AI LAB EXP 1

Sheel Patel – RA1911003010439

CAMEL AND BANANA PROBLEM

Aim- **To find the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed)**

Procedure- **We have a total of 3000 bananas.**

The destination is 1000KMs

Only 1 mode of transport.

Camel can carry a maximum of 1000 banana at a time.

Camel eats a banana every km it travels.

With all these points, we can say that person won't we able to transfer any banana to the destination as the camel is going to eat all the banana on its way to the destination.

But the trick here is to have intermediate drop points, then, the camel can make several short trips in between.

Also, we try to maintain the number of bananas at each point to be multiple of 1000.

Code-

```python
total=int(input('Enter no. of bananas at starting '))
distance=int(input('Enter distance you want to cover '))
capacity=int(input('Enter max load capacity of your camel '))
n=0
start=total
for i in range(distance):
    while start>0:
        start=start-capacity
        if start==1:
            n=n-1
        n=n+2
    n=n-1
    start=total-n
    if start==0:
        break
print(start)
```

Output-

```
Rmanjula:~/environment/RA1911003010439/Exp1 $ python Camel.py
Enter no. of bananas at starting 5000
Enter distance you want to cover 100
Enter max load capacity of your camel 100
665
Rmanjula:~/environment/RA1911003010439/Exp1 $
```

CANNIBAL AND MISSIONARY PROBLEM

Aim- How can everyone get across the river without the missionaries risking being eaten?

Procedure- 1. Number of cannibals should lesser than the missionaries on either side.

2. Only one boat is available to travel.

3. Only one or maximum of two people can go in the boat at a time.

4. All the six have to cross the river from bank.

5. There is no restriction on the number of trips that can be made to reach of the goal.

6. Both the missionaries and cannibals can row the boat.

Code-

```
from copy import deepcopy
from collections import deque
import sys
import time

class State(object):
  def __init__(self, missionaries, cannibals, boats):
    self.missionaries = missionaries
    self.cannibals = cannibals
    self.boats = boats

  def successors(self):
    if self.boats == 1:
      sgn = -1
      direction = "from the original shore to the new shore"
    else:
      sgn = 1
      direction = "back from the new shore to the original shore"
    for m in range(3):
      for c in range(3):
        newState = State(self.missionaries+sgn*m, self.cannibals+sgn*c, self.boats+sgn*1);
        if m+c >= 1 and m+c <= 2 and newState.isValid():
          action = "take %d missionaries and %d cannibals %s. %r" % ( m, c, direction, newState)
          yield action, newState

  def isValid(self):
    if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibals > 3 or
(self.boats != 0 and self.boats != 1):
      return False

    if self.cannibals > self.missionaries and self.missionaries > 0:
      return False
    if self.cannibals < self.missionaries and self.missionaries < 3:
      return False
    return True

  def is_goal_state(self):
    return self.cannibals == 0 and self.missionaries == 0 and self.boats == 0

  def __repr__(self):
    return "< State (%d, %d, %d) >" % (self.missionaries, self.cannibals, self.boats)



class Node(object):
  def __init__(self, parent_node, state, action, depth):
    self.parent_node = parent_node
    self.state = state
    self.action = action
```

```
    self.depth = depth

  def expand(self):
    for (action, succ_state) in self.state.successors():
      succ_node = Node(
                  parent_node=self,
                  state=succ_state,
                  action=action,
                  depth=self.depth + 1)
      yield succ_node

  def extract_solution(self):
    solution = []
    node = self
    while node.parent_node is not None:
      solution.append(node.action)
      node = node.parent_node
    solution.reverse()
    return solution


def breadth_first_tree_search(initial_state):
  initial_node = Node(
                 parent_node=None,
                 state=initial_state,
                 action=None,
                 depth=0)
  fifo = deque([initial_node])
  num_expansions = 0
  max_depth = -1
  while True:
    if not fifo:
      print("%d expansions" % num_expansions)
      return None
    node = fifo.popleft()
    if node.depth > max_depth:
      max_depth = node.depth
      print("[depth = %d] %.2fs" % (max_depth, time.clock()))
    if node.state.is_goal_state():
      print("%d expansions" % num_expansions)
      solution = node.extract_solution()
      return solution
    num_expansions += 1
    fifo.extend(node.expand())


def usage():
  print >> sys.stderr, "usage:"
  print >> sys.stderr, "    %s" % sys.argv[0]
  raise SystemExit(2)
```

```
def main():
  initial_state = State(3,3,1)
  solution = breadth_first_tree_search(initial_state)
  if solution is None:
    print("no solution")
  else:
    print("solution (%d steps):" % len(solution))
    for step in solution:
      print("%s" % step)
  print("elapsed time: %.2fs" % time.clock())


if __name__ == "__main__":
  main()
```

Output-

```
Rmanjula:~/environment/RA1911003010439/Exp1 $ python missionary.py
[depth = 0] 0.05s
[depth = 1] 0.05s
[depth = 2] 0.05s
[depth = 3] 0.05s
[depth = 4] 0.06s
[depth = 5] 0.06s
[depth = 6] 0.06s
[depth = 7] 0.08s
[depth = 8] 0.10s
[depth = 9] 0.17s
[depth = 10] 0.28s
[depth = 11] 0.36s
10963 expansions
solution (11 steps):
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (3, 1, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (3, 2, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (3, 0, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (3, 1, 1) >
take 2 missionaries and 0 cannibals from the original shore to the new shore. < State (1, 1, 0) >
take 1 missionaries and 1 cannibals back from the new shore to the original shore. < State (2, 2, 1) >
take 2 missionaries and 0 cannibals from the original shore to the new shore. < State (0, 2, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (0, 3, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (0, 1, 0) >
take 0 missionaries and 1 cannibals back from the new shore to the original shore. < State (0, 2, 1) >
take 0 missionaries and 2 cannibals from the original shore to the new shore. < State (0, 0, 0) >
elapsed time: 0.43s
```

