|  |  |
| --- | --- |
| Programme: Artificial Intelligence | Degree: B.E. |
| Course Code: 3170716 | Semester: 7 |
| Credits: 5 | Contact hours: 3 (Theory) + 2 (Laboratory) |
| Faculty Name: Jigna Jadav | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr.**  **No.** | **Topics** | **CO**  **covered** | **Date** | **Page** |
| 1. | A) Write a PROLOG program to implement different kinds of knowledge bases.  B) Write a PROLOG program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece. | CO2  CO5 | 30/6/21 | 1 |
| 2. | Write a PROLOG program to implement Water-Jug Problem. | CO1  CO5 | 28/7/21 | 6 |
| 3. | Solve 8 Puzzle Problem using A\* Algorithm in any programming Language. | CO1 | 11/8/21 | 8 |
| 4. | Convert the given PROLOG predicates into Semantic Net.  cat(tom).  cat(cat1). mat(mat1).  sat\_on(cat1,mat1).  bird(bird1).  caught(tom,bird1).  like(X,cream) :– cat(X).  mammal(X) :– cat(X).  has(X,fur) :– mammal(X).  animal(X) :– mammal(X).  animal(X) :– bird(X).  owns(john,tom). is\_coloured(tom,ginger). | CO2 | 7/7/21 | 15 |
| 5. | Write the Conceptual Dependency for the given statements.  (a)John gives Mary a book  (b)John gave Mary the book yesterday. | CO2 | 14/7/21 | 16 |
| 6. | Implement Bayesian Networks algorithm. | CO4 | 1/9/21 | 17 |
| 7. | Implement Min Max Algorithm for any problem. | CO3 | 15/9/21 | 20 |
| 8. | Demonstrate Connectionist Model using Tool. | CO4 | 29/9/21 | 25 |
| 9. | Implement Genetic Algorithm. | CO4 | 25/8/21 | 28 |
| 10. | Write a PROLOG program based on list:  A) To find the length of a list.  B) To sum all numbers of list.  C) To find whether given element is a member of a list.  D) To Append the list.  E) To Reverse the list.  F) To find the last element of a list.  G) To delete the first occurrence of an element from a list.  H) To delete every occurrence of an element from a list.  To find Nth element from the list. | CO5 | 22/9/21 | 41 |

**Practical List**

**AIM:**

(A). Write a PROLOG program to implement different kinds of knowledge bases.

* **Knowledge Base 1**

woman(mia).

woman(jody).

woman(yolanda).

playsAirGuitar(jody).

party.

|  |
| --- |
| ?- pwd.  % f:/soft/swi-prolog/swipl/bin/  true.  ?- cd('../../').  true.  ?- pwd.  % f:/soft/swi-prolog/  true.  ?- ls.  % kb1.pl kb2.pl kb3.pl kb4.pl swipl/  true.  ?- [kb1].  true.  ?- woman(mia).  true.  ?- playsAirGuitar(jody).  true.  ?- playsAirGuitar(mia).  false.  ?- playsAirGuitar(vincent).  false. |

* **Knowledge Base 2**

happy(yolanda).

listen2Music(mia).

listen2Music(yolanda):- happy(yolanda).

playsAirGuitar(mia):- listen2Music(mia).

playsAirGuitar(yolanda):- listen2Music(yolanda).

|  |
| --- |
| ?- [kb2].  true.  ?- playsAirGuitar(mia).  true.  ?- playsAirGuitar(yolanda).  true. |

* **Knowledge Base 3**

happy(vincent).

listens2Music(butch).

playsAirGuitar(vincent):- listens2Music(vincent),happy(vincent).

playsAirGuitar(butch):- happy(butch).

playsAirGuitar(butch):- listens2Music(butch).

|  |
| --- |
| ?- [kb3].  true.  ?- playsAirGuitar(vincent).  false.  ?- playsAirGuitar(butch).  true. |

* **Knowledge Base 4 & 5**

woman(jody).

woman(mia).

woman(yolanda).

loves(vincent,mia).

loves(marsellus,mia).

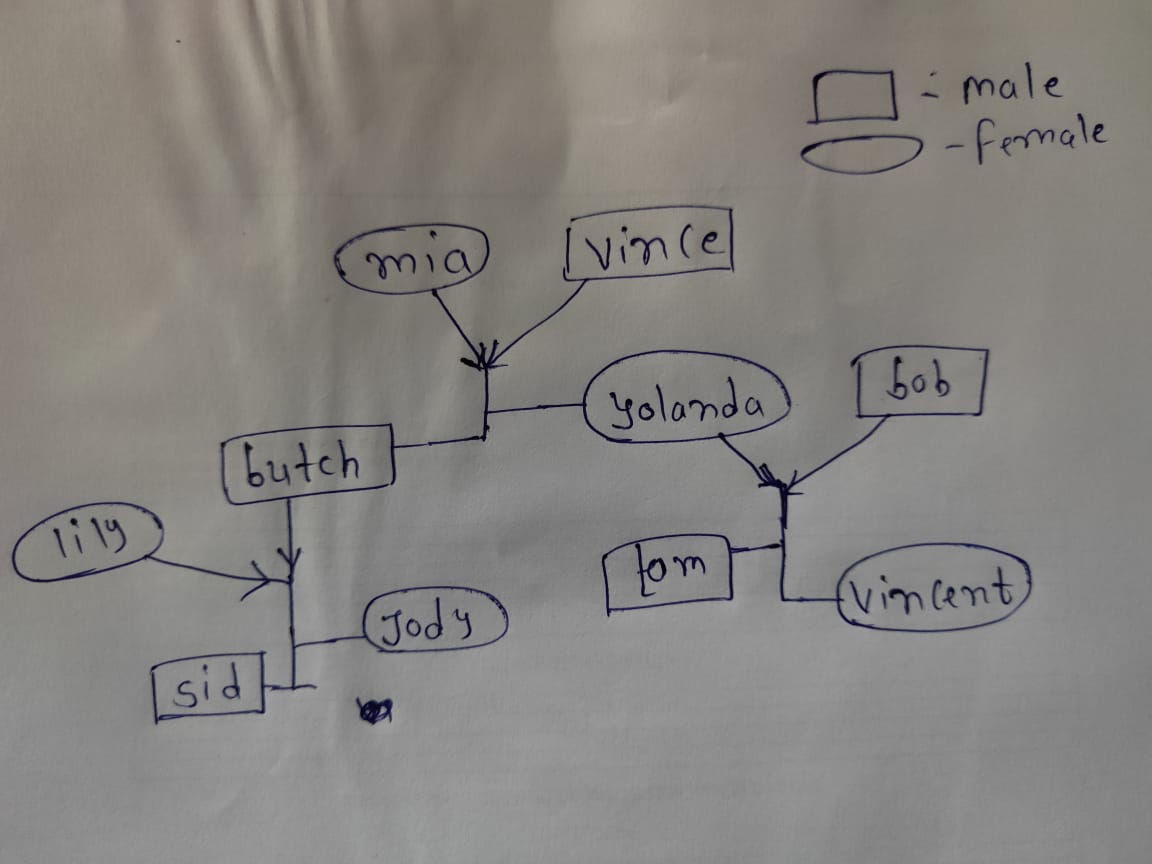
loves(pumpkin,honey\_bunny).

loves(honey\_bunny,pumpkin).

jealous(X,Y):- loves(X,Z), loves(Y,Z).

|  |
| --- |
| ?- [kb4].  true.  ?- woman(X).  X = jody ;  X = mia ;  X = yolanda.  ?- jealou(marsellus,W).  Correct to: "jealous(marsellus,W)"? yes  W = vincent ;  W = marsellus |

B) Write a PROLOG program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece.



male(bob).

male(butch).

male(sid).

male(tom).

male(vince).

female(jody).

female(lily).

female(mia).

female(vincent).

female(yolanda).

parent(mia,butch).

parent(mia,yolanda).

parent(vince,butch).

parent(vince,yolanda).

parent(butch,jody).

parent(butch,sid).

parent(yolanda,vincent).

parent(yolanda,tom).

parent(bob,vincent).

parent(bob,tom).

parent(lily,jody).

parent(lily,sid).

mother(X,Y) :- parent(X,Y),female(X).

father(X,Y) :- parent(X,Y),male(X).

grandfather(X,Y) :- parent(Z,Y),father(X,Z).

grandmother(X,Y) :- parent(Z,Y),mother(X,Z).

sister(X,Y) :- mother(Z,X),mother(Z,Y),female(X),X\==Y.

brother(X,Y) :- mother(Z,X),mother(Z,Y),male(X),X\==Y.

uncle(X,Y) :- parent(Z,Y),mother(W,Z),mother(W,X),male(X),X\==Z.

aunt(X,Y) :- parent(Z,Y),mother(W,Z),mother(W,X),female(X),X\==Z.

aunt(X,Y) :- uncle(Z,Y),mother(X,W),father(Z,W),male(W).

niece(X,Y) :- uncle(Y,X),female(X).

niece(X,Y) :- aunt(Y,X),female(X).

nephew(X,Y) :- uncle(Y,X),male(X).

nephew(X,Y) :- aunt(Y,X),male(X).

|  |
| --- |
| ?- [practical].  true.  ?- niece(X,Y).  X = vincent,  Y = butch ;  X = jody,  Y = yolanda ;  X = vincent,  Y = lily ;  false.  ?- grandfather(X,Y).  X = vince,  Y = jody ;  X = vince,  Y = sid ;  X = vince,  Y = vincent ;  X = vince,  Y = tom ;  false.  ?- aunt(X,Y).  X = yolanda,  Y = jody ;  X = yolanda,  Y = sid ;  X = lily,  Y = vincent ;  X = lily,  Y = tom ;  false.  ?- brother(X,Y).  X = butch,  Y = yolanda ;  X = tom,  Y = vincent ;  X = sid,  Y = jody ;  false.  ?- mother(mia,sid).  false. |

AIM: Write a PROLOG program to implement Water-Jug Problem.

* A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

**State Representation and Initial State:**

we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note 0 ≤x ≤4, and 0≤y ≤3.

