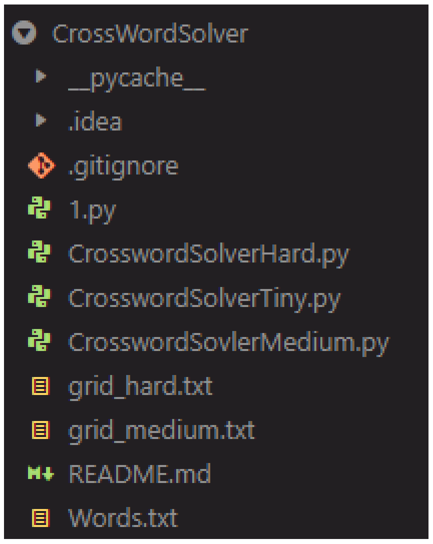
# CS480 Assignment -3

## Analysis: -

The below code is available at git repo => https://github.com/ashishodu2023/CrosswordsolverAi.git with readme files with all the instructions on how to execute the cross wore puzzle.  The medium and hard crossword puzzle uses Forward Checking as of the Heuristics to reduce the domain and search space. The execution time for each of the puzzle is given after code completion with results of each of the execution.

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and the overall amount of work done.

## Folder Structure: -



```
CrossWordSolver
  __pycache__
  .idea
  .gitignore
  1.py
  CrosswordSolverHard.py
  CrosswordSolverTiny.py
  CrosswordSovlerMedium.py
  grid_hard.txt
  grid_medium.txt
  README.md
  Words.txt
```

## Grid Medium Structure: -

```
##-------###
##-#####-##-
##-#####-##-
##-##-----#-
##-####-###-
####-##-----
####-##-#-#-
#-#-----#-##
#-##-##-#-##
------###-##
#-##-####-##
```

## Grid Hard Structure: -

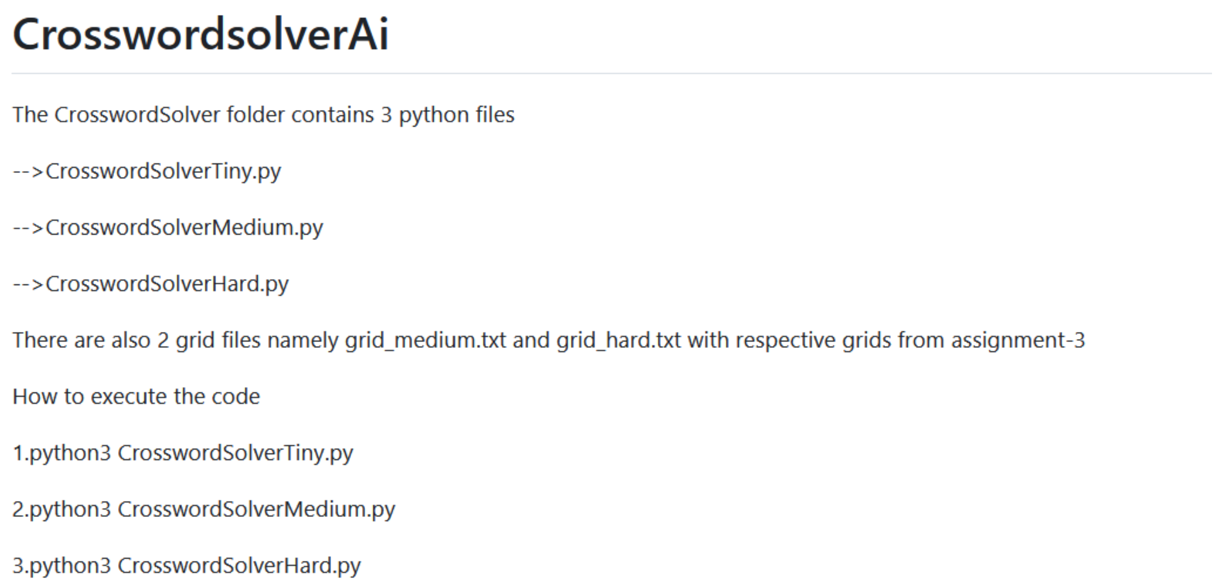
```
##---###---##
#-----#-----#
#-----#-----#
---#-----#---
----#---#----
----#---#----
#----###----#
##----#----##
###-------###
```