WATER JUG PROBLEM

Sheel Patel
RA1911003010439

Aim- **Given two jugs with the maximum capacity of m and n liters respectively. The jugs don't have markings on them which can help us to measure smaller quantities. The task is to measure d liters of water using these two jugs. Hence our goal is to reach from initial state (m, n) to final state (0, d) or (d, 0)**

Code-

```python
from collections import defaultdict
jug1, jug2, aim = 4, 3, 2
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))
    else:
        return False

print("Steps: ")
waterJugSolver(0, 0)
```

Output-

```
bash - "ip-172-31-12-168' ×
Rmanjula:~/environment $ cd RA1911003010439
Rmanjula:~/environment/RA1911003010439 $ cd Exp2
Rmanjula:~/environment/RA1911003010439/Exp2 $ python Jug.py
Steps:
(0, 0)
(4, 0)
(4, 3)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
(0, 2)
Rmanjula:~/environment/RA1911003010439/Exp2 $
```

RESULT- Thus the water jug problem was successfully done.

Edge Coloring of a graph

Sheel Patel – RA1911003010439

Aim- An edge coloring of a graph is an assignment of "colors" to the edges of the graph so that no two incident edges have the same color.

Procedure-

1.Use BFS traversal to start traversing the graph.
2. Pick any vertex and give different colors to all of the edges connected to it, and mark those edges as colored.
3. Traverse one of it's edges. Repeat step to with a new vertex until all edges are colored.

Code-

```
class Graph:

    def __init__(self, edges, N):
        self.adj = [[] for _ in range(N)]
        for (src, dest) in edges:
            self.adj[src].append(dest)
            self.adj[dest].append(src)

def colorGraph(graph):
    result = {}
    for u in range(N):
        assigned = set([result.get(i) for i in graph.adj[u] if i in result])
        color = 1
        for c in assigned:
            if color != c:
                break
            color = color + 1
        result[u] = color
    for v in range(N):
        print("Color assigned to vertex", v, "is", colors[result[v]])
```

```
if __name__ == '__main__':
    colors = ["", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK",
              "BLACK", "BROWN", "WHITE", "PURPLE", "VIOLET"]
    edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]
    N = 6
    graph = Graph(edges, N)
    colorGraph(graph)
```

Output-

```
RA1911003010437:~/environment/RA1911003010439/Exp2 $ python edgecolor.py
('Color assigned to vertex', 0, 'is', 'BLUE')
('Color assigned to vertex', 1, 'is', 'GREEN')
('Color assigned to vertex', 2, 'is', 'BLUE')
('Color assigned to vertex', 3, 'is', 'RED')
('Color assigned to vertex', 4, 'is', 'RED')
('Color assigned to vertex', 5, 'is', 'GREEN')
```

Result- Thus the edge coloring problem was successfully solved and implemented.

EXPERIMENT 3

Sheel Patel – RA1911003010439

Aim- To solve the Cryptarithmetic problem of the CSP.

Algorithm-

```
Begin
  if n letters are assigned, then
    for all digits i from 0 to 9, do
      if digit i is not used, then
        nodeList[n].value := i
        if isValid(nodeList, count, word1, word2, word3) = true
          for all items j in the nodeList, do
            show the letter and corresponding values.
          done
          return true
    done
    return false

  for all digits i from 0 to 9, do
    if digit i is not used, then
      nodeList[n].value := i
      mark as i is used
      if permutation(nodeList, count, n+1, word1, word2, word3),
        return true
      otherwise mark i as not used
  done
  return false
End
```

Code-

```
import itertools
def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor * substitution[letter]
        factor *= 10
    return s

def solve2(equation):
    # split equation in left and right
    left, right = equation.lower().replace(' ', '').split('=')
    # split words in left part
    left = left.split('+')
    # create list of used letters
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)
    digits = range(10)
    for perm in itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))

        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
            print(' + '.join(str(get_value(word, sol)) for word in left) + " = {} (mapping: {})".format(get_value(right, sol), sol))
```

```
if __name__ == '__main__':
    solve2('SEND + MORE = MONEY')
```