Our initial state: (0,0)

Goal Predicate state = (2,y) where 0 ≤y ≤3.

|  |  |  |
| --- | --- | --- |
| Gals in 4-gal jug | Gals in 3-gal jug | Rule Applied |
| 0 | 0 | 1. Fill 4 |
| 4 | 0 | 6. Pour 4 into 3 to fill |
| 1 | 3 | 4. Empty 3 |
| 1 | 0 | 8. Pour all of 4 into 3 |
| 0 | 1 | 1. Fill 4 |
| 4 | 1 | 6. Pour into 3 |
| 2 | 3 |  |

start**(**2**,**0**):-write(**' 4lit Jug: 2 | 3lit Jug: 0|\n'**),**

**write(**'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n'**),**

**write(**'Goal Reached! Congrats!!\n'**),**

**write(**'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n'**).**

start**(**X**,**Y**):-write(**' 4lit Jug: '**),write(**X**),write(**'| 3lit Jug: '**),**

**write(**Y**),write(**'|\n'**),**

**write(**' Enter the move::'**),**

**read(**N**),**

contains**(**X**,**Y**,**N**).**

contains**(**\_**,**Y**,**1**):-**start**(**4**,**Y**).**

contains**(**X**,**\_**,**2**):-**start**(**X**,**3**).**

contains**(**\_**,**Y**,**3**):-**start**(**0**,**Y**).**

contains**(**X**,**\_**,**4**):-**start**(**X**,**0**).**

contains**(**X**,**Y**,**5**):-**N is Y**-**4**+**X**,** start**(**4**,**N**).**

contains**(**X**,**Y**,**6**):-**N is X**-**3**+**Y**,** start**(**N**,**3**).**

contains**(**X**,**Y**,**7**):-**N is X**+**Y**,** start**(**N**,**0**).**

contains**(**X**,**Y**,**8**):-**N is X**+**Y**,** start**(**0**,**N**).**

main**():-write(**' Water Jug Game \n'**),**

**write(**'Intial State: 4lit Jug- 0lit\n'**),**

**write(**' 3lit Jug- 0lit\n'**),**

**write(**'Final State: 4lit Jug- 2lit\n'**),**

**write(**' 3lit Jug- 0lit\n'**),**

**write(**'Follow the Rules: \n'**),**

**write(**'Rule 1: Fill 4lit Jug\n'**),**

**write(**'Rule 2: Fill 3lit Jug\n'**),**

**write(**'Rule 3: Empty 4lit Jug\n'**),**

**write(**'Rule 4: Empty 3lit Jug\n'**),**

**write(**'Rule 5: Pour water from 3lit Jug to fill 4lit Jug\n'**),**

**write(**'Rule 6: Pour water from 4lit Jug to fill 3lit Jug\n'**),**

**write(**'Rule 7: Pour all of water from 3lit Jug to 4lit Jug\n'**),**

**write(**'Rule 8: Pour all of water from 4lit Jug to 3lit Jug\n'**),**

**write(**' 4lit Jug: 0 | 3lit Jug: 0'**),**nl**,**

**write(**' Enter the move::'**),**

**read(**N**),**nl**,**

contains**(**0**,**0**,**N**).**

|  |
| --- |
| ?- [water\_jug].  ?- main().  Water Jug Game  Intial State: 4lit Jug- 0lit  3lit Jug- 0lit  Final State: 4lit Jug- 2lit  3lit Jug- 0lit  Follow the Rules:  Rule 1: Fill 4lit Jug  Rule 2: Fill 3lit Jug  Rule 3: Empty 4lit Jug  Rule 4: Empty 3lit Jug  Rule 5: Pour water from 3lit Jug to fill 4lit Jug  Rule 6: Pour water from 4lit Jug to fill 3lit Jug  Rule 7: Pour all of water from 3lit Jug to 4lit Jug  Rule 8: Pour all of water from 4lit Jug to 3lit Jug  4lit Jug: 0 | 3lit Jug: 0  Enter the move::1.  4lit Jug: 4| 3lit Jug: 0|  Enter the move::|: 6.  4lit Jug: 1| 3lit Jug: 3|  Enter the move::|: 4.  4lit Jug: 1| 3lit Jug: 0|  Enter the move::|: 8.  4lit Jug: 0| 3lit Jug: 1|  Enter the move::|: 1.  4lit Jug: 4| 3lit Jug: 1|  Enter the move::|: 6.  4lit Jug: 2| 3lit Jug: 3|  Enter the move::|: 4.  4lit Jug: 2 | 3lit Jug: 0|  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  Goal Reached! Congrats!!  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  true . |

AIM: Solve 8 Puzzle Problem using A\* Algorithm in any programming Language.

**8-puzzle Problem**

* It is 3x3 matrix with 8 square blocks containing 1 to 8 and a blank square. The main idea of 8 puzzle is to reorder these squares into a numerical order of 1 to 8 and last square as blank. Each of the squares adjacent to blank block can move up, down, left or right depending on the edges of the matrix.

**A\* Algorithm**

* A\* is a recursive algorithm that calls itself until a solution is found. In this algorithm we consider two heuristic functions, misplaced tiles heuristic and manhattan distance heuristic. Misplaced tiles heuristic calculates the misplaced number of tiles of the current state as compared to the goal state. Manhattan distance heuristic function measures the least steps needed for each of the tiles in the 8-puzzle initial or current state to arrive to the goal state position. In this project we have implemented the state space generation using both the heuristics.
* We are also calculating g(n) which is a measure of step cost for each move made from current state to next state, initially it is set to zero. For each of the heuristic we have implemented f(n)= g(n)+h(n), where g(n) is step cost and h(n) is the heuristic function used. Each of the states is explored using priority queue, which stores the position and f(n) value as key value pair. Then using merge sort technique priority queue is sorted and the next node to be explored is selected based on the least f(n) value. The program comes to an end if the A\* algorithm has not found an optimal path within a runtime limit of two minutes.

**from** copy **import** deepcopy

**import** numpy **as** np

**import** time

# takes the input of current states and evaluvates the best path to goal state

**def** bestsolution**(**state**):**

bestsol **=** np**.**array**([],** **int).**reshape**(-**1**,** 9**)**

count **=** **len(**state**)** **-** 1

**while** count **!=** **-**1**:**

bestsol **=** np**.**insert**(**bestsol**,** 0**,** state**[**count**][**'puzzle'**],** 0**)**

count **=** **(**state**[**count**][**'parent'**])**

**return** bestsol**.**reshape**(-**1**,** 3**,** 3**)**

# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.

**def** all**(**checkarray**):**

**set=[]**

**for** it **in** **set:**

**for** checkarray **in** it**:**

**return** 1

**else:**

**return** 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state

**def** manhattan**(**puzzle**,** goal**):**

a **=** **abs(**puzzle **//** 3 **-** goal **//** 3**)**

b **=** **abs(**puzzle **%** 3 **-** goal **%** 3**)**

mhcost **=** a **+** b

**return** **sum(**mhcost**[**1**:])**

# will calcuates the number of misplaced tiles in the current state as compared to the goal state

**def** misplaced\_tiles**(**puzzle**,**goal**):**

mscost **=** np**.sum(**puzzle **!=** goal**)** **-** 1

**return** mscost **if** mscost **>** 0 **else** 0

#3[on\_true] if [expression] else [on\_false]

# will indentify the coordinates of each of goal or initial state values

**def** coordinates**(**puzzle**):**

pos **=** np**.**array**(range(**9**))**

**for** p**,** q **in** **enumerate(**puzzle**):**

pos**[**q**]** **=** p

**return** pos

# start of 8 puzzle evaluvation, using Manhattan heuristics

**def** evaluvate**(**puzzle**,** goal**):**

steps **=** np**.**array**([(**'up'**,** **[**0**,** 1**,** 2**],** **-**3**),(**'down'**,** **[**6**,** 7**,** 8**],** 3**),(**'left'**,** **[**0**,** 3**,** 6**],** **-**1**),(**'right'**,** **[**2**,** 5**,** 8**],** 1**)],**

dtype **=** **[(**'move'**,** **str,** 1**),(**'position'**,** **list),(**'head'**,** **int)])**

dtstate **=** **[(**'puzzle'**,** **list),(**'parent'**,** **int),(**'gn'**,** **int),(**'hn'**,** **int)]**

# initializing the parent, gn and hn, where hn is manhattan distance function call

costg **=** coordinates**(**goal**)**

parent **=** **-**1

gn **=** 0

hn **=** manhattan**(**coordinates**(**puzzle**),** costg**)**

state **=** np**.**array**([(**puzzle**,** parent**,** gn**,** hn**)],** dtstate**)**

# We make use of priority queues with position as keys and fn as value.

dtpriority **=** **[(**'position'**,** **int),(**'fn'**,** **int)]**

priority **=** np**.**array**(** **[(**0**,** hn**)],** dtpriority**)**

**while** 1**:**

priority **=** np**.**sort**(**priority**,** kind**=**'mergesort'**,** order**=[**'fn'**,** 'position'**])**

position**,** fn **=** priority**[**0**]**

priority **=** np**.**delete**(**priority**,** 0**,** 0**)**

# sort priority queue using merge sort,the first element is picked for exploring remove from queue what we are exploring

puzzle**,** parent**,** gn**,** hn **=** state**[**position**]**

puzzle **=** np**.**array**(**puzzle**)**

# Identify the blank square in input

blank **=** **int(**np**.**where**(**puzzle **==** 0**)[**0**])**

gn **=** gn **+** 1

c **=** 1

start\_time **=** time**.**time**()**

**for** s **in** steps**:**

c **=** c **+** 1

**if** blank **not** **in** s**[**'position'**]:**

# generate new state as copy of current

openstates **=** deepcopy**(**puzzle**)**

openstates**[**blank**],** openstates**[**blank **+** s**[**'head'**]]** **=** openstates**[**blank **+** s**[**'head'**]],** openstates**[**blank**]**

# The all function is called, if the node has been previously explored or not

**if** **~(**np**.all(list(**state**[**'puzzle'**])** **==** openstates**,** 1**)).any():**

end\_time **=** time**.**time**()**

**if** **((** end\_time **-** start\_time **)** **>** 2**):**

**print(**" The 8 puzzle is unsolvable ! \n"**)**

**exit**

# calls the manhattan function to calcuate the cost

hn **=** manhattan**(**coordinates**(**openstates**),** costg**)**

# generate and add new state in the list

q **=** np**.**array**([(**openstates**,** position**,** gn**,** hn**)],** dtstate**)**

state **=** np**.**append**(**state**,** q**,** 0**)**

# f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state

fn **=** gn **+** hn

q **=** np**.**array**([(len(**state**)** **-** 1**,** fn**)],** dtpriority**)**

priority **=** np**.**append**(**priority**,** q**,** 0**)**

# Checking if the node in openstates are matching the goal state.

**if** np**.**array\_equal**(**openstates**,** goal**):**

**print(**' The 8 puzzle is solvable ! \n'**)**

**return** state**,** **len(**priority**)**

**return** state**,** **len(**priority**)**

# start of 8 puzzle evaluvation, using Misplaced tiles heuristics

**def** evaluvate\_misplaced**(**puzzle**,** goal**):**

steps **=** np**.**array**([(**'up'**,** **[**0**,** 1**,** 2**],** **-**3**),(**'down'**,** **[**6**,** 7**,** 8**],** 3**),(**'left'**,** **[**0**,** 3**,** 6**],** **-**1**),(**'right'**,** **[**2**,** 5**,** 8**],** 1**)],**

dtype **=** **[(**'move'**,** **str,** 1**),(**'position'**,** **list),(**'head'**,** **int)])**

dtstate **=** **[(**'puzzle'**,** **list),(**'parent'**,** **int),(**'gn'**,** **int),(**'hn'**,** **int)]**

costg **=** coordinates**(**goal**)**

# initializing the parent, gn and hn, where hn is misplaced\_tiles function call

parent **=** **-**1

gn **=** 0

hn **=** misplaced\_tiles**(**coordinates**(**puzzle**),** costg**)**

state **=** np**.**array**([(**puzzle**,** parent**,** gn**,** hn**)],** dtstate**)**

# We make use of priority queues with position as keys and fn as value.

dtpriority **=** **[(**'position'**,** **int),(**'fn'**,** **int)]**

priority **=** np**.**array**([(**0**,** hn**)],** dtpriority**)**

**while** 1**:**

priority **=** np**.**sort**(**priority**,** kind**=**'mergesort'**,** order**=[**'fn'**,** 'position'**])**

position**,** fn **=** priority**[**0**]**

# sort priority queue using merge sort,the first element is picked for exploring.

priority **=** np**.**delete**(**priority**,** 0**,** 0**)**

puzzle**,** parent**,** gn**,** hn **=** state**[**position**]**

puzzle **=** np**.**array**(**puzzle**)**

# Identify the blank square in input

blank **=** **int(**np**.**where**(**puzzle **==** 0**)[**0**])**

# Increase cost g(n) by 1

gn **=** gn **+** 1

c **=** 1

start\_time **=** time**.**time**()**

**for** s **in** steps**:**

c **=** c **+** 1

**if** blank **not** **in** s**[**'position'**]:**

# generate new state as copy of current

openstates **=** deepcopy**(**puzzle**)**

openstates**[**blank**],** openstates**[**blank **+** s**[**'head'**]]** **=** openstates**[**blank **+** s**[**'head'**]],** openstates**[**blank**]**

