Assignment 1 CS205: Artificial Intelligence. Dr. Eamonn Keogh

Rahul Sailesh Wadhwa
SID 862309846
Email rwadh004@ucr.edu
Date: May 13,2022

In completing this assignment I consulted:

- https://docs.python.org/3.6/contents.html
- The Blind Search and Heuristic Search lecture slides provided by Dr. Eamonn Keogh
- The assignment materials and sample report provided by Dr. Eamonn Keogh

All my code is original and implemented from scratch by me. I also made use of following built in subroutines-

- All subroutines from **numpy**, for handling the different states and structuring and restructuring them as required.

Outline of this report:

CS205: Assignment 1: The 8-puzzle
Rahul Sailesh Wadhwa, SID 862309846 Date: May 13,2022

Introduction

The 8 puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It consists of a 3-by-3 grid containing 8 blocks labeled 1 through 8 and a blank block. The goal of the puzzle is to rearrange the blocks so that the numbers are in order by sliding the different numbered blocks to fill the blank space (i.e. the blocks cannot be swapped at random and the only legal moves allowed are to slide the blocks to fill the blank space). An example of the puzzle can be seen in the image shown.

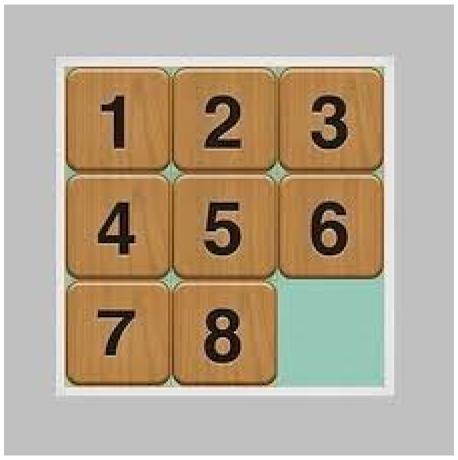


Figure 1: A picture of the 8 puzzle

This assignment is the first project assigned by Dr. Eamonn Keogh as a part of the Artificial Intelligence course at the University of California during the quarter of Spring 2022. The following write up is to detail my findings as I completed this project. It explored 3 different searching techniques- Uniform Cost Search, A* search with the Misplaced Tile heuristic and A* search with the Manhattan distance heuristic. My language of choice was Python (version 3) and the full code for the project is included as well.

Comparison of the different algorithms

The 3 algorithms that were required to be implemented as a part of this project are as follows-

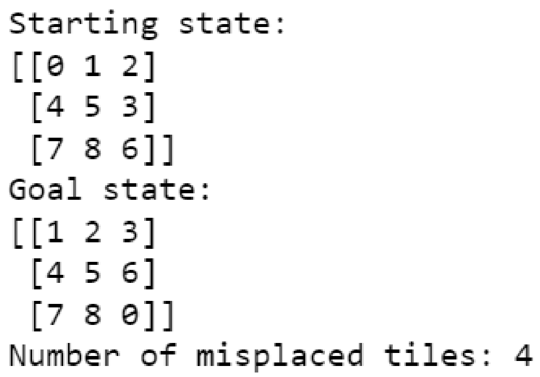
- Uniform Cost Search
- A* Search with the Misplaced Tile heuristic
- A* search with the Manhattan Distance heuristic

Uniform Cost Search

Uniform Cost search is a variation of A* search where the heuristic is hard coded to 0. For this problem, it is equivalent to Breadth First search. This means that for a given depth, every state encountered has to be expanded before moving to the next level. The cost to expand any given node is considered to be 1 and there are no weights assigned to each of the expanded states. This is the case as the effort required to slide the tile in any direction will be the same.

The Misplaced Tile Heuristic

The second algorithm implemented as a part of this project is A* search with the Misplaced Tile Heuristic. This heuristic will keep track of the number of “misplaced” tiles in the puzzle. A particular tile is said to be “misplaced” if its position is different from the corresponding numbered tile in the goal state. An example of this heuristic can be seen below-