####-----####

#####ash#####

######-######

## ReadMe :-



### CrosswordsolverAi

The CrosswordSolver folder contains 3 python files

-->CrosswordSolverTiny.py

-->CrosswordSolverMedium.py

-->CrosswordSolverHard.py

There are also 2 grid files namely grid_medium.txt and grid_hard.txt with respective grids from assignment-3

How to execute the code

1.python3 CrosswordSolverTiny.py

2.python3 CrosswordSolverMedium.py

3.python3 CrosswordSolverHard.py

## Tiny Puzzle

```python
import time

def CrosswordSolverTiny(grid, word_data):

    # Function to check if a word can be placed in the given direction at the specified position
    def CheckHorizontalVertical(word, row, col, direction):
        if direction == "ACROSS":
            if col + len(word) > len(grid[0]):
                return False
            for i in range(len(word)):
                if grid[row][col + i] != " " and grid[row][col
+ i] != word[i]:
                    return False
                if grid[row][col + i] == "#":
                    return False
        else:  # direction == "down"
            if row + len(word) > len(grid):
                return False
            for i in range(len(word)):
                if grid[row + i][col] != " " and grid[row +
i][col] != word[i]:
                    return False
                if grid[row + i][col] == "#":
                    return False
        return True

    # Function to set a word in the grid at the specified
position and direction
    def SetWord(word, row, col, direction):
        if direction == "ACROSS":
            for i in range(len(word)):
                grid[row][col + i] = word[i]
        else:  # direction == "down"
            for j in range(len(word)):
                grid[row + j][col] = word[j]

    # Function to clear a word from the grid at the specified
position and direction
    def GetWord(word, row, col, direction):
        if direction == "ACROSS":
            for i in range(len(word)):
                grid[row][col + i] = " "
        else:  # direction == "down"
            for j in range(len(word)):
                grid[row + j][col] = " "
```

```python
    # Recursive function to solve the crossword puzzle
    def Solve(grid, word_data):
        if not word_data:
            return True

        variable, start_cell, domain = word_data[0]
        for row, col in start_cell:
            for word in list(domain):
                if CheckHorizontalVertical(word, row, col,
variable[1:]):
                    SetWord(word, row, col, variable[1:])
                    domain.remove(word)
                    if Solve(grid, word_data[1:]):
                        return True
                    GetWord(word, row, col, variable[1:])
                    domain.add(word)

        return False

    # Main function to solve the crossword puzzle and print the
result
    start_time = time.time()
    if Solve(grid, word_data):
        for row in grid:
            print("".join(row))
        end_time = (time.time() - start_time)
        print(f'\n---Time taken for code execution %s seconds -
--{end_time}')
    else:
        print("No solution found")


def main():
    # Example crossword grid and word data
    crossword_grid = [
        [" ", " ", " ", " ", " "],
        ["#", "#", " ", "#", " "],
        ["#", " ", " ", " ", " "],
        [" ", "#", " ", " ", " "],
        [" ", " ", " ", " ", " "],
        [" ", "#", "#", " ", "#"],
    ]

    word_data = [
        ("1ACROSS", [(0, 0)], {"HOSES", "LASER", "SAILS",
"SHEET", "STEER"}),
        ("4ACROSS", [(2, 1)], {"HEEL", "HIKE", "KEEL", "KNOT",
"LINE"}),
        ("7ACROSS", [(3, 2)], {"AFT", "ALE", "EEL", "LEE",
"TIE"}),
        ("8ACROSS", [(4, 0)], {"HOSES", "LASER", "SAILS",
"SHEET", "STEER"}),
        ("2DOWN", [(0, 2)], {"HOSES", "LASER", "SAILS",
"SHEET", "STEER"}),
        ("3DOWN", [(0, 4)], {"HOSES", "LASER", "SAILS",
"SHEET", "STEER"}),
        ("5DOWN", [(2, 3)], {"HEEL", "HIKE", "KEEL", "KNOT",
"LINE"}),
        ("6DOWN", [(3, 0)], {"AFT", "ALE", "EEL", "LEE",
"TIE"}),
    ]

    # Call the crossword solver function
    CrosswordSolverTiny(crossword_grid, word_data)


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

Result of the above code execution.