# The check function is called, if the node has been previously explored or not.

**if** **~(**np**.all(list(**state**[**'puzzle'**])** **==** openstates**,** 1**)).any():**

end\_time **=** time**.**time**()**

**if** **((** end\_time **-** start\_time **)** **>** 2**):**

**print(**" The 8 puzzle is unsolvable \n"**)**

**break**

# calls the Misplaced\_tiles function to calcuate the cost

hn **=** misplaced\_tiles**(**coordinates**(**openstates**),** costg**)**

# generate and add new state in the list

q **=** np**.**array**([(**openstates**,** position**,** gn**,** hn**)],** dtstate**)**

state **=** np**.**append**(**state**,** q**,** 0**)**

# f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state

fn **=** gn **+** hn

q **=** np**.**array**([(len(**state**)** **-** 1**,** fn**)],** dtpriority**)**

priority **=** np**.**append**(**priority**,** q**,** 0**)**

# Checking if the node in openstates are matching the goal state.

**if** np**.**array\_equal**(**openstates**,** goal**):**

**print(**' The 8 puzzle is solvable \n'**)**

**return** state**,** **len(**priority**)**

**return** state**,** **len(**priority**)**

# ---------- Program start -----------------

# User input for initial state

puzzle **=** **[]**

**print(**" Input vals from 0-8 for start state "**)**

**for** i **in** **range(**0**,**9**):**

x **=** **int(input(**"enter vals :"**))**

puzzle**.**append**(**x**)**

# User input of goal state

goal **=** **[]**

**print(**" Input vals from 0-8 for goal state "**)**

**for** i **in** **range(**0**,**9**):**

x **=** **int(input(**"Enter vals :"**))**

goal**.**append**(**x**)**

#n = int(input("1. Manhattan distance \n2. Misplaced tiles"))

n**=**2

**if(**n **==**1 **):**

state**,** visited **=** evaluvate**(**puzzle**,** goal**)**

bestpath **=** bestsolution**(**state**)**

**print(str(**bestpath**).**replace**(**'['**,** ' '**).**replace**(**']'**,** ''**))**

totalmoves **=** **len(**bestpath**)** **-** 1

**print(**'Steps to reach goal:'**,**totalmoves**)**

visit **=** **len(**state**)** **-** visited

**print(**'Total nodes visited: '**,**visit**,** "\n"**)**

**print(**'Total generated:'**,** **len(**state**))**

**if(**n **==** 2**):**

state**,** visited **=** evaluvate\_misplaced**(**puzzle**,** goal**)**

bestpath **=** bestsolution**(**state**)**

**print(str(**bestpath**).**replace**(**'['**,** ' '**).**replace**(**']'**,** ''**))**

totalmoves **=** **len(**bestpath**)** **-** 1

**print(**'Steps to reach goal:'**,**totalmoves**)**

visit **=** **len(**state**)** **-** visited

**print(**'Total nodes visited: '**,**visit**,** "\n"**)**

**print(**'Total generated:'**,** **len(**state**))**

|  |
| --- |
| F:\Windowcmd\python exam practice>python "8 puzzle problem.py"  Input vals from 0-8 for start state  enter vals :1  enter vals :2  enter vals :3  enter vals :0  enter vals :4  enter vals :5  enter vals :6  enter vals :7  enter vals :8  Input vals from 0-8 for goal state  Enter vals :1  Enter vals :2  Enter vals :3  Enter vals :4  Enter vals :5  Enter vals :6  Enter vals :7  Enter vals :8  Enter vals :0  The 8 puzzle is solvable  1 2 3  0 4 5  6 7 8  1 2 3  4 0 5  6 7 8  1 2 3  4 5 0  6 7 8  1 2 3  4 5 8  6 7 0  1 2 3  4 5 8  6 0 7  1 2 3  4 5 8  0 6 7  1 2 3  0 5 8  4 6 7  1 2 3  5 0 8  4 6 7  1 2 3  5 6 8  4 0 7  1 2 3  5 6 8  4 7 0  1 2 3  5 6 0  4 7 8  1 2 3  5 0 6  4 7 8  1 2 3  0 5 6  4 7 8  1 2 3  4 5 6  0 7 8  1 2 3  4 5 6  7 0 8  1 2 3  4 5 6  7 8 0  Steps to reach goal: 15  Total nodes visited: 495  Total generated: 795 |

**AIM:** Convert the given PROLOG predicates into Semantic Net.

cat(tom).

cat(cat1).

mat(mat1).

sat\_on(cat1,mat1).

bird(bird1).

caught(tom,bird1).

like(X,cream) :– cat(X).

mammal(X) :– cat(X).

has(X,fur) :– mammal(X).

animal(X) :– mammal(X).

animal(X) :– bird(X).

owns(john,tom).

is\_coloured(tom,ginger).

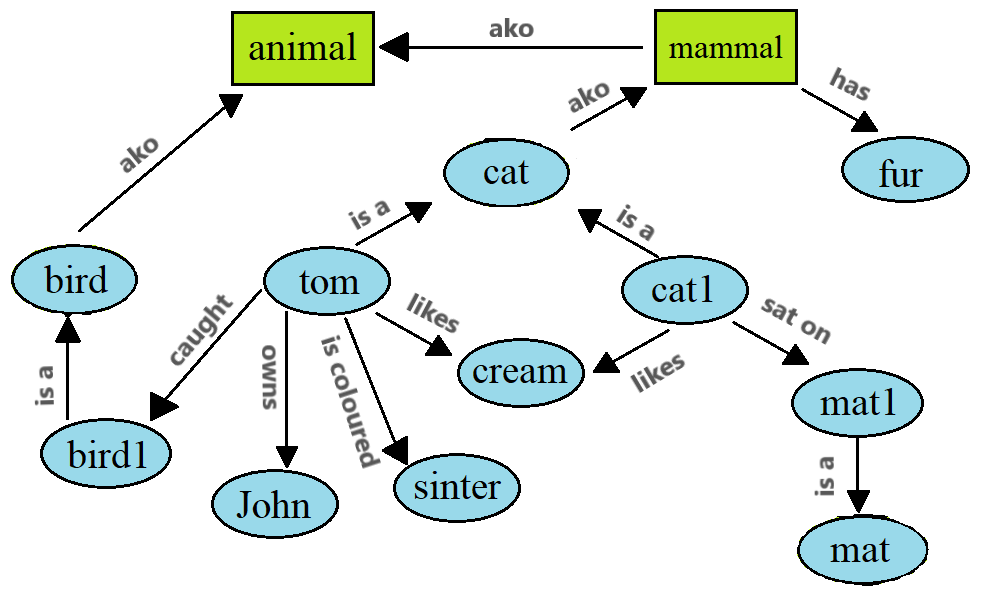


Fig - 1

**AIM:** Write the Conceptual Dependency for the given statements.

(a) John gives Mary a book

(b)John gave Mary the book yesterday.

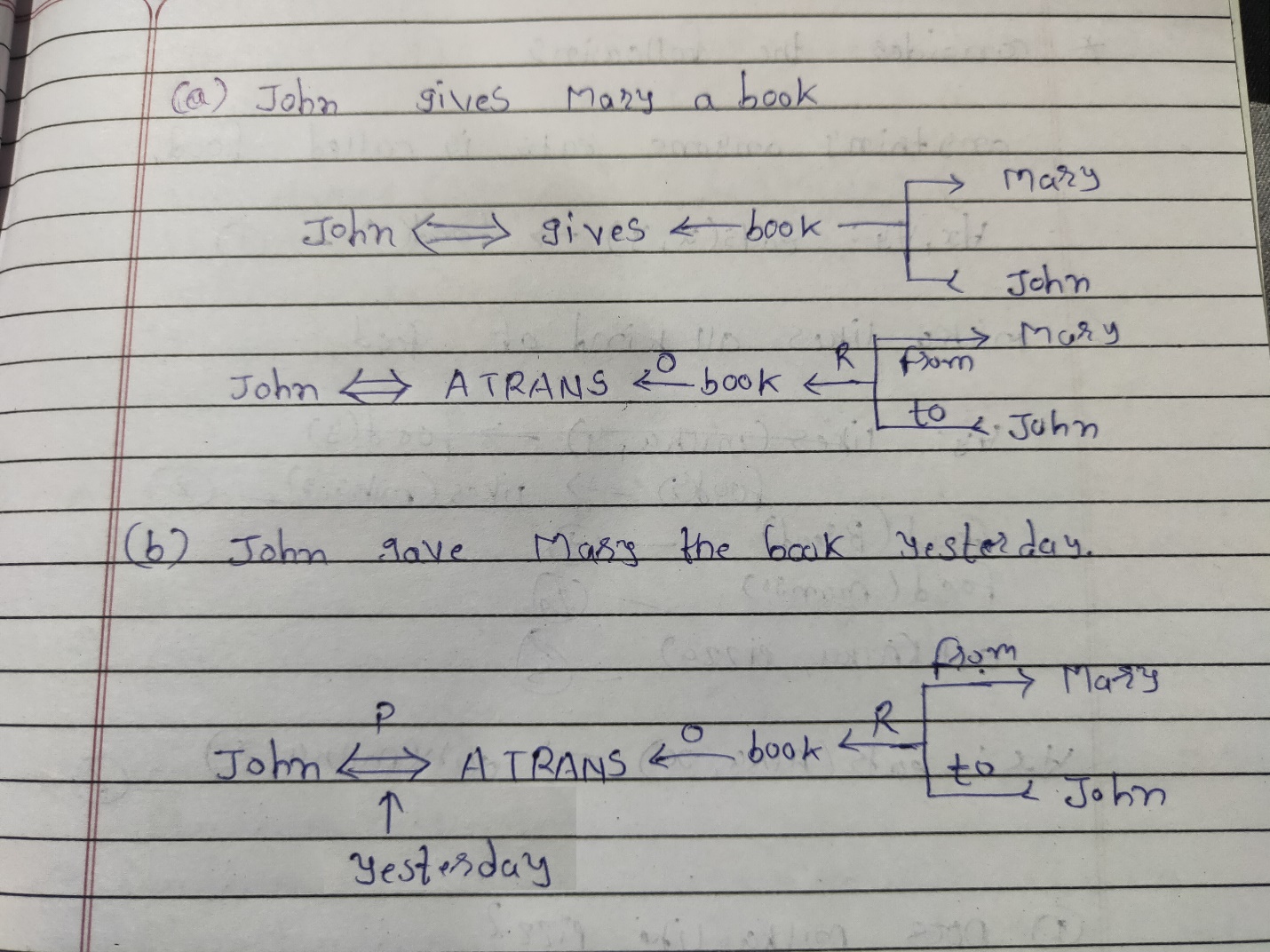


Fig - 1

**AIM**: Implement Bayesian Networks algorithm.

What Is A Bayesian Network?

A Bayesian Network falls under the category of Probabilistic Graphical Modelling (PGM) technique that is used to compute uncertainties by using the concept of probability. Popularly known as Belief Networks, Bayesian Networks are used to model uncertainties by using Directed Acyclic Graphs (DAG).

Math behind Bayesian Networks

As mentioned earlier, Bayesian models are based on the simple concept of probability. So let’s understand what conditional probability and Joint probability distribution mean.

What Is Joint Probability?

* Joint Probability is a statistical measure of two or more events happening at the same time, i.e., P (A, B, C), the probability of event A, B and C occurring. It can be represented as the probability of the intersection two or more events occurring.

What Is Conditional Probability?

Conditional Probability of an event X is the probability that the event will occur given that an event Y has already occurred.

P (X| Y) is the probability of event X occurring, given that event, Y occurs.