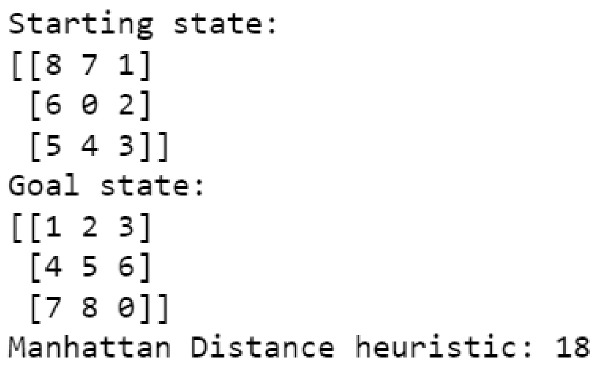
```
Starting state:
[[0 1 2]
 [4 5 3]
 [7 8 6]]
Goal state:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Number of misplaced tiles: 4
```

Figure 2: Example of the Misplaced Tile heuristic

The blank space is not counted while taking the heuristic into consideration, and only the numbered tiles are counted. The algorithm will choose a path to the goal state such that the number of misplaced tiles either stays the same or decreases (ideally). In this way the puzzle is solved by continually expanding the nodes till the number of misplaced tiles is 0 (i.e. the goal state is reached).

The Manhattan Distance Heuristic

This heuristic is similar to the Misplaced Tile Heuristic in that it considers each of the misplaced tiles and the number of moves that it would take for them to reach their goal states. The result of this heuristic is the sum of cost of the moves required to make each of the misplaced tiles reach their goal states.



```
Starting state:
[[8 7 1]
 [6 0 2]
 [5 4 3]]
Goal state:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Manhattan Distance heuristic: 18
```

Figure 3: An example of the Manhattan Distance heuristic

Comparison of Algorithms on Sample Puzzles

As provided by Dr. Keogh, the different algorithms were tested with each of the sample puzzles shown below.

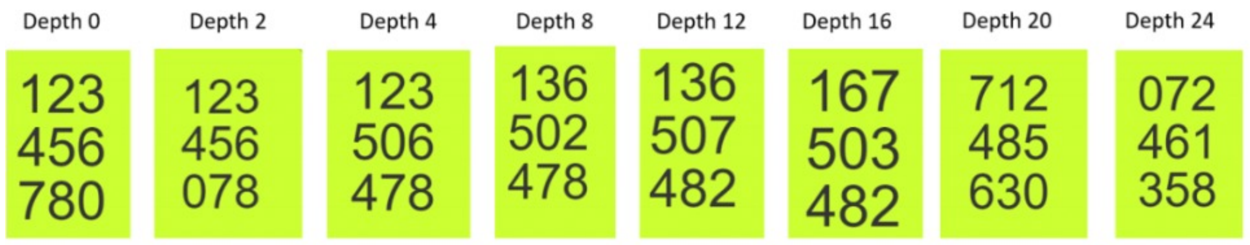


Figure 4: Sample problems for the 8 puzzle problem

It was observed that in case of the puzzles in which the optimal solution was found at a relatively low depth, there was not much of a difference in finding the solution between the 3 algorithms. However, in the case of algorithms that had a greater depth, there was a significant difference observed between the performance of the 3 algorithms. The number of nodes expanded also varied greatly, with the most number of nodes being expanded in the case of Uniform Cost search, followed by A* search with the Misplaced Tile Heuristic and finally the A* search with the Manhattan Distance heuristic. Below we can observe a graph of performance for some of the pre-defined problems used in the algorithm.