```
(msds)
C:\Users\Ashish\PycharmProjects\CrossWordSolver>c:/Users/Ashish/msds/Scripts/python.exe
c:/Users/Ashish/PycharmProjects/CrossWordSolver/CrosswordSolverTiny.py

HOHES

##O#T

#HSKE

A#EEE

LASER

E##L#
```

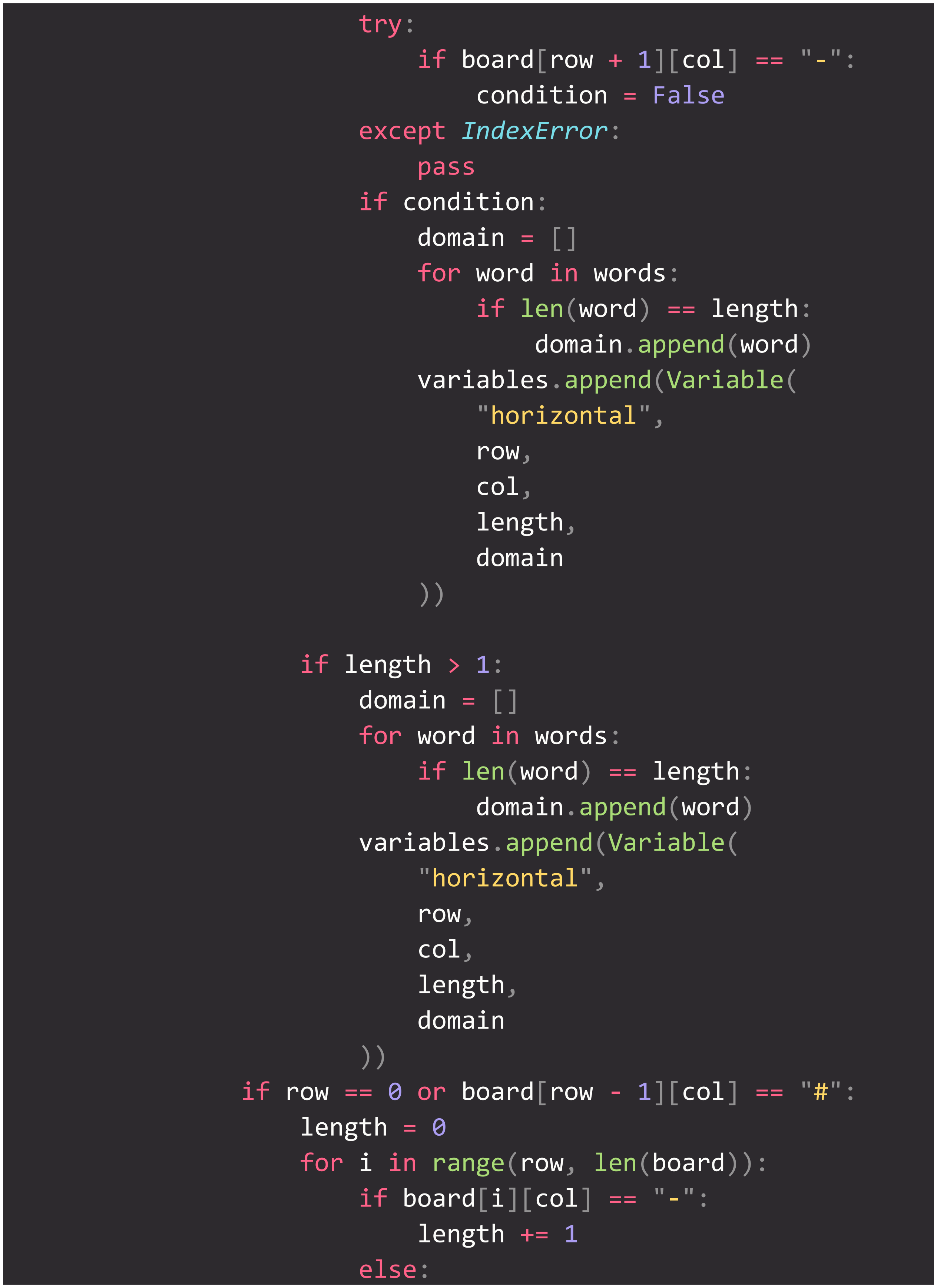
**---Time taken for code execution %s seconds ---0.00024819374084472656**

## Medium Puzzle