If X and Y are dependent events then the expression for conditional probability is given by: P (X| Y) = P (X and Y) / P (Y)

If A and B are independent events then the expression for conditional probability is given by: P (X| Y) = P (X)

To learn more about the concepts of statistics and probability, you can go through this, All You Need to Know about Statistics and Probability blog.

Now let’s take example of Bayesian Network that will model the marks (m) of a student on his examination. The marks will depend on:

Exam level (e): This is a discrete variable that can take two values, (difficult, easy)

IQ of the student (i): A discrete variable that can take two values (high, low)

The marks will intern predict whether or not he/she will get admitted (a) to a university.

The IQ will also predict the aptitude score (s) of the student.

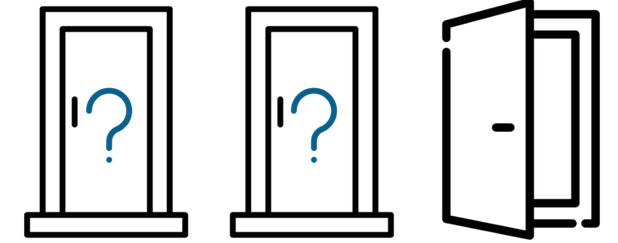
With this information, we can build a Bayesian Network that will model the performance of a student on an exam. The Bayesian Network can be represented as a DAG where each node denotes a variable that predicts the performance of the student.

Bayesian Networks Python

We’ll be using Bayesian Networks to solve the famous Monty Hall Problem.

The Monty Hall problem named after the host of the TV series, ‘Let’s Make A Deal’, is a paradoxical probability puzzle that has been confusing people for over a decade.

So this is how it works. The game involves three doors, given that behind one of these doors is a car and the remaining two have goats behind them. So you start by picking a random door, say #2. On the other hand, the host knows where the car is hidden and he opens another door, say #1 (behind which there is a goat). Here’s the catch, you’re now given a choice, the host will ask you if you want to pick door #3 instead of your first choice i.e. #2.



Is it better if you switch your choice or should you stick to your first choice?

This is exactly what we’re going to model. We’ll be creating a Bayesian Network to understand the probability of winning if the participant decides to switch his choice.

Let’s understand the dependencies here, the door selected by the guest and the door containing the car are completely random processes. However, the door Monty chooses to open is dependent on both the doors; the door selected by the guest, and the door the prize is behind. Monty has to choose in such a way that the door does not contain the prize and it cannot be the one chosen by the guest.

|  |
| --- |
| import math  from pomegranate import \*    # Initially the door selected by the guest is completely random  guest =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )    # The door containing the prize is also a random process  prize =DiscreteDistribution( { 'A': 1./3, 'B': 1./3, 'C': 1./3 } )    # The door Monty picks, depends on the choice of the guest and the prize door  monty =ConditionalProbabilityTable(  [[ 'A', 'A', 'A', 0.0 ],  [ 'A', 'A', 'B', 0.5 ],  [ 'A', 'A', 'C', 0.5 ],  [ 'A', 'B', 'A', 0.0 ],  [ 'A', 'B', 'B', 0.0 ],  [ 'A', 'B', 'C', 1.0 ],  [ 'A', 'C', 'A', 0.0 ],  [ 'A', 'C', 'B', 1.0 ],  [ 'A', 'C', 'C', 0.0 ],  [ 'B', 'A', 'A', 0.0 ],  [ 'B', 'A', 'B', 0.0 ],  [ 'B', 'A', 'C', 1.0 ],  [ 'B', 'B', 'A', 0.5 ],  [ 'B', 'B', 'B', 0.0 ],  [ 'B', 'B', 'C', 0.5 ],  [ 'B', 'C', 'A', 1.0 ],  [ 'B', 'C', 'B', 0.0 ],  [ 'B', 'C', 'C', 0.0 ],  [ 'C', 'A', 'A', 0.0 ],  [ 'C', 'A', 'B', 1.0 ],  [ 'C', 'A', 'C', 0.0 ],  [ 'C', 'B', 'A', 1.0 ],  [ 'C', 'B', 'B', 0.0 ],  [ 'C', 'B', 'C', 0.0 ],  [ 'C', 'C', 'A', 0.5 ],  [ 'C', 'C', 'B', 0.5 ],  [ 'C', 'C', 'C', 0.0 ]], [guest, prize] )    d1 = State( guest, name="guest" )  d2 = State( prize, name="prize" )  d3 = State( monty, name="monty" )    #Building the Bayesian Network  network = BayesianNetwork( "Solving the Monty Hall Problem With Bayesian Networks" )  network.add\_states(d1, d2, d3)  network.add\_edge(d1, d3)  network.add\_edge(d2, d3)  network.bake()  beliefs = network.predict\_proba({ 'guest' : 'A' })  beliefs = map(str, beliefs)  print("n".join( "{}t{}".format( state.name, belief ) for state, belief in zip( network.states, beliefs ) ))  beliefs = network.predict\_proba({'guest' : 'A', 'monty' : 'B'})  print("n".join( "{}t{}".format( state.name, str(belief) ) for state, belief in zip( network.states, beliefs ))) |

In the above code ‘A’, ‘B’, ‘C’, represent the doors picked by the guest, prize door and the door picked by Monty respectively. Here we’ve drawn out the conditional probability for each of the nodes. Since the prize door and the guest door are picked randomly there isn’t much to consider. However, the door picked by Monty depends on the other two doors, therefore in the above code, I’ve drawn out the conditional probability considering all possible scenarios.

The next step is to make predictions using this model. One of the strengths of Bayesian networks is their ability to infer the values of arbitrary ‘hidden variables’ given the values from ‘observed variables.’ These hidden and observed variables do not need to be specified beforehand, and the more variables which are observed the better the inference will be on the hidden variables.

We’ve assumed that the guest picks door ‘A’. Given this information, the probability of the prize door being ‘A’, ‘B’, ‘C’ is equal (1/3) since it is a random process. However, the probability of Monty picking ‘A’ is obviously zero since the guest picked door ‘A’. And the other two doors have a 50% chance of being picked by Monty since we don’t know which the prize door is.

Notice the output, the probability of the car being behind door ‘C’ is approx. 66%. This proves that if the guest switches his choice, he has a higher probability of winning. Though this might seem confusing to some of you, it’s a known fact that:

Guests who decided to switch doors won about 2/3 of the time

Guests who refused to switch won about 1/3 of the time

Bayesian Networks are used in such cases that involve predicting uncertain tasks and outcomes. In the below section you’ll understand how Bayesian Networks can be used to solve more such problems.

|  |
| --- |
| F:\work of pro\SEM7\3170316-Artificial intelligence>python monty.py  guesttAnprizet{  "class" : "Distribution",  "dtype" : "str",  "name" : "DiscreteDistribution",  "parameters" : [  {  "A" : 0.3333333333333333,  "B" : 0.3333333333333333,  "C" : 0.3333333333333333  }  ],  "frozen" : false  }nmontyt{  "class" : "Distribution",  "dtype" : "str",  "name" : "DiscreteDistribution",  "parameters" : [  {  "C" : 0.49999999999999983,  "A" : 0.0,  "B" : 0.49999999999999983  }  ],  "frozen" : false  }  guesttAnprizet{  "class" : "Distribution",  "dtype" : "str",  "name" : "DiscreteDistribution",  "parameters" : [  {  "A" : 0.3333333333333334,  "B" : 0.0,  "C" : 0.6666666666666664  }  ],  "frozen" : false  }nmontytB |

**Aim:** Implement Min Max Algorithm for any problem.

from sys import maxsize

class Node(object):

def \_\_init\_\_(self, i\_depth, i\_playernum, i\_stickRemaining, i\_value=0):

self.i\_depth = i\_depth

self.i\_playernum = i\_playernum

self.i\_stickRemaining = i\_stickRemaining

self.i\_value = i\_value

self.children = []

self.CreateChildren()

def CreateChildren(self):

if(self.i\_depth >= 0):

for i in range(1, 3):

v = self.i\_stickRemaining - i

self.children.append( Node(self.i\_depth - 1, self.i\_playernum, v, self.RealVal(v)))

def RealVal(self, val):

if val == 0:

return maxsize\*self.i\_playernum

elif val < 0:

return maxsize\*-self.i\_playernum

else:

return 0

def MinMax(node, i\_depth, i\_playernum):

if i\_depth == 0 or abs(node.i\_value) == maxsize:

return node.i\_value

i\_bestvalue = maxsize\*-i\_playernum

for i in range(len(node.children)):

child = node.children[i]

i\_val = MinMax(child, i\_depth-1, -i\_playernum)

if abs(maxsize \* i\_playernum - i\_val) < abs(maxsize \* i\_playernum \* i\_bestvalue):

i\_bestvalue = i\_val

return i\_bestvalue

def WinCheck(i\_sticks, i\_playernum):

if i\_sticks <= 0:

print("\*"\*30)

if i\_playernum > 0:

if i\_sticks == 0:

print("\tYou Win!!")

else:

print("\tYou Lose [~.~]")

else:

if i\_sticks == 0:

print("\tComp Wins [~.~]!!")

else:

print("\tComp ERROR!!")

print("\*"\*30)

return 0

return 1

i\_stickCount = 11

i\_depth = 4

i\_curPlayer = 1

print("INSTRUCTIONS: Be the Player to Pick up the Last Stick \@/")

print("\t\t\t\tYou can pick up 1 or 2 Sticks at a time.")

while(i\_stickCount > 0):

print("\n%d Sticks Remains. How many would you like to pick up??"%(i\_stickCount), end = " ")

i\_choice = input("1 or 2 ?: ")

i\_stickCount -= int(float(i\_choice))

if WinCheck(i\_stickCount, i\_curPlayer):

i\_curPlayer \*= -1

node = Node(i\_depth, i\_curPlayer, i\_stickCount)

bestChoice = -100

i\_bestvalue = -i\_curPlayer \* maxsize

for i in range(len(node.children)):

n\_child = node.children[i]

i\_val = MinMax(n\_child, i\_depth, -i\_curPlayer)

if abs(i\_curPlayer \* maxsize - i\_val) <= abs(i\_curPlayer \* maxsize - i\_bestvalue):

i\_bestvalue = i\_val

bestChoice = i

bestChoice += 1

print("Comp Choses: ", str(bestChoice), "\tBased on the value: ", str(i\_bestvalue))

i\_stickCount -= bestChoice

WinCheck(i\_stickCount, i\_curPlayer)

i\_curPlayer \*= -1

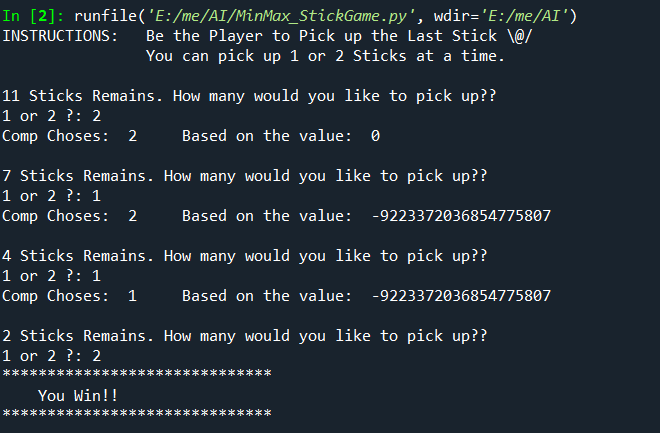


Fig 7.1

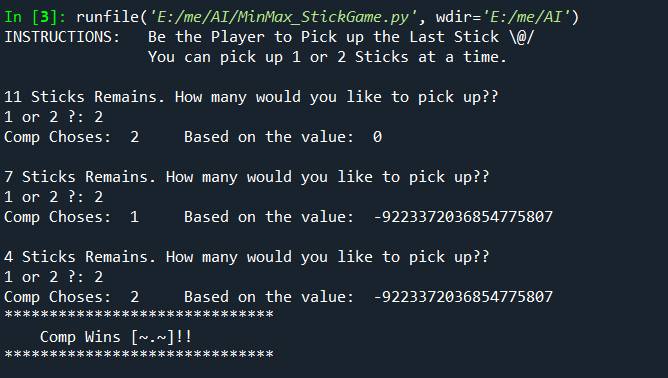


Fig 7.2

**AIM:** Demonstrate Connectionist Model using Tool.

**CONNECTIONIST MODELS**

In contrast to the symbolist architectures in which the mind is assumed to be a physical symbol-processing system, connectionist systems are networks of large numbers of interconnected “units.” Each unit can have associated with it a certain amount of activation. Connections to other units are given explicit weights (including negative weights). Activation spreads from one unit to another as a function of the weighted links. For example, the function of a typical link might be to multiply the input activation by its weight and then apply a threshold function. A typical unit would sum all of its input activations, then divide this among all its links. The weights on the links are adjustable with experience. Some of the links may represent sensory inputs from the outside world; some may represent output to effectors to the outside world. Units in connectionist models are usually taken to be below the level of a symbol. For example, different units may represent visual features of a letter such as verticalness or roundedness.