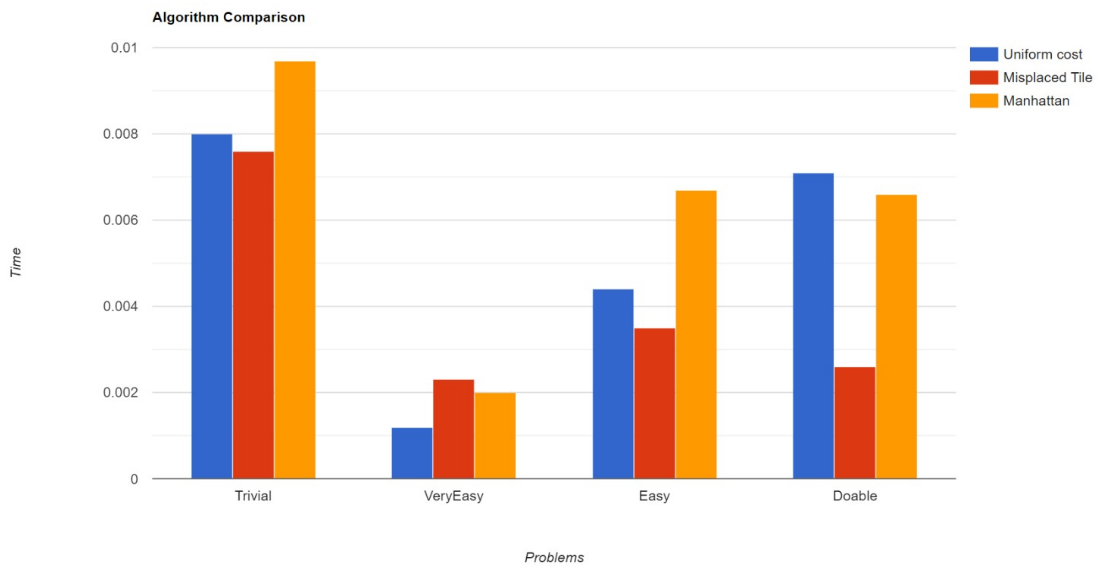


Figure 5: Graph showing the comparison of the 3 algorithms for the different pre-defined problems

Conclusion

Considering the list of 3 algorithms and the comparisons between them:
Uniform Cost Search, A* Search with Misplaced Tile Heuristic, A* Search with Manhattan Distance Heuristic

- It can be seen that out of the 3 algorithms, the A* search algorithm along with the Manhattan distance heuristic performed the best overall in terms of time to reach the goal state as well as the number of nodes expanded before reaching the goal state.
- The Misplaced Tile and Manhattan Distance heuristics improved the efficiency of search as compared to the Uniform Cost Search algorithm (whose heuristic is hard coded to 0) and has a time complexity of $O(b^d)$.

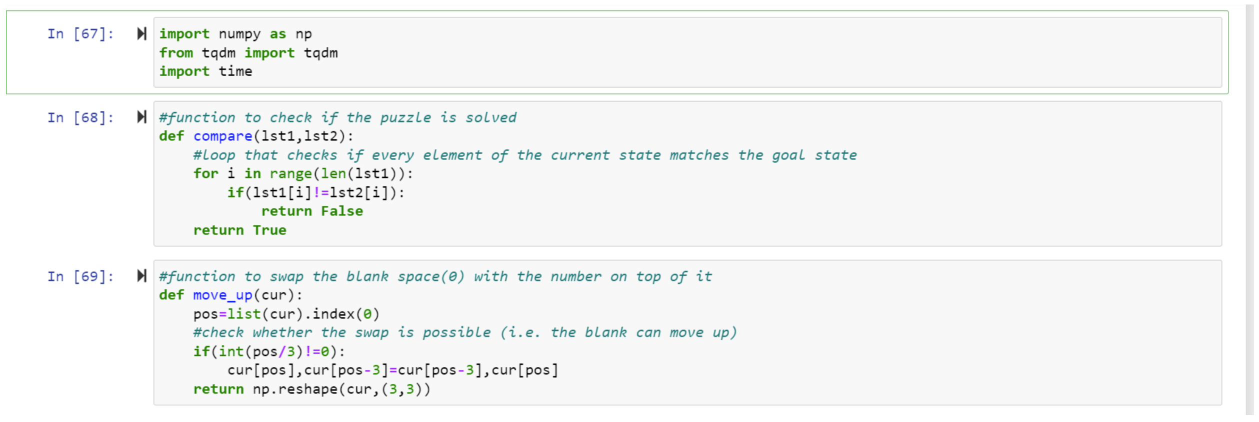
The following is a Sample Walkthrough of an **easy** puzzle (Depth 2)

```
Enter the difficulty of puzzle:
 1. Trivial
 2. Very Easy
 3. Easy
 4. Doable
 5. Hard
 6. Custom
Enter your choice: 3
Start state:
[[1 2 0]
 [4 5 3]
 [7 8 6]]
Goal state:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Enter the choice of algorithm you wish to run-
 1. Uniform Cost search
 2. A* search(Misplaced Tile heuristic)
 3. A* search(Manhattan Distance heuristic)
Enter your choice: 3

A* search with Manhattan Distance heuristic
[[1 2 0]
 [4 5 3]
 [7 8 6]]
h(n) = 2
Depth: 0

[[1 2 3]
 [4 5 0]
 [7 8 6]]
h(n) = 1
Depth: 1

[[1 2 3]
 [4 5 6]
 [7 8 0]]
h(n) = 0
Depth: 2

Success
Nodes expanded:  3
Depth:  2
```