OUTPUT-

```
Run        Command: RA1911003010439/CSP.py

9567 + 1085 = 10652 (mapping: {'o': 0, 'm': 1, 'e': 5, 'y': 2, 's': 9, 'n': 6, 'd': 7, 'r': 8})
2817 + 368 = 3185 (mapping: {'o': 3, 'm': 0, 'e': 8, 'y': 5, 's': 2, 'n': 1, 'd': 7, 'r': 6})
2819 + 368 = 3187 (mapping: {'o': 3, 'm': 0, 'e': 8, 'y': 7, 's': 2, 'n': 1, 'd': 9, 'r': 6})
3719 + 457 = 4176 (mapping: {'o': 4, 'm': 0, 'e': 7, 'y': 6, 's': 3, 'n': 1, 'd': 9, 'r': 5})
3712 + 467 = 4179 (mapping: {'o': 4, 'm': 0, 'e': 7, 'y': 9, 's': 3, 'n': 1, 'd': 2, 'r': 6})
3829 + 458 = 4287 (mapping: {'o': 4, 'm': 0, 'e': 8, 'y': 7, 's': 3, 'n': 2, 'd': 9, 'r': 5})
3821 + 468 = 4289 (mapping: {'o': 4, 'm': 0, 'e': 8, 'y': 9, 's': 3, 'n': 2, 'd': 1, 'r': 6})
5731 + 647 = 6378 (mapping: {'o': 6, 'm': 0, 'e': 7, 'y': 8, 's': 5, 'n': 3, 'd': 1, 'r': 4})
5732 + 647 = 6379 (mapping: {'o': 6, 'm': 0, 'e': 7, 'y': 9, 's': 5, 'n': 3, 'd': 2, 'r': 4})
5849 + 638 = 6487 (mapping: {'o': 6, 'm': 0, 'e': 8, 'y': 7, 's': 5, 'n': 4, 'd': 9, 'r': 3})
6419 + 724 = 7143 (mapping: {'o': 7, 'm': 0, 'e': 4, 'y': 3, 's': 6, 'n': 1, 'd': 9, 'r': 2})
6415 + 734 = 7149 (mapping: {'o': 7, 'm': 0, 'e': 4, 'y': 9, 's': 6, 'n': 1, 'd': 5, 'r': 3})
6524 + 735 = 7259 (mapping: {'o': 7, 'm': 0, 'e': 5, 'y': 9, 's': 6, 'n': 2, 'd': 4, 'r': 3})
6853 + 728 = 7581 (mapping: {'o': 7, 'm': 0, 'e': 8, 'y': 1, 's': 6, 'n': 5, 'd': 3, 'r': 2})
6851 + 738 = 7589 (mapping: {'o': 7, 'm': 0, 'e': 8, 'y': 9, 's': 6, 'n': 5, 'd': 1, 'r': 3})
7316 + 823 = 8139 (mapping: {'o': 8, 'm': 0, 'e': 3, 'y': 9, 's': 7, 'n': 1, 'd': 6, 'r': 2})
7429 + 814 = 8243 (mapping: {'o': 8, 'm': 0, 'e': 4, 'y': 3, 's': 7, 'n': 2, 'd': 9, 'r': 1})
7539 + 815 = 8354 (mapping: {'o': 8, 'm': 0, 'e': 5, 'y': 4, 's': 7, 'n': 3, 'd': 9, 'r': 1})
7531 + 825 = 8356 (mapping: {'o': 8, 'm': 0, 'e': 5, 'y': 6, 's': 7, 'n': 3, 'd': 1, 'r': 2})
7534 + 825 = 8359 (mapping: {'o': 8, 'm': 0, 'e': 5, 'y': 9, 's': 7, 'n': 3, 'd': 4, 'r': 2})
7649 + 816 = 8465 (mapping: {'o': 8, 'm': 0, 'e': 6, 'y': 5, 's': 7, 'n': 4, 'd': 9, 'r': 1})
7643 + 826 = 8469 (mapping: {'o': 8, 'm': 0, 'e': 6, 'y': 9, 's': 7, 'n': 4, 'd': 3, 'r': 2})
8324 + 913 = 9237 (mapping: {'o': 9, 'm': 0, 'e': 3, 'y': 7, 's': 8, 'n': 2, 'd': 4, 'r': 1})
8432 + 914 = 9346 (mapping: {'o': 9, 'm': 0, 'e': 4, 'y': 6, 's': 8, 'n': 3, 'd': 2, 'r': 1})
8542 + 915 = 9457 (mapping: {'o': 9, 'm': 0, 'e': 5, 'y': 7, 's': 8, 'n': 4, 'd': 2, 'r': 1})


Process exited with code: 0
```

EXPERIMENT 4

Sheel Patel – RA1911003010439

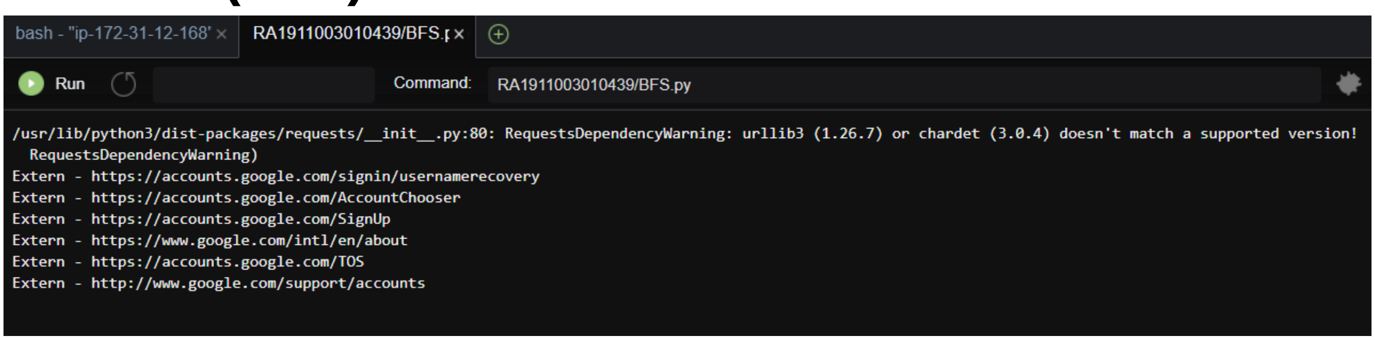
Aim- Implementation and Analysis of DFS and BFS for an application

BFS code-

```
from urllib.request import urljoin
from bs4 import BeautifulSoup
import requests
from urllib.request import urlparse


links_intern = set()
input_url = "https://classroom.google.com/u/2/c/NDUwOTY2NTQyODUy"
depth = 1

links_extern = set()


def level_crawler(input_url):
    temp_urls = set()
    current_url_domain = urlparse(input_url).netloc

    beautiful_soup_object = BeautifulSoup(
        requests.get(input_url).content, "lxml")

    for anchor in beautiful_soup_object.findAll("a"):
        href = anchor.attrs.get("href")
        if(href != "" or href != None):
            href = urljoin(input_url, href)
            href_parsed = urlparse(href)
            href = href_parsed.scheme
            href += "://"
            href += href_parsed.netloc
            href += href_parsed.path
            final_parsed_href = urlparse(href)
            is_valid = bool(final_parsed_href.scheme) and bool(
                final_parsed_href.netloc)
            if is_valid:
                if current_url_domain not in href and href not in links_extern:
```

```
                    print("Extern - {}".format(href))
                    links_extern.add(href)
                if current_url_domain in href and href not in links_intern:
                    print("Intern - {}".format(href))
                    links_intern.add(href)
                    temp_urls.add(href)
    return temp_urls


if(depth == 0):
    print("Intern - {}".format(input_url))

elif(depth == 1):
    level_crawler(input_url)

else:
    queue = []
    queue.append(input_url)
    for j in range(depth):
        for count in range(len(queue)):
            url = queue.pop(0)
            urls = level_crawler(url)
            for i in urls:
                queue.append(i)
```