**Neural networks are by far the most commonly used connectionist model today.** Though there are a large variety of neural network models, they almost always follow two basic principles regarding the mind:

Any mental state can be described as an (N)-dimensional vector of numeric activation values over neural units in a network.

Memory is created by modifying the strength of the connections between neural units. The connection strengths, or "weights", are generally represented as an N×N matrix.

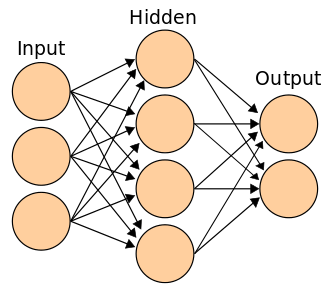


Fig 8.1 - Connectionist (ANN) model with a hidden layer

**Exploring scikit-learn and tenserflow for Visualization of MLP weights on MNIST**

Sometimes looking at the learned coefficients of a neural network can provide insight into the learning behavior. For example if weights look unstructured, maybe some were not used at all, or if very large coefficients exist, maybe regularization was too low or the learning rate too high.

This example shows how to plot some of the first layer weights in a MLPClassifier trained on the MNIST dataset.

The input data consists of 28x28 pixel handwritten digits, leading to 784 features in the dataset. Therefore the first layer weight matrix have the shape (784, hidden\_layer\_sizes[0]). We can therefore visualize a single column of the weight matrix as a 28x28 pixel image.

To make the example run faster, we use very few hidden units, and train only for a very short time. Training longer would result in weights with a much smoother spatial appearance. The example will throw a warning because it doesn’t converge, in this case this is what we want because of CI’s time constraints.

import warnings

import matplotlib.pyplot as plt

from sklearn.datasets import [fetch\_openml](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_openml.html#sklearn.datasets.fetch_openml)

from sklearn.exceptions import [ConvergenceWarning](https://scikit-learn.org/stable/modules/generated/sklearn.exceptions.ConvergenceWarning.html#sklearn.exceptions.ConvergenceWarning)

from sklearn.neural\_network import [MLPClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier)

print(\_\_doc\_\_)

# Load data from https://www.openml.org/d/554

X, y = [fetch\_openml](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_openml.html#sklearn.datasets.fetch_openml)('mnist\_784', version=1, return\_X\_y=True)

X = X / 255.

# rescale the data, use the traditional train/test split

X\_train, X\_test = X[:60000], X[60000:]

y\_train, y\_test = y[:60000], y[60000:]

mlp = [MLPClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier)(hidden\_layer\_sizes=(50,), max\_iter=10, alpha=1e-4,

solver='sgd', verbose=10, random\_state=1,

learning\_rate\_init=.1)

# this example won't converge because of CI's time constraints, so we catch the

# warning and are ignore it here

with [warnings.catch\_warnings](https://docs.python.org/3/library/warnings.html#warnings.catch_warnings)():

[warnings.filterwarnings](https://docs.python.org/3/library/warnings.html#warnings.filterwarnings)("ignore", category=[ConvergenceWarning](https://scikit-learn.org/stable/modules/generated/sklearn.exceptions.ConvergenceWarning.html#sklearn.exceptions.ConvergenceWarning),

module="sklearn")

mlp.fit(X\_train, y\_train)

print("Training set score: %f" % mlp.score(X\_train, y\_train))

print("Test set score: %f" % mlp.score(X\_test, y\_test))

fig, axes = [plt.subplots](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html#matplotlib.pyplot.subplots)(4, 4)

# use global min / max to ensure all weights are shown on the same scale

vmin, vmax = mlp.coefs\_[0].min(), mlp.coefs\_[0].max()

for coef, ax in zip(mlp.coefs\_[0].T, axes.ravel()):

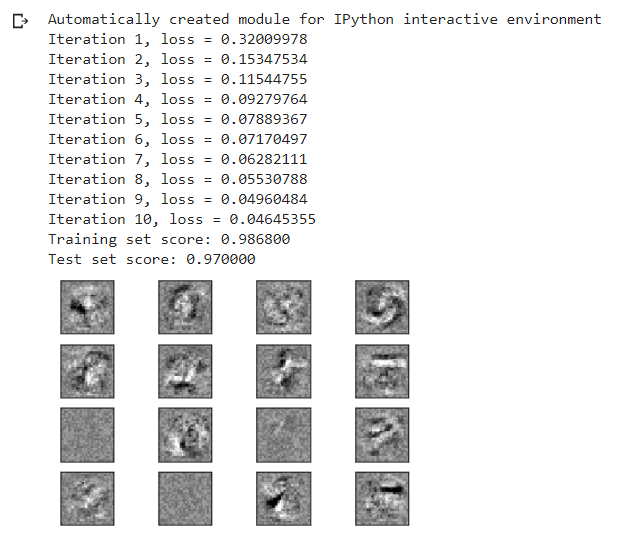
ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 \* vmin,

vmax=.5 \* vmax)

ax.set\_xticks(())

ax.set\_yticks(())

[plt.show](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.show.html#matplotlib.pyplot.show)()



**AIM:** Implement Genetic Algorithm.

we need to **solve the N-Queen problem using a Genetic Algorithm**. We need to use the principle of evolution to find a solution to a problem.

In order to solve the N-Queen problem the following steps are needed:

1) Chromosome design

2) Initialization

3) Fitness evaluation

4) Selection

5) Crossover

6) Mutation

7) Update generation

8) Go back to 3)

**Chromosome design:**

Firstly, we need to create a chromosome representation. For showing a chromosome, the best way is to represent it as a list of length N where in our case N=8. The value of each refers to the row index of each queen. The value of each index is from 1 to 8.

**Initialization:**

In the initialization process, we need to arrange a random population of chromosomes (potential solutions) are created. Here is the initial population, I took 4 chromosomes, each of which has a length 8. They are

[0,1,4,7,3,5,6,2]

[4,5,0,7,6,1,3,2]

[1,7,7,3,4,4,2,5]

[0,0,1,2,4,6,7,3]

**Fitness evaluation:**

Here we are taking N=5 for sake of simplicty.

First of all, the fitness function is pairs of non-attacking queens. So, higher scores are better is better for us. In order to solve the fitness function for the chromosome [5 2 4 3 5], I assigned each queen uniquely as Q1, Q2, Q3, Q4 and Q5. And to find the fitness function value I made the following equation:

Fitness function = F1+F2+F3+F4+F5

where:

F1 = number of pairs of nonattacking queens with queen Q1.

F2 = number of pairs of nonattacking queens with queen Q2.

F3 = number of pairs of nonattacking queens with queen Q3.

F4 = number of pairs of nonattacking queens with queen Q4.

F5 = number of pairs of nonattacking queens with queen Q5.

Here for example if we already counted pair Q1 and Q2 to F1, we should not count the same pair Q2 and Q1 to F2.

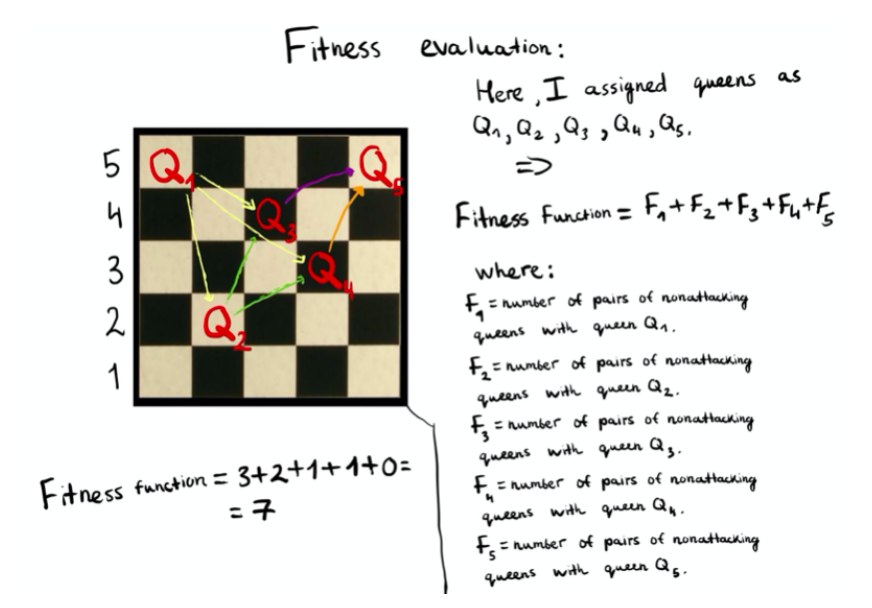


Fig 9.1 – Fitness function

Then we need to compute the probability of being chosen from the fitness function. This will be needed for the next selection step. First, we need to add all fitness functions which will be equal as the following:

[5 2 4 3 5] Fitness Function = 7

[4 3 5 1 4] Fitness Function = 6

[2 1 3 2 4] Fitness Function = 6

[5 2 3 4 1] Fitness Function = 5

7+6+6+5=24

[5 2 4 3 5] probability of being chosen = 7/24 \*100% = 29

**Selection:**

we randomly choose the two pairs to reproduce based on probabilities which we counted on the previous step. In other words certain number of chromosomes will survive into next generator using selection operator. Here selected chromosomes act as parents that are combined using crossover operator to make children. Here we took randomly following chromosomes based on their probabilities:

[4 3 5 1 4]

[5 2 4 3 5]

[4 3 5 1 4]

[2 1 3 2 4]

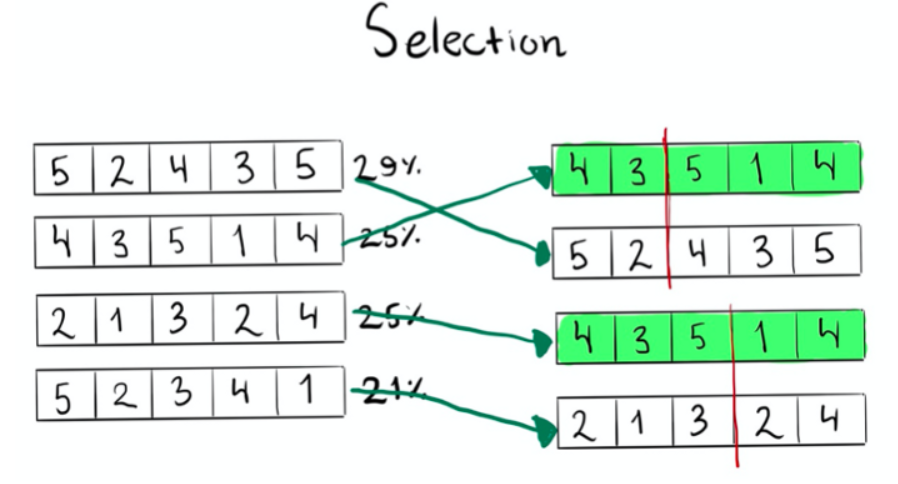


Fig 9.2 Selection

**Crossover:**

Selected chromosomes act as parents that are combined using crossover operator to make children. In other words, it combines the genetic information of two parents to generate new offspring.