The following is a Sample Walkthrough of a **hard** puzzle (Depth 16)

```
Enter the difficulty of puzzle:
 1. Trivial
 2. Very Easy
 3. Easy
 4. Doable
 5. Hard
 6. Custom
Enter your choice: 6
Enter the puzzle using 0 to indicate the blank. Please enter only valid puzzles with the numbers
separated by spaces.
Enter first row: 1 6 7
Enter second row: 5 0 3
Enter third row: 4 8 2
Start state:
[[1 6 7]
 [5 0 3]
 [4 8 2]]
Goal state:
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Enter the choice of algorithm you wish to run-
 1. Uniform Cost search
 2. A* search(Misplaced Tile heuristic)
 3. A* search(Manhattan Distance heuristic)
Enter your choice: 3

A* search with Manhattan Distance heuristic
[[1 6 7]
 [5 0 3]
 [4 8 2]]
h(n) = 12
Depth: 0

……. Here I deleted several pages of the trace to save space

[[1 2 3]
 [4 5 6]
 [7 0 8]]
1
Depth: 15

[[1 2 3]
 [4 5 6]
 [7 8 0]]
0
Depth: 16

Success
Nodes expanded:  177
Depth:  16
```

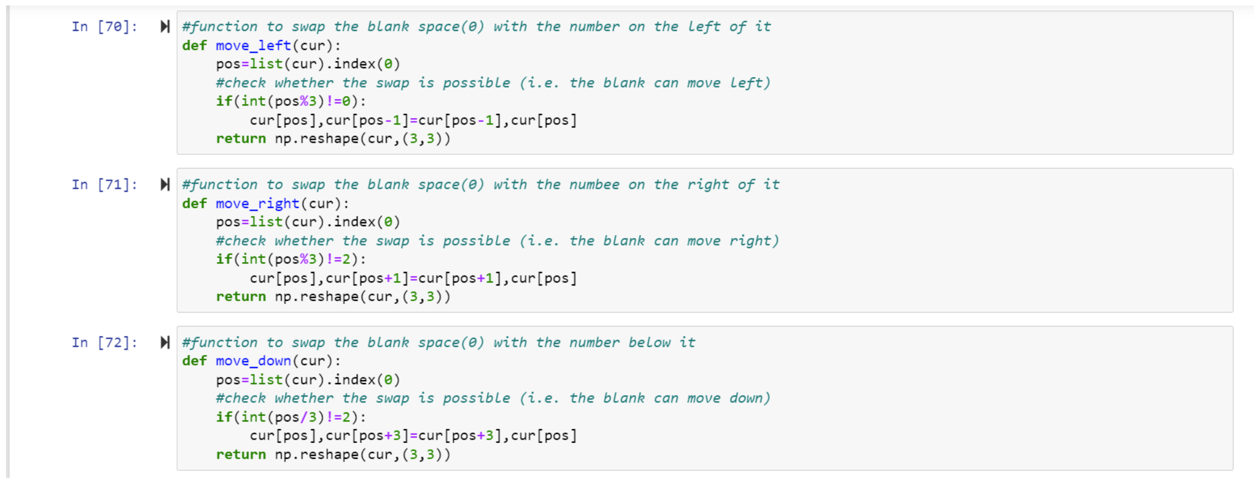
URL to me code is:
https://drive.google.com/file/d/1J0TxjEntNXztdde6hQ09Y5QYBrEe9bc_/view?usp=sharing



```
In [67]:  import numpy as np
          from tqdm import tqdm
          import time
```

```
In [68]:  #function to check if the puzzle is solved
          def compare(lst1,lst2):
              #Loop that checks if every element of the current state matches the goal state
              for i in range(len(lst1)):
                  if(lst1[i]!=lst2[i]):
                      return False
              return True
```