OUTPUT(BFS)-

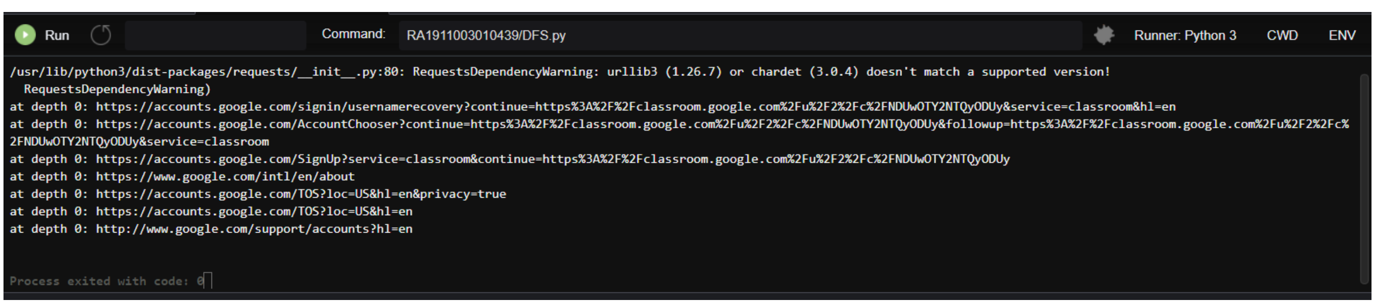


DFS Code-

```
import requests
from bs4 import BeautifulSoup
```

```
def get_links_recursive(base, path, visited, max_depth=3, depth=0):
    if depth < max_depth:
        try:
            soup = BeautifulSoup(requests.get(base + path).text, "html.parser")

            for link in soup.find_all("a"):
                href = link.get("href")

                if href not in visited:
                    visited.add(href)
                    print(f"at depth {depth}: {href}")

                    if href.startswith("http"):
                        get_links_recursive(href, "", visited, 1, depth + 1)
                    else:
                        get_links_recursive(base, href, visited, 1, depth + 1)
        except:
            pass


get_links_recursive("https://classroom.google.com/u/2/c/
NDUwOTY2NTQyODUy", "", set(["google"]))
```

OUTPUT-



RESULT-

Thus we successfully implemented web scraping using DFS and BFS.

EXPERIMENT 5

Sheel Patel - RA1911003010439

Best first Search -

Algorithm-

1. Create 2 empty lists: OPEN and CLOSED
2. Start from the initial node (say N) and put it in the 'ordered' OPEN list
3. Repeat the next steps until GOAL node is reached
    1. If OPEN list is empty, then EXIT the loop returning 'False'
    2. Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also capture the information of the parent node
    3. If N is a GOAL node, then move the node to the Closed list and exit the loop returning 'True'. The solution can be found by backtracking the path
    4. If N is not the GOAL node, expand node N to generate the 'immediate' next nodes linked to node N and add all those to the OPEN list
    5. Reorder the nodes in the OPEN list in ascending order according to an evaluation function f(n)

This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by **O(n*logn)** .

Code-

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pi;

vector<vector<pi> > graph;

void addedge(int x, int y, int cost)
{
	graph[x].push_back(make_pair(cost, y));
	graph[y].push_back(make_pair(cost, x));
}
```

```
void best_first_search(int source, int target, int n)
{
	vector<bool> visited(n, false);
	priority_queue<pi, vector<pi>, greater<pi> > pq;
	pq.push(make_pair(0, source));
	int s = source;
	visited[s] = true;
	while (!pq.empty()) {
		int x = pq.top().second;
		cout << x << " ";
		pq.pop();
		if (x == target)
			break;

		for (int i = 0; i < graph[x].size(); i++) {
			if (!visited[graph[x][i].second]) {
				visited[graph[x][i].second] = true;
				pq.push(make_pair(graph[x][i].first,graph[x][i].second));
			}
		}
	}
}

int main()
{

	int v = 14;
	graph.resize(v);
	addedge(0, 1, 3);
	addedge(0, 2, 6);
	addedge(0, 3, 5);
	addedge(1, 4, 9);
	addedge(1, 5, 8);
	addedge(2, 6, 12);
	addedge(2, 7, 14);
	addedge(3, 8, 7);
	addedge(8, 9, 5);
	addedge(8, 10, 6);
	addedge(9, 11, 1);
	addedge(9, 12, 10);
	addedge(9, 13, 2);

	int source = 0;
```

```
    int target = 9;

    best_first_search(source, target, v);

    return 0;
}
```

Output-

```
Running /home/ubuntu/environment/439/A-star.cpp
0 1 3 2 8 9

Process exited with code: 0
```

A* search algorithm-

Algorithm-

1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)

3. while the open list is not empty
   a) find the node with the least f on
      the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their
      parents to q

   d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for successor
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-

Manhattan, Diagonal and Euclidean
Heuristics)

successor.f = successor.g + successor.h

iii) if a node with the same position as
successor is in the OPEN list which has a
lower f than successor, skip this successor

iV) if a node with the same position as
successor is in the CLOSED list which has
a lower f than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

Code-

```
from collections import deque

class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
```

```
g = {}

g[start_node] = 0

parents = {}
parents[start_node] = start_node

while len(open_list) > 0:
    n = None

    for v in open_list:
        if n == None or g[v] + self.h(v) < g[n] + self.h(n):
            n = v;

    if n == None:
        print('Path does not exist!')
        return None

    if n == stop_node:
        reconst_path = []

        while parents[n] != n:
            reconst_path.append(n)
            n = parents[n]

        reconst_path.append(start_node)

        reconst_path.reverse()

        print('Path found: {}'.format(reconst_path))
        return reconst_path

    for (m, weight) in self.get_neighbors(n):

        if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight

        else:
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n
```

```
                if m in closed_list:
                    closed_list.remove(m)
                    open_list.add(m)

            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

Output-

Result- Thus best first search and A* search algorithms were successfully implemented.