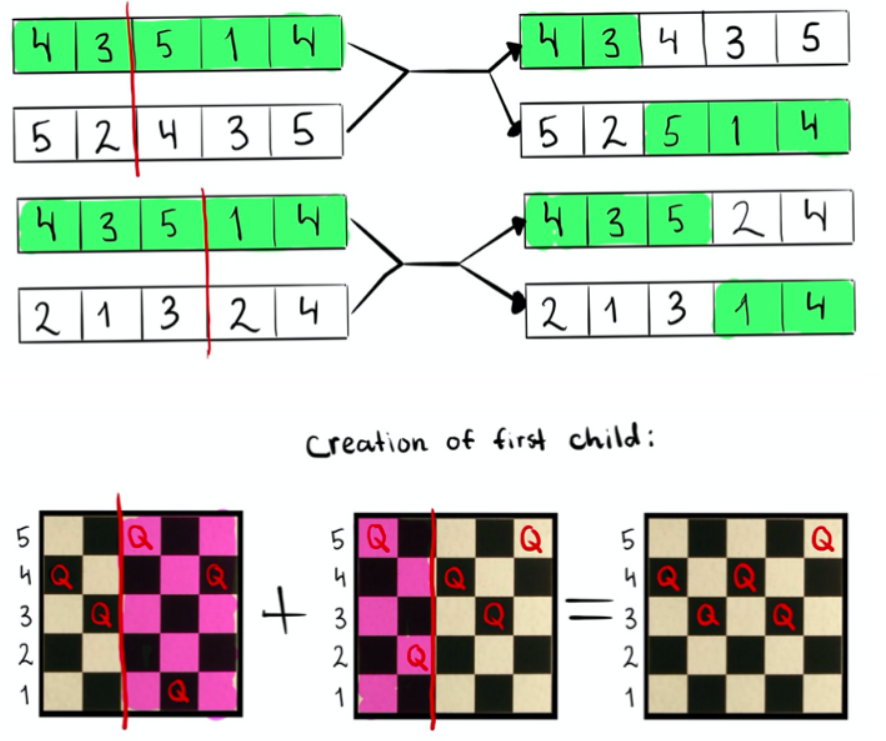


Fig 9.3 Crossover

**Mutation:**

In mutation process we alter one or more gene values in chromosomes which we found after crossover. So it randomly changes few gens and the mutation probability is low.

[4 3 4 3 5] → [4 3 1 3 5]

[5 2 5 1 4] → [5 2 3 1 4]

[4 3 5 2 4] → [4 3 5 2 4]

[2 1 3 1 4] → [2 1 3 5 4]

Where we can notice that the third gene in the chromosome [4 3 4 3 5] changed from 4 to 1. Also third gene in the chromosome [5 2 5 1 4] changed from 5 to 3. In addition to this, fourth gene in the chromosome [2 1 3 1 4] changed from 1 to 5.

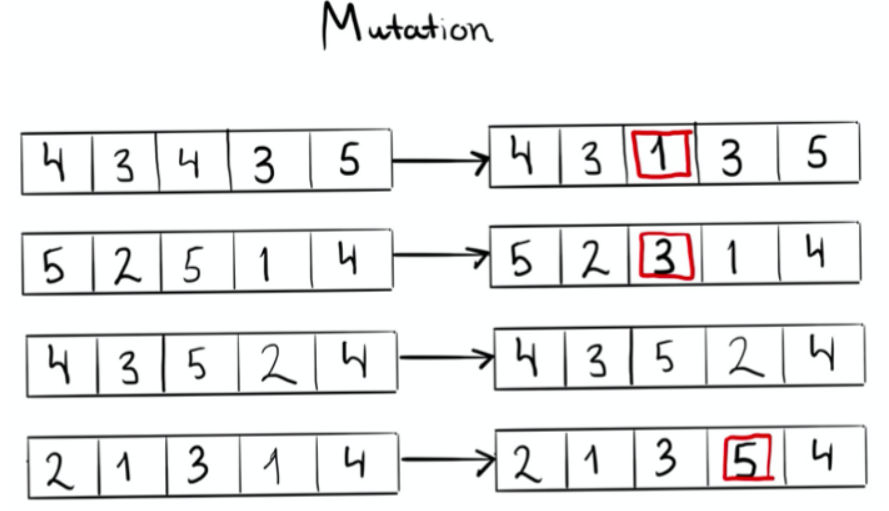


Fig 9.4 Mutation

The evolution of the GA for reaching the optimal solution in which 0 attacks exists.

