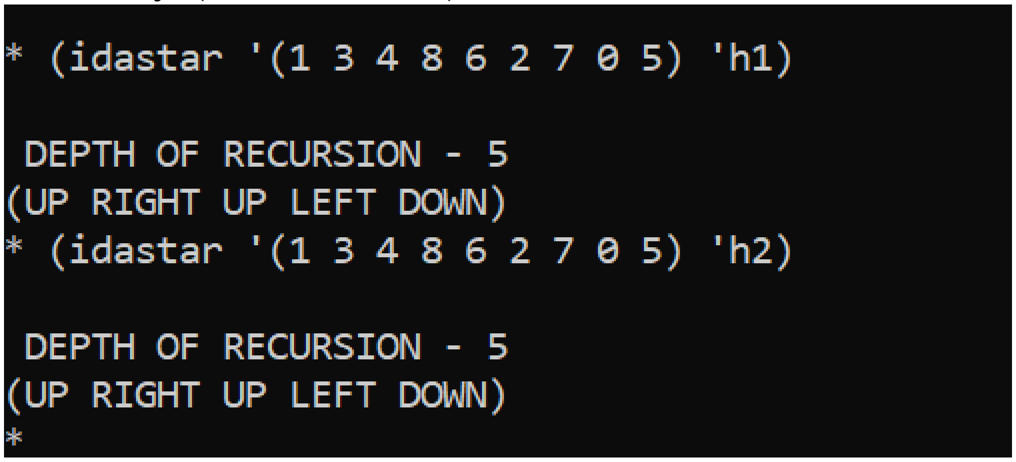
A* is complete, optimal, and it has a time and **space complexity** of **O(b^m)**. So, in general, A* uses more memory than greedy BFS. A* becomes impractical when the search space is huge. However, A* also guarantees that the found path between the starting node and the goal node is the optimal one and that the algorithm eventually terminates

4. **IDA* Algorithm**

Iterative deepening A* (IDA*) is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph. It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm. Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus does not go to the same depth everywhere in the search tree. Unlike A*, IDA* does not utilize dynamic programming and therefore often ends up exploring the same nodes many times.

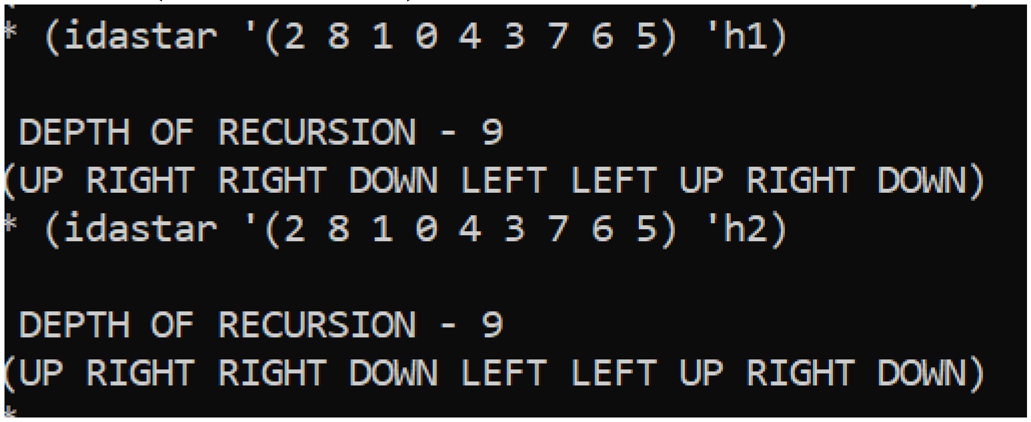
Results:

1 Easy: (1 3 4 8 6 2 7 0 5)



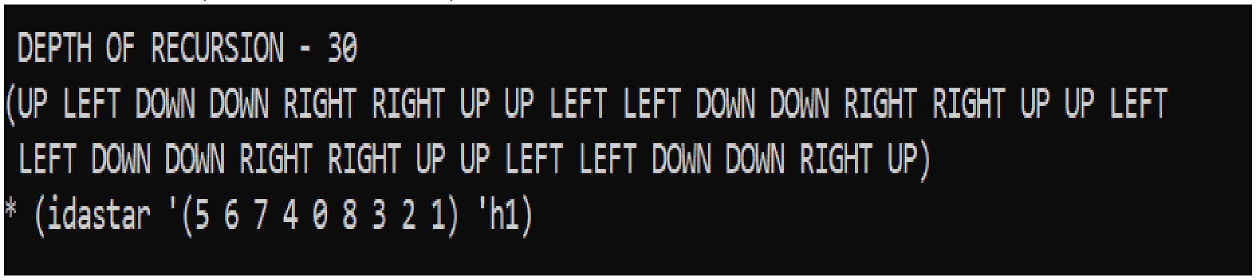
```
* (idastar '(1 3 4 8 6 2 7 0 5) 'h1)

 DEPTH OF RECURSION - 5
(UP RIGHT UP LEFT DOWN)
* (idastar '(1 3 4 8 6 2 7 0 5) 'h2)

 DEPTH OF RECURSION - 5
(UP RIGHT UP LEFT DOWN)
*
```

2 Medium: (2 8 1 0 4 3 7 6 5)



```
* (idastar '(2 8 1 0 4 3 7 6 5) 'h1)

 DEPTH OF RECURSION - 9
(UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
* (idastar '(2 8 1 0 4 3 7 6 5) 'h2)

 DEPTH OF RECURSION - 9
(UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN)
*
```

3 Hard: (5 6 7 4 0 8 3 2 1)



```
 DEPTH OF RECURSION - 30
(UP LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT
 LEFT DOWN DOWN RIGHT RIGHT UP UP LEFT LEFT DOWN DOWN RIGHT UP)
* (idastar '(5 6 7 4 0 8 3 2 1) 'h1)
```

Analysis:

As observed from the above results, we can see that for easy and medium cases, it finds the goal state at a depth of 5 and 9 respectively for both heuristic calculations whereas for Hard case Manhattan Distance heuristic, it finds the goal state at a depth of 30. Hard Case Missed tiles takes too long to return output, thus had to terminate it. This happens because H1 is not as efficient a heuristic function as h2 and thus 8-Puzzle takes time in processing output with H1 as input as compared to H2 as input. This is because h1 only takes into account whether a tile is misplaced or not, but it doesn't take into account how far away that tile is from being correct: a tile that is 1 square away from its ultimate destination is treated the same as a tile that is far away from where it belongs. In contrast, h2 does take this information into account. Instead of treating each tile as either "correct" or "incorrect" (a binary decision), h2 introduces shades of grey that consider how far the tile is from where it belongs.

Time Complexity:

IDA∗ uses the cost function f (n) = g(n) + h(n), where g(n) is the sum of the edge costs from the initial state to node n, and h(n) is an estimate of the cost of reaching a goal from node n. Each iteration is a depth-first search where a branch is pruned when it reaches a node whose total cost exceeds the cost threshold of that iteration. The cost threshold for the first iteration is the heuristic value of the initial state, and increases in each iteration to the lowest cost of all nodes pruned on the previous iteration. It continues until a goal node is found whose cost does not exceed the current cost threshold.

The running time of IDA∗ depends on the branching factor, the heuristic distribution, and the optimal solution cost. Since the number of nodes in a problem-space tree grows by a factor of b with each succeeding depth, a lower bound on the maximum optimal solution depth is the log base b of the number of reachable states, rounded up to the next larger integer.:

Time complexity: O(b^d)

Space complexity: O(d)

Where
b: branching factor
d: depth of first solution