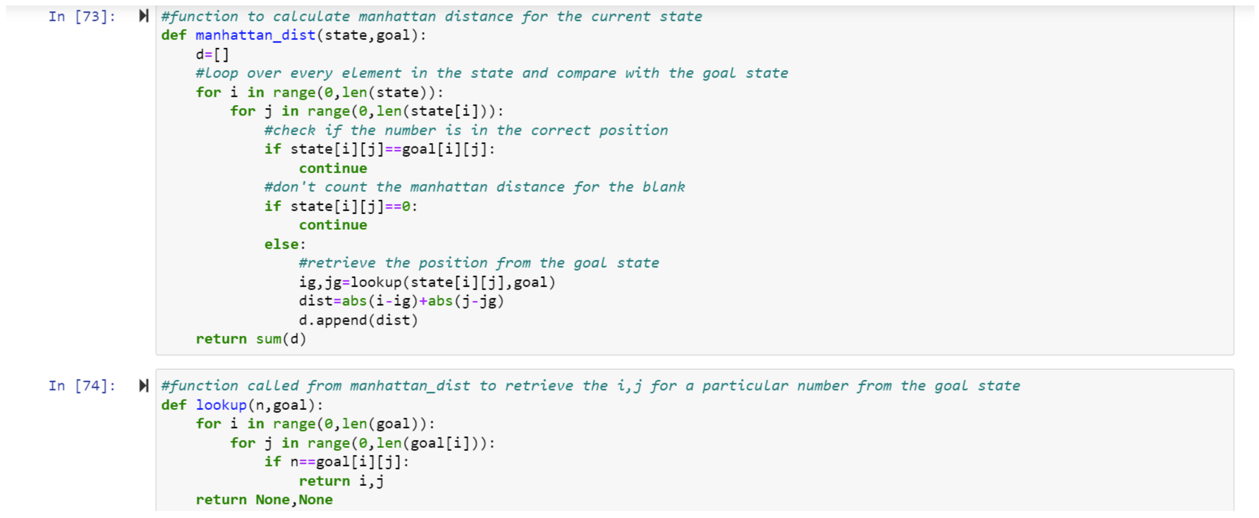
```
In [69]:  #function to swap the blank space(0) with the number on top of it
          def move_up(cur):
              pos=list(cur).index(0)
              #check whether the swap is possible (i.e. the blank can move up)
              if(int(pos/3)!=0):
                  cur[pos],cur[pos-3]=cur[pos-3],cur[pos]
              return np.reshape(cur,(3,3))
```



```
In [70]:  #function to swap the blank space(0) with the number on the left of it
          def move_left(cur):
              pos=list(cur).index(0)
              #check whether the swap is possible (i.e. the blank can move left)
              if(int(pos%3)!=0):
                  cur[pos],cur[pos-1]=cur[pos-1],cur[pos]
              return np.reshape(cur,(3,3))
```

```
In [71]:  #function to swap the blank space(0) with the numbee on the right of it
          def move_right(cur):
              pos=list(cur).index(0)
              #check whether the swap is possible (i.e. the blank can move right)
              if(int(pos%3)!=2):
                  cur[pos],cur[pos+1]=cur[pos+1],cur[pos]
              return np.reshape(cur,(3,3))
```

```
In [72]:  #function to swap the blank space(0) with the number below it
          def move_down(cur):
              pos=list(cur).index(0)
              #check whether the swap is possible (i.e. the blank can move down)
              if(int(pos/3)!=2):
                  cur[pos],cur[pos+3]=cur[pos+3],cur[pos]
              return np.reshape(cur,(3,3))
```



```
In [73]:  #function to calculate manhattan distance for the current state
          def manhattan_dist(state,goal):
              d=[]
              #loop over every element in the state and compare with the goal state
              for i in range(0,len(state)):
                  for j in range(0,len(state[i])):
                      #check if the number is in the correct position
                      if state[i][j]==goal[i][j]:
                          continue
                      #don't count the manhattan distance for the blank
                      if state[i][j]==0:
                          continue
                      else:
                          #retrieve the position from the goal state
                          ig,jg=lookup(state[i][j],goal)
                          dist=abs(i-ig)+abs(j-jg)
                          d.append(dist)
              return sum(d)
```

```
In [74]:  #function called from manhattan_dist to retrieve the i,j for a particular number from the goal state
          def lookup(n,goal):
              for i in range(0,len(goal)):
                  for j in range(0,len(goal[i])):
                      if n==goal[i][j]:
                          return i,j
              return None,None
```