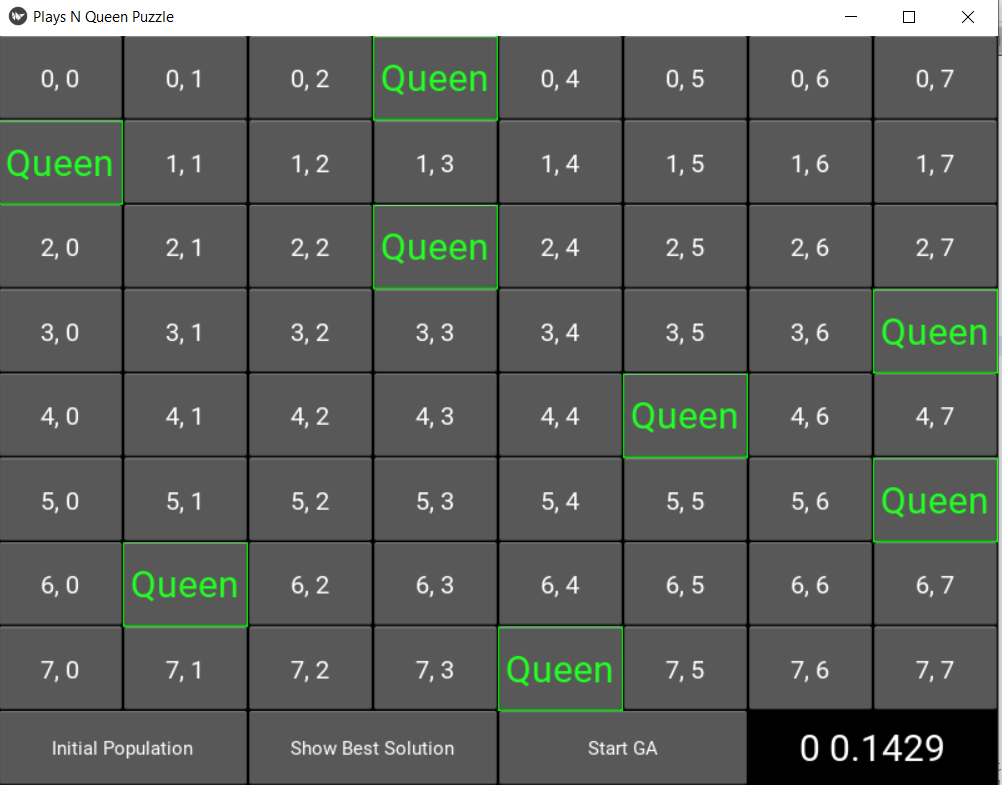


Fig 9.5 – initialization

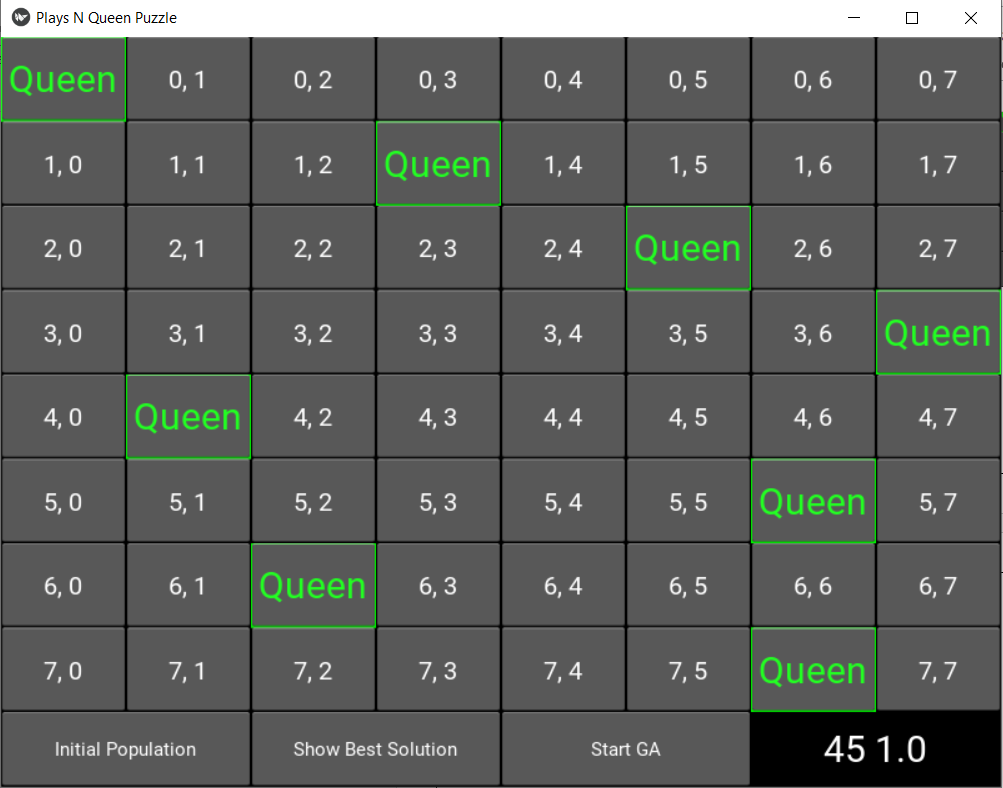


Fig 9.6 after 45 generation

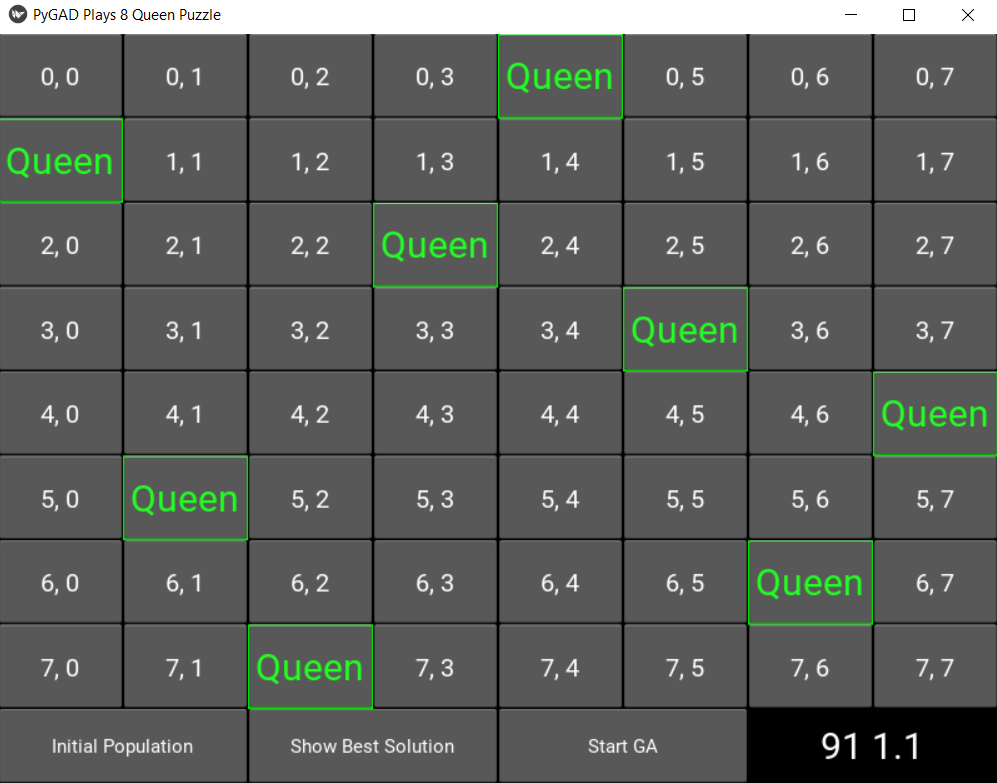


Fig 9.7 – Eight Queen Solution

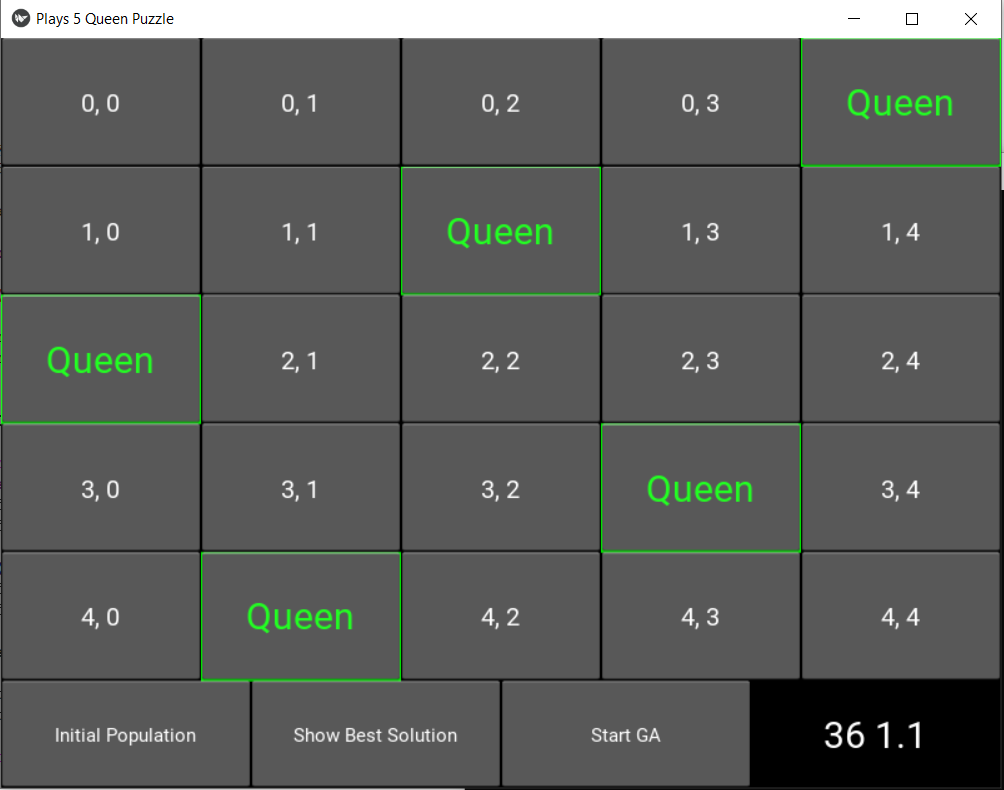


Fig 9.8 – five queen solution



Fig 9.9 – Ten Queen Solution

**import** kivy**.**app

**import** kivy**.**uix**.**gridlayout

**import** kivy**.**uix**.**boxlayout

**import** kivy**.**uix**.**button

**import** kivy**.**uix**.**textinput

**import** kivy**.**uix**.**label

**import** kivy**.**graphics

**import** numpy

**import** pygad

**import** threading

n\_queen **=** **int(**10**)**

n\_queen\_r **=** 10.0

n\_gen **=** **int(**100**)**

n\_queen1 **=** n\_queen**+**1

n\_queen2 **=** n\_queen**+**2

**class** **PygadThread(**threading**.**Thread**):**

**def** \_\_init\_\_**(**self**,** app**,** ga\_instance**):**

**super().**\_\_init\_\_**()**

self**.**ga\_instance **=** ga\_instance

self**.**app **=** app

**def** run**(**self**):**

self**.**ga\_instance**.**run**()**

self**.**ga\_instance**.**plot\_result**()**

**class** **BuzzleApp(**kivy**.**app**.**App**):**

old\_best\_sol\_fitness **=** **-**1

old\_best\_sol\_idx **=** **-**1

**def** start\_ga**(**self**,** **\***args**):**

**if** **(not** **(**"pop\_created" **in** **vars(**self**))** **or** **(**self**.**pop\_created **==** 0**)):**

**print(**"No Population Created Yet. Create the initial Population by Pressing the \"Initial Population\" Button in Order to Call the initialize\_population() Method At First."**)**

self**.**num\_attacks\_Label**.**text **=** "Press \"Initial Population\""

**return**

pygadThread **=** PygadThread**(**self**,** self**.**ga\_instance**)**

pygadThread**.**start**()**

**def** initialize\_population**(**self**,** **\***args**):**

self**.**num\_solutions **=** 100

# print("Number of Solutions within the Population : ", self.num\_solutions)

self**.**reset\_board\_text**()**

self**.**population\_1D\_vector **=** numpy**.**zeros**(**shape**=(**self**.**num\_solutions**,** n\_queen**))** # Each solution is represented as a row in this array. If there are 5 rows, then there are 5 solutions.

# Creating the initial population RANDOMLY as a set of 1D vectors.

**for** solution\_idx **in** **range(**self**.**num\_solutions**):**

initial\_queens\_y\_indices **=** numpy**.**random**.**rand**(**n\_queen**)\***n\_queen

initial\_queens\_y\_indices **=** initial\_queens\_y\_indices**.**astype**(**numpy**.**uint8**)**

self**.**population\_1D\_vector**[**solution\_idx**,** **:]** **=** initial\_queens\_y\_indices

self**.**vector\_to\_matrix**()**

# print("Population 1D Vectors : ", self.population\_1D\_vector)

# print("Population 2D Matrices : ", self.population)

self**.**pop\_created **=** 1 # indicates that the initial population is created in order to enable drawing solutions on GUI.

self**.**num\_attacks\_Label**.**text **=** "Initialized"

self**.**ga\_instance **=** pygad**.**GA**(**num\_generations**=**n\_gen**,**

num\_parents\_mating**=**50**,**

fitness\_func**=**fitness**,**

num\_genes**=**n\_queen**,**

initial\_population**=**self**.**population\_1D\_vector**,**

mutation\_percent\_genes**=**0.01**,**

mutation\_type**=**"random"**,**

mutation\_num\_genes**=**1**,**

mutation\_by\_replacement**=True,**

random\_mutation\_min\_val**=**0.0**,**

random\_mutation\_max\_val**=**n\_queen\_r**,**

on\_generation**=**on\_gen\_callback**,**

delay\_after\_gen**=**0.2**)**

**def** vector\_to\_matrix**(**self**):**

# Converts the 1D vector solutions into a 2D matrix solutions represrnting the board, where 1 means a queen exists. The matrix form of the solutions makes calculating the fitness value much easier.

self**.**population **=** numpy**.**zeros**(**shape**=(**self**.**num\_solutions**,** n\_queen**,** n\_queen**))**

solution\_idx **=** 0

**for** current\_solution **in** self**.**population\_1D\_vector**:**

# print(self.population\_1D\_vector[solution\_idx, :])

current\_solution **=** numpy**.**uint8**(**current\_solution**)**

row\_idx **=** 0

**for** col\_idx **in** current\_solution**:**

self**.**population**[**solution\_idx**,** row\_idx**,** col\_idx**]** **=** 1

row\_idx **=** row\_idx **+** 1

# print(self.population[solution\_idx, :])

solution\_idx **=** solution\_idx **+** 1

**def** reset\_board\_text**(**self**):**

# Reset board on GUI.

**for** row\_idx **in** **range(**self**.**all\_widgets**.**shape**[**0**]):**

**for** col\_idx **in** **range(**self**.**all\_widgets**.**shape**[**1**]):**

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**text**=**"[color=ffffff]"**+str(**row\_idx**)+**", "**+str(**col\_idx**)+**"[/color]"

**with** self**.**all\_widgets**[**row\_idx**,** col\_idx**].**canvas**.**before**:**

kivy**.**graphics**.**Color**(**0**,** 0**,** 0**,** 1**)** # green; colors range from 0-1 not 0-255

self**.**rect **=** kivy**.**graphics**.**Rectangle**(**size**=**self**.**all\_widgets**[**row\_idx**,** col\_idx**].**size**,** pos**=**self**.**all\_widgets**[**row\_idx**,** col\_idx**].**pos**)**

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**font\_size **=** 20

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**canvas**.**ask\_update**()**

self**.**gridLayout**.**do\_layout**()**

**def** update\_board\_UI**(**self**,** **\***args**):**

**if** **(not** **(**"pop\_created" **in** **vars(**self**))** **or** **(**self**.**pop\_created **==** 0**)):**

**print(**"No Population Created Yet. Create the initial Population by Pressing the \"Initial Population\" Button in Order to Call the initialize\_population() Method At First."**)**

# self.num\_attacks\_Label.text = "Press \"Initial Population\""

**return**

\_**,** max\_fitness**,** best\_sol\_idx **=** self**.**ga\_instance**.**best\_solution**()**

best\_sol **=** self**.**population**[**best\_sol\_idx**,** **:].**copy**()**

# self.num\_attacks\_Label.text = "Max Fitness = " + str(numpy.round(max\_fitness, 4))

self**.**num\_attacks\_Label**.**text **=** **str(**self**.**ga\_instance**.**generations\_completed**)** **+** " " **+** **str(**numpy**.round(**max\_fitness**,** 4**))**

**if** **abs(**BuzzleApp**.**old\_best\_sol\_fitness **-** max\_fitness**)** **<** 0.001**:**

**return**

self**.**reset\_board\_text**()**

**for** row\_idx **in** **range(**n\_queen**):**

**for** col\_idx **in** **range(**n\_queen**):**

**if** **(**best\_sol**[**row\_idx**,** col\_idx**]** **==** 1**):**

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**text **=** "[color=22ff22]Queen[/color]"

**with** self**.**all\_widgets**[**row\_idx**,** col\_idx**].**canvas**.**before**:**

kivy**.**graphics**.**Color**(**0**,** 1**,** 0**,** 1**)** # green; colors range from 0-1 not 0-255

self**.**rect **=** kivy**.**graphics**.**Rectangle**(**size**=**self**.**all\_widgets**[**row\_idx**,** col\_idx**].**size**,** pos**=**self**.**all\_widgets**[**row\_idx**,** col\_idx**].**pos**)**

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**font\_size **=** 30

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**canvas**.**ask\_update**()**

self**.**gridLayout**.**do\_layout**()**

BuzzleApp**.**old\_best\_sol\_fitness **=** max\_fitness

BuzzleApp**.**old\_best\_sol\_idx **=** best\_sol\_idx

**def** build**(**self**):**

self**.**boxLayout **=** kivy**.**uix**.**boxlayout**.**BoxLayout**(**orientation**=**"vertical"**)**

self**.**gridLayout **=** kivy**.**uix**.**gridlayout**.**GridLayout**(**rows**=**n\_queen**,** size\_hint\_y**=**n\_queen1**)**

self**.**boxLayout\_buttons **=** kivy**.**uix**.**boxlayout**.**BoxLayout**(**orientation**=**"horizontal"**)**

self**.**boxLayout**.**add\_widget**(**self**.**gridLayout**)**

self**.**boxLayout**.**add\_widget**(**self**.**boxLayout\_buttons**)**

# Preparing the 8x8 board.

self**.**all\_widgets **=** numpy**.**zeros**(**shape**=(**n\_queen**,**n\_queen**),** dtype**=**"O"**)**