```python
import time

# Class to represent a variable in the crossword puzzle
class Variable:
    def __init__(self, direction, row, col, length, domain):
        self.word = ""
        self.direction = direction
        self.row = row
        self.col = col
        self.length = length
        self.domain = domain
        self.removed_domain = {}


# Function to display the crossword board
def ShowBoard(grid, assignment):
    board = grid.split("\n")
    board = [list(row) for row in board]
    for v in assignment:
        val = assignment[v]
        if v.direction == "horizontal":
            for i in range(v.length):
                board[v.row][v.col + i] = val[i]
        else:
            for i in range(v.length):
                board[v.row + i][v.col] = val[i]
    for row in board:
        print(" ".join(map(str, row)))


# Function to check if a given assignment satisfies the
constraints
def SatisfyConstraint(V, assignment, Vx, val):
    for v in V:
        Cxv = MakeConstraint(Vx, v)
        if v != Vx and v in assignment and Cxv:
            if val[Cxv[0]] != assignment[v][Cxv[1]]:
                return False
    return True


# Function to find an unassigned variable
def UnassignedVariable(V, assignment):
    unassigned = []
    for v in V:
        if v not in assignment:
            unassigned.append(v)

    unassigned.sort(key=lambda x: len(x.domain))
    return unassigned[0]


# Function to reduce the domain of variables based on
constraints
```

```python
def ForwardChecking(V, assignment, Vx, val):
    for v in V:
        Cxv = MakeConstraint(Vx, v)
        if v != Vx and v not in assignment and Cxv:
            v.domain = [word for word in v.domain if
val[Cxv[0]] == word[Cxv[1]]]


# Function to restore the original domain of variables
def OriginalDomain(V, assignment, Vx, val):
    for v in V:
        Cxv = MakeConstraint(Vx, v)
        if v != Vx and v not in assignment and Cxv:
            if v in Vx.removed_domain:
                for word in Vx.removed_domain[v]:
                    if val[Cxv[0]] != word[Cxv[1]]:
                        v.domain.append(word)
                        Vx.removed_domain[v].remove(word)


# Backtracking algorithm to solve the crossword puzzle with
forward checking
def BacktrackingAlgo(V, assignment):
    if len(assignment) == len(V):
        return True
    Vx = UnassignedVariable(V, assignment)
    for val in Vx.domain:
        if val in assignment.values():
            continue
        if SatisfyConstraint(V, assignment, Vx, val):
            assignment[Vx] = val
            ForwardChecking(V, assignment, Vx, val)
            result = BacktrackingAlgo(V, assignment)
            if result:
                return True
        assignment.pop(Vx, None)
        OriginalDomain(V, assignment, Vx, val)
    return False