EXPERIMENT 6

Sheel Patel - RA1911003010439

Aim- Implementation of mini max algorithm for an application(Application - Tic Tac Toe).

Algorithm-

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state :
        return value of the board

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth+1, false)
            bestVal = max( bestVal, value)
        return bestVal

    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth+1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

Code-

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system

"""
An implementation of Minimax AI Algorithm in Tic Tac Toe,
using Python.
This software is available under GPL license.
Author: Clederson Cruz
Year: 2017
```

```
License: GNU GENERAL PUBLIC LICENSE (GPL)
"""

HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]


def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


def wins(state, player):
    """
    This function tests if a specific player wins. Possibilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
```

```python
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False


def game_over(state):
    """
    This function test if the human or computer wins
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)


def empty_cells(state):
    """
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells


def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
```

```
    else:
        return False


def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
```

```
            if score[2] > best[2]:
                best = score
        else:
            if score[2] < best[2]:
                best = score

    return best


def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')


def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)


def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
```

```
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)


def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
```

```
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'
```

```
    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(board):
        if first == 'N':
            ai_turn(c_choice, h_choice)
            first = ''

        human_turn(c_choice, h_choice)
        ai_turn(c_choice, h_choice)

    # Game over message
    if wins(board, HUMAN):
        clean()
        print(f'Human turn [{h_choice}]')
        render(board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins(board, COMP):
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')

    exit()


if __name__ == '__main__':
    main()
```

Output-

```
---------------
Computer turn [O]

---------------
| o || o || o |
---------------
|   || x || x |
---------------
| o || x || x |
---------------
YOU LOSE!
```

Result-Thus min max algorithm was implemented with the application of tictactoe.

EXPERIMENT 7

Sheel Patel - RA1911003010439

AIM - Implementation of unification and resolution for real world problems.

UNIFICATION

CODE -

```
def get_index_comma(string):

    index_list = list()
    par_count = 0
    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1
    return index_list

def is_variable(expr):

    for i in expr:
        if i == '(':
            return False
    return True

def process_expression(expr):

    expr = expr.replace(' ', '')
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break
```

```
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')


    expr = expr[1:len(expr) - 1]


    arg_list = list()


    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list

def get_arg_list(expr):

    _, arg_list = process_expression(expr)
    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list
```

```
def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False

def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)


        if predicate_symbol_1 != predicate_symbol_2:
            return False

        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:

            sub_list = list()
```

```
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)


            return sub_list

if __name__ == '__main__':

    f1 = 'p(a, g(x,a), f(y)) '
    f2 = 'p(a, g(f(b),a), x)'
    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successfully!')
        print(result)
```

OUTPUT -

```
RA1911003010443:~/environment/439 $ python unif.py
Unification successfully!
['f(b)/x', 'f(y)/x']
```

RESOLUTION

CODE -

```
import copy
import time
```

```
class Parameter:
    variable_count = 1
    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name


class Predicate:
    def __init__(self, name, params):
        self.name = name
        self.params = params

    def __eq__(self, other):
        return self.name == other.name and all(a == b for a, b in zip(self.params,
other.params))

    def __str__(self):
        return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

    def getNegatedPredicate(self):
        return Predicate(negatePredicate(self.name), self.params)
```

```
class Sentence:
    sentence_count = 0

    def __init__(self, string):
        self.sentence_index = Sentence.sentence_count
        Sentence.sentence_count += 1
        self.predicates = []
        self.variable_map = {}
        local = {}

        for predicate in string.split("|"):
            name = predicate[:predicate.find("(")]
            params = []

            for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):
                if param[0].islower():
                    if param not in local:  # Variable
                        local[param] = Parameter()
                        self.variable_map[local[param].name] = local[param]
                    new_param = local[param]
                else:
                    new_param = Parameter(param)
                    self.variable_map[param] = new_param

                params.append(new_param)

            self.predicates.append(Predicate(name, params))

    def getPredicates(self):
        return [predicate.name for predicate in self.predicates]

    def findPredicates(self, name):
        return [predicate for predicate in self.predicates if predicate.name == name]

    def removePredicate(self, predicate):
        self.predicates.remove(predicate)
        for key, val in self.variable_map.items():
            if not val:
                self.variable_map.pop(key)
```

```
    def containsVariable(self):
        return any(not param.isConstant() for param in self.variable_map.values())

    def __eq__(self, other):
        if len(self.predicates) == 1 and self.predicates[0] == other:
            return True
        return False

    def __str__(self):
        return "".join([str(predicate) for predicate in self.predicates])


class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(
                    predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceIdx]:
                self.inputSentences[sentenceIdx] = negateAntecedent(
                    self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
```

```
        prev_sentence_map = copy.deepcopy(self.sentence_map)
        self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
            negatedPredicate.name, []) + [negatedQuery]
        self.timeLimit = time.time() + 40

        try:
            result = self.resolve([negatedPredicate], [
                                  False]*(len(self.inputSentences) + 1))
        except:
            result = False

        self.sentence_map = prev_sentence_map

        if result:
            results.append("TRUE")
        else:
            results.append("FALSE")

    return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
            queryPredicate = negatedQuery
            for kb_sentence in self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