**for** row\_idx **in** **range(**self**.**all\_widgets**.**shape**[**0**]):**

**for** col\_idx **in** **range(**self**.**all\_widgets**.**shape**[**1**]):**

self**.**all\_widgets**[**row\_idx**,** col\_idx**]** **=** kivy**.**uix**.**button**.**Button**(**text**=str(**row\_idx**)+**", "**+str(**col\_idx**),** font\_size**=**20**)**

self**.**all\_widgets**[**row\_idx**,** col\_idx**].**markup **=** **True**

self**.**gridLayout**.**add\_widget**(**self**.**all\_widgets**[**row\_idx**,** col\_idx**])**

# Preparing buttons inside the child BoxLayout.

initial\_button **=** kivy**.**uix**.**button**.**Button**(**text**=**"Initial Population"**,** font\_size**=**15**,** size\_hint\_x**=**2**)**

initial\_button**.**bind**(**on\_press**=**self**.**initialize\_population**)**

ga\_solution\_button **=** kivy**.**uix**.**button**.**Button**(**text**=**"Show Best Solution"**,** font\_size**=**15**,** size\_hint\_x**=**2**)**

ga\_solution\_button**.**bind**(**on\_press**=**self**.**update\_board\_UI**)**

start\_ga\_button **=** kivy**.**uix**.**button**.**Button**(**text**=**"Start GA"**,** font\_size**=**15**,** size\_hint\_x**=**2**)**

start\_ga\_button**.**bind**(**on\_press**=**self**.**start\_ga**)**

self**.**num\_attacks\_Label **=** kivy**.**uix**.**label**.**Label**(**text**=**"Max Fitness"**,** font\_size**=**30**,** size\_hint\_x**=**2**)**

self**.**boxLayout\_buttons**.**add\_widget**(**initial\_button**)**

self**.**boxLayout\_buttons**.**add\_widget**(**ga\_solution\_button**)**

self**.**boxLayout\_buttons**.**add\_widget**(**start\_ga\_button**)**

self**.**boxLayout\_buttons**.**add\_widget**(**self**.**num\_attacks\_Label**)**

**return** self**.**boxLayout

**def** fitness**(**solution\_vector**,** solution\_idx**):**

**if** solution\_vector**.**ndim **==** 2**:**

solution **=** solution\_vector

**else:**

solution **=** numpy**.**zeros**(**shape**=(**n\_queen**,** n\_queen**))**

row\_idx **=** 0

**for** col\_idx **in** solution\_vector**:**

solution**[**row\_idx**,** **int(**col\_idx**)]** **=** 1

row\_idx **=** row\_idx **+** 1

total\_num\_attacks\_column **=** attacks\_column**(**solution**)**

total\_num\_attacks\_diagonal **=** attacks\_diagonal**(**solution**)**

total\_num\_attacks **=** total\_num\_attacks\_column **+** total\_num\_attacks\_diagonal

**if** total\_num\_attacks **==** 0**:**

total\_num\_attacks **=** 1.1 # float("inf")

**else:**

total\_num\_attacks **=** 1.0**/**total\_num\_attacks

**return** total\_num\_attacks

**def** attacks\_diagonal**(**ga\_solution**):**

total\_num\_attacks **=** 0 # Number of attacks for the solution (diagonal only).

# Badding zeros around the solution board for being able to index the boundaries (leftmost/rightmost coumns & top/bottom rows). # This is by adding 2 rows (1 at the top and another at the bottom) and adding 2 columns (1 left and another right).

temp **=** numpy**.**zeros**(**shape**=(**n\_queen2**,** n\_queen2**))**

# Copying the solution board inside the badded array.

temp**[**1**:**n\_queen1**,** 1**:**n\_queen1**]** **=** ga\_solution

# print("Solution Board after Badding : ", temp)

# Returning the indices (rows and columns) of the 8 queeens.

row\_indices**,** col\_indices **=** numpy**.**where**(**ga\_solution **==** 1**)**

# Adding 1 to the indices because the badded array is 1 row/column far from the original array.

row\_indices **=** row\_indices **+** 1

col\_indices **=** col\_indices **+** 1

# print("Column indices of the queens : ", col\_indices

total **=** 0 # total number of attacking pairs diagonally for each solution.

**for** element\_idx **in** **range(**n\_queen**):**

x **=** row\_indices**[**element\_idx**]**

y **=** col\_indices**[**element\_idx**]**

# print("ROW index of the current queen : ", x)

# print("COL index of the current queen : ", y)

mat\_bottom\_right **=** temp**[**x**:,** y**:]**

total **=** total **+** diagonal\_attacks**(**mat\_bottom\_right**)**

# print("Bottom Right : ", total)

mat\_bottom\_left **=** temp**[**x**:,** y**:**0**:-**1**]**

total **=** total **+** diagonal\_attacks**(**mat\_bottom\_left**)**

# print("Bottom Left : ", total)

mat\_top\_right **=** temp**[**x**:**0**:-**1**,** y**:]**

total **=** total **+** diagonal\_attacks**(**mat\_top\_right**)**

# print("Top Right : ", total)

mat\_top\_left **=** temp**[**x**:**0**:-**1**,** y**:**0**:-**1**]**

total **=** total **+** diagonal\_attacks**(**mat\_top\_left**)**

# print("Top Left : ", total)

total\_num\_attacks **=** total\_num\_attacks **+** total **/**2

**return** total\_num\_attacks

**def** diagonal\_attacks**(**mat**):**

**if** **(**mat**.**shape**[**0**]** **<** 2 **or** mat**.**shape**[**1**]** **<** 2**):**

# print("LESS than 2x2.")

**return** 0

num\_attacks **=** mat**.**diagonal**().sum()-**1

**return** num\_attacks

**def** attacks\_column**(**ga\_solution**):**

# For a given queen, how many queens sharing the same coulmn? This is how the fitness value is calculated.

total\_num\_attacks **=** 0 # Number of attacks for the solution (column only).

**for** queen\_y\_pos **in** **range(**n\_queen**):**

# Vertical

col\_sum **=** numpy**.sum(**ga\_solution**[:,** queen\_y\_pos**])**

**if** **(**col\_sum **==** 0 **or** col\_sum **==** 1**):**

col\_sum **=** 0

**else:**

col\_sum **=** col\_sum **-** 1 # To avoid regarding a queen attacking itself.

total\_num\_attacks **=** total\_num\_attacks **+** col\_sum

**return** total\_num\_attacks

**def** on\_gen\_callback**(**ga\_instance**):**

**global** buzzleApp

**print(**"Generation = {gen}"**.format(**gen**=**ga\_instance**.**generations\_completed**))**

**print(**"Fitness = {fitness}"**.format(**fitness**=**ga\_instance**.**best\_solution**()[**1**]))**

buzzleApp**.**population\_1D\_vector **=** ga\_instance**.**population

buzzleApp**.**vector\_to\_matrix**()**

buzzleApp**.**update\_board\_UI**()**

# buzzleApp.gridLayout.export\_to\_png("gen\_"+str(ga\_instance.generations\_completed)+".png")

**from** kivy**.**config **import** Config

Config**.set(**'graphics'**,** 'width'**,** '1000'**)**

Config**.set(**'graphics'**,** 'height'**,** '600'**)**

buzzleApp **=** BuzzleApp**()**

buzzleApp**.**title **=** "Plays "**+str(**n\_queen**)+**" Queen Puzzle"

buzzleApp**.**run**()**

AIM: Write a PROLOG program based on list.

A) To find the length of a list.

list\_length([],0).

list\_length([\_|TAIL],N) :- list\_length(TAIL,N1), N is N1 + 1.

|  |
| --- |
| ?- [list\_op]  | .  true.  ?- list\_length([1,23,43,12,67,44,89,32],Len).  Len = 8.  ?- list\_length([],Len).  Len = 0.  ?- list\_length([[a,b],[c,d],[e,f]],Len).  Len = 3. |

B) To sum all numbers of list.

list\_sum([],0).

list\_sum([Head|Tail], Sum) :- list\_sum(Tail,SumTemp), Sum is Head + SumTemp.

|  |
| --- |
| ?- list\_sum([5,12,69,112,48,4],Sum).  Sum = 250.  ?- list\_sum([1,2,3,4,5,6,7,8,9],Sum).  Sum = 45. |

C) To find whether given element is a member of a list.

list\_member(X,[X|\_]).

list\_member(X,[\_|TAIL]) :- list\_member(X,TAIL).

|  |
| --- |
| ?- list\_member(g,[a,b,u]).  false.  ?- list\_member(a,[a,b,u]).  true .  ?- list\_member([b,c],[a,[b,c]]).  true . |

D) To append the list.

list\_append(A,T,T) :- list\_member(A,T),!.

list\_append(A,T,[A|T]).

|  |
| --- |
| ?- list\_append(a,[e,i,o,u],NewList).  NewList = [a, e, i, o, u].  ?- list\_append(e,[e,i,o,u],NewList).  NewList = [e, i, o, u].  ?- list\_append([a,b],[e,i,o,u],NewList).  NewList = [[a, b], e, i, o, u]. |

E) To reverse the list.

list\_concat([],L,L).

list\_concat([X1|L1],L2,[X1|L3]) :- list\_concat(L1,L2,L3).

list\_rev([],[]).

list\_rev([Head|Tail],Reversed) :-

list\_rev(Tail, RevTail),list\_concat(RevTail, [Head],Reversed).

|  |
| --- |
| ?- list\_rev([a,b,c,d,e],NewList).  NewList = [e, d, c, b, a].  ?- list\_rev([a,b,c,d,e],[e,d,c,b,a]). |

F) To find the last element of a list.

list\_last([],[]).

list\_last([X],X).

list\_last([\_|TAIL],NewList) :- list\_last(TAIL,NewList).

|  |
| --- |
| ?- list\_last([a,b,c,d,e],NewList).  NewList = e .  ?- list\_last([],NewList).  NewList = []. |

G) To delete the first occurrence of an element from a list.

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

|  |
| --- |
| ?- list\_delete(a,[a,b,c,d,e],NewList).  NewList = [b, c, d, e] .  ?- list\_delete([a,b],[a,b,c,d,e],NewList).  false.  ?- list\_delete([a,b],[[a,b],c,d,e],NewList).  NewList = [c, d, e] . |

H) To delete every occurrence of an element from a list.

list\_delete\_r(X, [], []) :- !.

list\_delete\_r(X, [X|Xs], Y) :- !, list\_delete\_r(X, Xs, Y).

list\_delete\_r(X, [T|Xs], Y) :- !, list\_delete\_r(X, Xs, Y2), append([T], Y2, Y).

|  |
| --- |
| ?- list\_delete\_r(a,[b,a,a,c,d,a,e],NewList).  NewList = [b, c, d, e].  ?- list\_delete\_r(a,[],NewList).  NewList = []. |

I) to find Nth element from the list.

list\_nth([H|\_],0,H) :- !.

list\_nth([\_|T],N,H) :- N > 0, N1 is N-1, list\_nth(T,N1,H).

|  |
| --- |
| ?- list\_nth([b,a,a,c,d,a,e],3,NewList).  NewList = c.  ?- list\_nth([b,a,a,c,d,a,e],2,NewList).  NewList = a. |