In [75]:

```
#function to count the number of misplaced tiles
def misplaced_tile(state,goal):
    ctr=0
    for i in range(0,len(state)):
        for j in range(0,len(state[i])):
            if state[i][j]==0:
                continue
            if state[i][j]!=goal[i][j]:
                ctr+=1
    return ctr
```

In [102]:

```
#uniform cost search(no heuristic involved)
def uniform_cost(start,goal):
    depth=[]
    nodes=[]
    sucmsg="Success"
    failmsg="Failure"
    nodes_expanded=0
    visited_nodes=[]
    starttime=time.time()
    #check if the start state is already in the goal state
    if(compare(start.flatten(),goal.flatten())):
        print(start)
        print("Solution found at depth 0")
        print("Nodes expanded: ",nodes_expanded)
        end=time.time()-starttime
        print("Solution found in: {:.4f}".format(end)," seconds")
        return sucmsg
    #if start state is not the goal state add it to the list of states to be expanded
    nodes.append(start)
    #keeping track of the current depth using a list
    depth.append(0)
    #infinite loop to keep the search going until the solution is found
    while True:
        #terminal condition for the loop when there are no states left to traverse (i.e. the solution does not exist)
        if(len(nodes)==0):
            return failmsg
        #popping the states in order for breadth first search
```

In [109]:

```
#A* search using the misplaced tile heuristic
def a_star_mtile(state, goal):
    depth=[]
    d=0
    nodes=[]
    sucmsg="Success"
    failmsg="Failure"
    nodes_expanded=0
    visited_nodes=[]
    starttime=time.time()
    #check if the start state is already in the goal state
    if(compare(state.flatten(),goal.flatten())):
        print(state)
        print("Solution found at depth 0")
        print("Nodes expanded: ",nodes_expanded)
        end=time.time()-starttime
        print("Solution found in: {:.4f}".format(end)," seconds")
        return sucmsg
    #if start state is not the goal state add it to the list of states to be expanded
    nodes.append(state)
    #keeping track of the current depth using a list
    depth.append(d)
    #infinite loop to keep the search going until the solution is found
    while True:
        #terminal condition for the loop when there are no states left to traverse (i.e. the solution does not exist)
        if(len(nodes)==0):
```

In [110]:

```python
#A* search using the manhattan distance heuristic
def a_star_man(state, goal):
    depth=[]
    d=0
    nodes=[]
    sucmsg="Success"
    failmsg="Failure"
    nodes_expanded=0
    visited_nodes=[]
    starttime=time.time()
    #check if the start state is already in the goal state
    if(compare(state.flatten(),goal.flatten())):
        print(state)
        print("Solution found at depth 0")
        print("Nodes expanded: ",nodes_expanded)
        end=time.time()-starttime
        print("Solution found in: {:.4f}".format(end)," seconds")
        return sucmsg
    #if start state is not the goal state add it to the list of states to be expanded
    nodes.append(state)
    #keeping track of the current depth using a list
    depth.append(d)
    #infinite loop to keep the search going until the solution is found
    while True:
        #terminal condition for the loop when there are no states left to traverse (i.e. the solution does not exist)
        if(len(nodes)==0):
            return failmsg
```

In [99]:

```python
#driver function
def main():
    #few simple testing problems
    trivial=np.reshape(np.array([1,2,3,4,5,6,7,8,0]),(3,3))
    veasy=np.reshape(np.array([1,2,3,4,5,6,7,0,8]),(3,3))
    easy=np.reshape(np.array([1,2,0,4,5,3,7,8,6]),(3,3))
    doable=np.reshape(np.array([0,1,2,4,5,3,7,8,6]),(3,3))
    ohboy=np.reshape(np.array([8,7,1,6,0,2,5,4,3]),(3,3))
    print("Enter the difficulty of puzzle:","\n","1. Trivial","\n","2. Very Easy","\n","3. Easy","\n","4. Doable","\n","5. Ha
    #input choice of puzzle
    ch=int(input("Enter you choice: "))
    if ch==1:
        start=trivial
    elif ch==2:
        start=veasy
    elif ch==3:
        start=easy
    elif ch==4:
        start=doable
    elif ch==5:
        start=ohboy
    else:
        #reading inputs from the user for a custom puzzle
        print("Enter the puzzle using 0 to indicate the blank. Please enter only valid puzzles with the numbers separated by
        lst=input("Enter first row: ")
        lst1=input("Enter second row: ")
        lst2=input("Enter third row: ")
```