                        canUnify, substitution = performUnification(
                            copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

                        if canUnify:
                            newSentence = copy.deepcopy(kb_sentence)
```

```
                newSentence.removePredicate(kbPredicate)
                newQueryStack = copy.deepcopy(queryStack)

                if substitution:
                    for old, new in substitution.items():
                        if old in newSentence.variable_map:
                            parameter = newSentence.variable_map[old]
                            newSentence.variable_map.pop(old)
                            parameter.unify(
                                "Variable" if new[0].islower() else "Constant", new)
                            newSentence.variable_map[new] = parameter

                    for predicate in newQueryStack:
                        for index, param in enumerate(predicate.params):
                            if param.name in substitution:
                                new = substitution[param.name]
                                predicate.params[index].unify(
                                    "Variable" if new[0].islower() else "Constant", new)

                for predicate in newSentence.predicates:
                    newQueryStack.append(predicate)

                new_visited = copy.deepcopy(visited)
                if kb_sentence.containsVariable() and len(kb_sentence.predicates)
> 1:
                    new_visited[kb_sentence.sentence_index] = True

                if self.resolve(newQueryStack, new_visited, depth + 1):
                    return True
        return False
    return True


def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
```

```
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
    return True, substitution


def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate


def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)
```

```
def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                          for _ in range(noOfSentences)]
        return inputQueries, inputSentences


def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()


if __name__ == '__main__':
    print("Resolution Output:!")
    inputQueries_, inputSentences_ = getInput('input-exp7.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)
```

INPUT -

```
6
F(Joe)
H(John)
~H(Alice)
~H(John)
G(Joe)
G(Tom)
14
~F(x) | G(x)
~G(x) | H(x)
~H(x) | F(x)
~R(x) | H(x)
~A(x) | H(x)
~D(x,y) | ~H(y)
~B(x,y) | ~C(x,y) | A(x)
B(John,Alice)
B(John,Joe)
~D(x,y) | ~Q(y) | C(x,y)
D(John,Alice)
Q(Joe)
D(John,Joe)
R(Tom)
```

OUTPUT -

```
RA1911003010443:~/environment/439 $ python resol.py
Resolution Output:!
['FALSE', 'TRUE', 'TRUE', 'FALSE', 'FALSE', 'TRUE']
```

RESULT -

Unification and Resolution were implemented successfully.

EXPERIMENT 8

Sheel Patel - RA1911003010439

AIM - To implement the Sudoku problem

CODE -

Sudoku.py

```
from typing import *
from utils import cross,chunk_string_by_len
ROWS = 'ABCDEFGHI'
COLS = '123456789'
boxes = cross(ROWS, COLS)
row_units = [cross(r,COLS) for r in ROWS]
col_units = [cross(ROWS,c) for c in COLS]
square_units = [cross(r,c) for r in chunk_string_by_len(ROWS) for c in
chunk_string_by_len(COLS)]
unit_list = row_units + col_units + square_units
def get_puzzle(complex:bool = False) -> str:
    if complex:
        return '4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......'
    return '..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9..5.1.3..'
def grid_values(puzzle:str,boxes:List[str],replace:bool=True) -> Dict[str,str]:
    assert len(puzzle) == 81
    return {key : ( '123456789' if value =='.' and replace else value) for key,value in
zip(boxes,puzzle)}
def display_sudoku(p_values:Dict[str,str]) -> None:
    assert (len(p_values) == 81),"There must be 81 values in the dictionary."
    max_len=len(max(list(p_values.values()),key=len))+2 #max length among all box
units
    print(f"\n{' SUDOKU '.center(max_len*9,'=')}\n")
    list_puzzle = list(p_values.items())
    n=9 #step
    for i in range(0,len(p_values),n):
        row=''
        for index,box in enumerate(list_puzzle[i:i+n]):
            if (index > 1 and index < 9) and index % 3 == 0 :
                row +='|' #to add a pipe in middle
```

```
            row +=box[1].center(max_len)
        print(row,'\n')
        if i == 18 or i== 45 : #to add a decorative line in middle
            pt='-'*(max_len*3) #tern
            print('+'.join([pt,pt,pt]),'\n')
def find_peers(box:str) -> List[str]:
    peers_list=[list for list in unit_list if box in list]
    peers = list(set([item for sub_list in peers_list for item in sub_list if item !=box]))
    return peers
def eliminate(grids:Dict[str,str]) -> Dict[str,str]:

    for key,value in grids.items():
        if len(value) > 1:
            peers = find_peers(key)
            peers_values = [grids.get(k) for k in peers if len(grids.get(k))==1]
            for v in peers_values:
                value=value.replace(v,"")
            grids[key]=value
    return grids
def only_choice(grids:Dict[str,str]) -> Dict[str,str]:
    for unit in unit_list:
        for digit in '123456789':
            d_places = [box for box in unit if digit in grids[box]]
            if len(d_places) == 1:
                grids[d_places[0]] = digit
    return grids
def reduce_puzzle(grids:Dict[str,str]) -> Union[Dict[str,str],bool]:
    stalled = False
    solved = False
    while not stalled:
        solved_values_before = len([value for value in grids.values() if
len(value)==1])#total units
        grids = eliminate(grids)
        grids = only_choice(grids)
        solved_values_after = len([value for value in grids.values() if len(value)==1])#total
units
        stalled = solved_values_before == solved_values_after
        if len([box for box in grids.keys() if len(grids[box]) == 0]):
            return False
    return grids
```

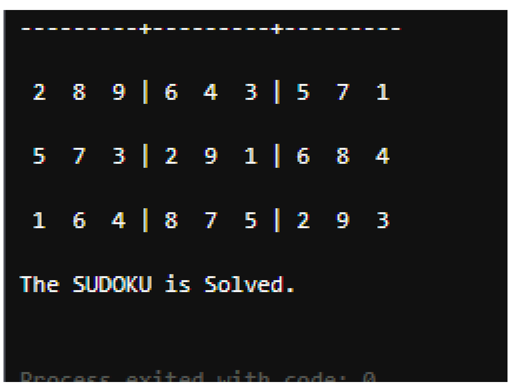
```
def search(values:Dict[str,str])->Dict[str,str]:
    values = reduce_puzzle(values)
    if values is False:
        return False ## Failed earlier
    if all(len(values[s]) == 1 for s in boxes):
        return values ## Solved!
    n,k = min((len(v),k) for k,v in values.items() if len(v)>1)
    for value in values[k]:
        new_sudoku=values.copy()
        new_sudoku[k]=value
        attempt = search(new_sudoku,)
        if attempt:
            return attempt
def check_if_sudoku_solved(grids:Dict[str,str]) -> bool:
    for unit in unit_list:
        unit_values_sum = sum([int(grids.get(k)) for k in unit])
        solved = unit_values_sum == 45
    return solved
def main(display_units:bool=False):
    if display_units:
        print(f"boxes : \n{boxes}\n")
        print(f"row_units : \n{row_units}\n")
        print(f"col_units : \n{col_units}\n")
        print(f"square_units : \n{square_units}\n")
        print(f"unit_lists : \n{unit_list}\n")
    puzzle = get_puzzle(complex=True)
    print("\nUnsolved Sudoku.")
    display_sudoku(grid_values(puzzle,boxes,replace=False))
    print("\nSudoku with replaced dots by 1-9.")
    grid_units = grid_values(puzzle,boxes)
    display_sudoku(grid_units) #display replaced
    print("\nSudoku with eliminated values.")
    eliminated_values=eliminate(grid_units)
    display_sudoku(eliminated_values) #display eliminated
    print("\nSudoku after replacing with only choices.")
    elimination_with_only_coices_values=only_choice(eliminated_values)
    display_sudoku(elimination_with_only_coices_values)
    print("\nSudoku after Constraint Propagation.")
    reduced_puzzle_values=reduce_puzzle(eliminated_values)
    display_sudoku(reduced_puzzle_values)
```

```
    solved = check_if_sudoku_solved(reduced_puzzle_values)
    if not solved:
        print("\nThe SUDOKU is UnSolved and needs searching.")
        print("Sudoku after Search.")
        solved_puzzle_with_search=search(eliminated_values)
        display_sudoku(solved_puzzle_with_search)
        solved = check_if_sudoku_solved(solved_puzzle_with_search)
    print(f'The SUDOKU is {"Solved" if solved else "UnSolved"}.')
if __name__ == "__main__":
    main()
```

Utils.py

```
from typing import List
def cross(row: str, col: str) -> List[str]:
    return [r + c for r in row for c in col]
def chunk_string_by_len(string: str, n: int = 3) -> List[str]:
    return [string[i:i+n] for i in range(0, len(string), n)]
```

OUTPUT -



RESULT -

Sudoku was implemented successfully.

EXPERIMENT 9

Sheel Patel - RA1911003010439

AIM - To implement the Monty Hall problem.

CODE -

```
import random
A = "A"
B = "B"
C = "C"
doors = ["A", "B", "C"]
prize = random.choice(doors)
selection = input("Select door 'A', 'B', or 'C': ")
if selection == prize:
   remaining = list(set(doors) - set(prize))
   open_door = random.choice(list(set(doors) -
set(random.choice(remaining))))
   alternate = random.choice(list(set(doors) - set(open_door) - set(prize)))
else:
   open_door = random.choice(list(set(doors) - set(selection) - set(prize)))
   alternate = random.choice(list(set(doors) - set(open_door) -
set(selection)))
print ("""The door I will now open is: %r""" % open_door)
second_chance = input("Would you like to select the third door? Type 'Yes'
or 'No': ")
if second_chance == "Yes":
   print ("""The door you will switch to is: %r """ % alternate)
   if alternate == prize:
      print( """Congrats, you win! The prize was behind the alternate, %r"""
% alternate)
   else:
      print ("""Sorry, the prize was behind the original door %r""" % prize)
if second_chance != "Yes":
```

```
    print ("""You decided to keep your initial door, %r""" % selection)
if selection != prize:
    print ("""Sorry, the prize was behind the alternate door, %r""" % prize)
else:
    print ("""Congrats, you win! The prize was behind your original selection,
%r""" % selection)
print( """This is a check:""")
print( "Prize: %r" % prize)
print ("Selection: %r " % selection)
print( "Alternate: %r " % alternate)
print ("Door opened: %r " % open_door)
```

OUTPUT -

```
Select door 'A', 'B', or 'C': B
The door I will now open is: 'C'
Would you like to select the third door? Type 'Yes' or 'No': No
You decided to keep your initial door, 'B'
Sorry, the prize was behind the alternate door, 'A'
This is a check:
Prize: 'A'
Selection: 'B'
Alternate: 'A'
Door opened: 'C'
```

RESULT -

The Monty Hall problem was implemented successfully.

Sheel Patel
RA1911003010439
Exp10

Aim- To implement Block World Program using Python.

Algorithm-
The algorithm is similar to a set of wooden blocks of various shapes and colors sitting on a table. The goal is to build one or more vertical stacks of blocks.

Code-

```
    #Base Classes

#PREDICATE - ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY
class PREDICATE:
  def __str__(self):
    pass
  def __repr__(self):
    pass
  def __eq__(self, other) :
    pass
  def __hash__(self):
    pass
  def get_action(self, world_state):
    pass


