Artificial Intelligence -Assignment 2
REPORT

jump_1.cpp implements **A* Algorithm** , **jump_2.cpp** implements **IDA* Algorithm**

State Representation: A matrix (NxN) with the same format as the input file: one entry for every space on the board, with zeros for empty spaces and a different number for each color.

Heuristic (**h**) used for Search (both **A*** and **IDA***):

- Given a state, **h**(*state*) = *(*#distinct colours in the grid corresponding to *state)* - 1
- It is obvious that at-least **h**(state) number of moves are required to finally leave 1 color on the grid. Hence an underestimate of the actual **h***(state)

Memory bounded A* algorithm used : Iterative Deepening A* (IDA*)

IDA_star function iteratively calls the search function with increasing ***bound*** values until a goal state is found, or no such state can be found.

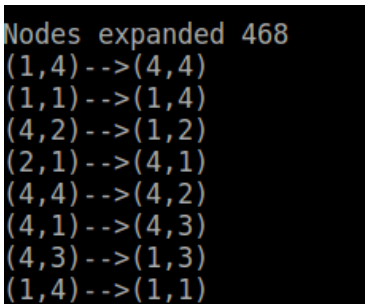
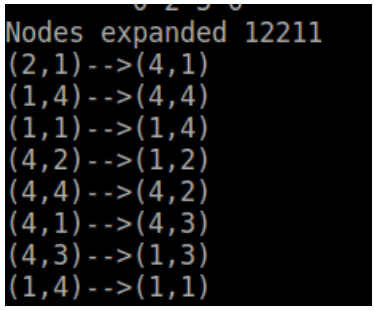
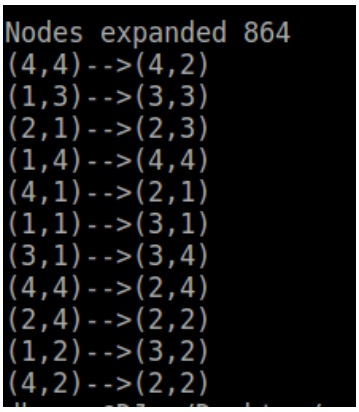
Search function implements a depth-first search algorithm until the **f** value of a node/state does not exceed the value of ***bound***, in which case the function stops expanding that node and returns back to the remaining nodes (if any, else it exits like DFS)

```
node              current node
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  path cost function
is_goal(node)     goal test
successors(node)  node expanding function

procedure ida_star(root)
  bound := h(root)
  loop
    t := search(root, 0, bound)
    if t = FOUND then return FOUND
    if t = ∞ then return NOT_FOUND
    bound := t
  end loop
end procedure

function search(node, g, bound)
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min := ∞
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

Results

Input	***A* algorithm***	***IDA* algorithm***
	#nodes expanded	
4 *3* *1 2 2 1* *2 1 3 2* *3 1 3 2* *0 2 3 0*	*468* 	*12211* 
4 *3* *3 3 2 1* *3 1 3 2* *3 2 0 2* *1 0 3 1*	*864* 	*65476*