#OPERATIONS - Stack, Unstack, Pickup, Putdown
class Operation:
  def __str__(self):
    pass
  def __repr__(self):
    pass
```

```
  def __eq__(self, other) :
    pass
  def precondition(self):
    pass
  def delete(self):
    pass
  def add(self):
    pass


class ON(PREDICATE):

  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "ON({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def __hash__(self):
      return hash(str(self))

  def get_action(self, world_state):
    return StackOp(self.X,self.Y)


class ONTABLE(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "ONTABLE({X})".format(X=self.X)
```

```
  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def __hash__(self):
      return hash(str(self))

  def get_action(self, world_state):
    return PutdownOp(self.X)


class CLEAR(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "CLEAR({X})".format(X=self.X)
    self.X = X

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state):
    for predicate in world_state:
      #If Block is on another block, unstack
      if isinstance(predicate,ON) and predicate.Y==self.X:
        return UnstackOp(predicate.X, predicate.Y)
    return None
```

```
class HOLDING(PREDICATE):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "HOLDING({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state):
    X = self.X
    #If block is on table, pick up
    if ONTABLE(X) in world_state:
      return PickupOp(X)
    #If block is on another block, unstack
    else:
      for predicate in world_state:
        if isinstance(predicate,ON) and predicate.X==X:
          return UnstackOp(X,predicate.Y)


class ARMEMPTY(PREDICATE):

  def __init__(self):
    pass

  def __str__(self):
    return "ARMEMPTY"
```

```
  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def __hash__(self):
    return hash(str(self))

  def get_action(self, world_state=[]):
    for predicate in world_state:
      if isinstance(predicate,HOLDING):
        return PutdownOp(predicate.X)
    return None


class StackOp(Operation):

  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def precondition(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def delete(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def add(self):
```

```
    return [ ARMEMPTY() , ON(self.X,self.Y) ]


class UnstackOp(Operation):

  def __init__(self, X, Y):
    self.X = X
    self.Y = Y

  def __str__(self):
    return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def precondition(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

  def delete(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]

  def add(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]


class PickupOp(Operation):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "PICKUP({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()
```

```
  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def precondition(self):
    return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

  def delete(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

  def add(self):
    return [ HOLDING(self.X) ]


class PutdownOp(Operation):

  def __init__(self, X):
    self.X = X

  def __str__(self):
    return "PUTDOWN({X})".format(X=self.X)

  def __repr__(self):
    return self.__str__()

  def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ ==
other.__class__

  def precondition(self):
    return [ HOLDING(self.X) ]

  def delete(self):
    return [ HOLDING(self.X) ]

  def add(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]


def isPredicate(obj):
```

```python
  predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]
  for predicate in predicates:
    if isinstance(obj,predicate):
      return True
  return False

def isOperation(obj):
  operations = [StackOp, UnstackOp, PickupOp, PutdownOp]
  for operation in operations:
    if isinstance(obj,operation):
      return True
  return False

def arm_status(world_state):
  for predicate in world_state:
    if isinstance(predicate, HOLDING):
      return predicate
  return ARMEMPTY()


class GoalStackPlanner:

  def __init__(self, initial_state, goal_state):
    self.initial_state = initial_state
    self.goal_state = goal_state

  def get_steps(self):

    #Store Steps
    steps = []

    #Program Stack
    stack = []

    #World State/Knowledge Base
    world_state = self.initial_state.copy()

    #Initially push the goal_state as compound goal onto the stack
    stack.append(self.goal_state.copy())
```

```
    #Repeat until the stack is empty
    while len(stack)!=0:

      #Get the top of the stack
      stack_top = stack[-1]

      #If Stack Top is Compound Goal, push its unsatisfied goals onto stack
      if type(stack_top) is list:
        compound_goal = stack.pop()
        for goal in compound_goal:
          if goal not in world_state:
            stack.append(goal)

      #If Stack Top is an action
      elif isOperation(stack_top):

        #Peek the operation
        operation = stack[-1]

        all_preconditions_satisfied = True

        #Check if any precondition is unsatisfied and push it onto program stack
        for predicate in operation.delete():
          if predicate not in world_state:
            all_preconditions_satisfied = False
            stack.append(predicate)

        #If all preconditions are satisfied, pop operation from stack and execute it
        if all_preconditions_satisfied:

          stack.pop()
          steps.append(operation)

          for predicate in operation.delete():
            world_state.remove(predicate)
          for predicate in operation.add():
            world_state.append(predicate)
```

```
      #If Stack Top is a single satisfied goal
      elif stack_top in world_state:
        stack.pop()

      #If Stack Top is a single unsatisfied goal
      else:
        unsatisfied_goal = stack.pop()

        #Replace Unsatisfied Goal with an action that can complete it
        action = unsatisfied_goal.get_action(world_state)

        stack.append(action)
        #Push Precondition on the stack
        for predicate in action.precondition():
          if predicate not in world_state:
            stack.append(predicate)

    return steps

if __name__ == '__main__':
  initial_state = [
    ON('B','A'),
    ONTABLE('A'),ONTABLE('C'),ONTABLE('D'),
    CLEAR('B'),CLEAR('C'),CLEAR('D'),
    ARMEMPTY()
  ]

  goal_state = [
    ON('B','D'),ON('C','A'),
    ONTABLE('D'),ONTABLE('A'),
    CLEAR('B'),CLEAR('C'),
    ARMEMPTY()
  ]

  goal_stack = GoalStackPlanner(initial_state=initial_state,
goal_state=goal_state)
  steps = goal_stack.get_steps()
  print(steps)
```

Output-

```
[PICKUP(C), PUTDOWN(C), UNSTACK(B,A), PUTDOWN(B), PICKUP(C), STACK(C,A), PICKUP(B), STACK(B,D)]
```

Result-

Thus we implemented block world program using Python.