# Function to create a constraint between two variables
def MakeConstraint(Vx, Vy):
    constraint = ()
    if Vx.direction != Vy.direction:
        if Vx.direction == "horizontal":
            if Vy.col >= Vx.col and Vy.col <= Vx.col +
Vx.length - 1:
                if Vx.row >= Vy.row and Vx.row <= Vy.row +
Vy.length - 1:
                    constraint = (Vy.col - Vx.col, Vx.row -
Vy.row)
        else:
            if Vy.row >= Vx.row and Vy.row <= Vx.row +
Vx.length - 1:
                if Vx.col >= Vy.col and Vx.col <= Vy.col +
Vy.length - 1:
                    constraint = (Vy.row - Vx.row, Vx.col -
Vy.col)
    return constraint


# Function to create arcs between variables based on
constraints
def MakeArc(V):
    arcs = []
    for i in range(len(V)):
        for j in range(i + 1, len(V)):
            if i != j:
                Cij = MakeConstraint(V[i], V[j])
                if len(Cij) > 0:
                    arcs.append((V[i], V[j], Cij))
    return arcs
```

```python
# Function to create variable objects from the crossword grid
and word list
def MakeVariables(grid, words):
    variables = []
    board = grid.split("\n")
    for row in range(len(board)):
        for col in range(len(board[row])):
            if board[row][col] == "-":
                if col == 0 or board[row][col - 1] == "#":
                    length = 0
                    for i in range(col, len(board[row])):
                        if board[row][i] == "-":
                            length += 1
                        else:
                            break
                    if length == 1:
                        condition = True
                        try:
                            if board[row][col + 1] == "-":
                                condition = False
                        except IndexError:
                            pass
                        try:
                            if board[row][col - 1] == "-" and
col - 1 >= 0:
                                condition = False
                        except IndexError:
                            pass
                        try:
                            if board[row - 1][col] == "-" and
row - 1 >= 0:
                                condition = False
                        except IndexError:
                            pass
```



```python
        try:
            if board[row + 1][col] == "-":
                condition = False
        except IndexError:
            pass
        if condition:
            domain = []
            for word in words:
                if len(word) == length:
                    domain.append(word)
            variables.append(Variable(
                "horizontal",
                row,
                col,
                length,
                domain
            ))

    if length > 1:
        domain = []
        for word in words:
            if len(word) == length:
                domain.append(word)
        variables.append(Variable(
            "horizontal",
            row,
            col,
            length,
            domain
        ))
if row == 0 or board[row - 1][col] == "#":
    length = 0
    for i in range(row, len(board)):
        if board[i][col] == "-":
            length += 1
        else:
```

```python
                            break
                    if length > 1:
                        domain = []
                        for word in words:
                            if len(word) == length:
                                domain.append(word)
                        variables.append(Variable(
                            "vertical",
                            row,
                            col,
                            length,
                            domain
                        ))
    return variables


# Function to get the crossword grid from a file
def GetGrid(file_path):
    with open(file_path) as file:
        return file.read()


# Main function to solve the crossword puzzle
def main():
    assignment = {}
    grid = GetGrid("grid_medium.txt")
    words = GetGrid("Words.txt").splitlines()
    words = [word.upper() for word in words]

    variables = MakeVariables(grid, words)
    variables.sort(key=lambda x: len(x.domain))
    BacktrackingAlgo(variables, assignment)
    ShowBoard(grid, assignment)


if __name__ == "__main__":
    start_time = time.time()
    main()
    print("\n---Time taken for code execution %s seconds ---" %
(time.time() - start_time))
```

## Result of the above code execution

```
(msds)
C:\Users\Ashish\PycharmProjects\CrossWordSolver>c:/Users/Ashish/msds/Scripts/python.exe
c:/Users/Ashish/PycharmProjects/CrossWordSolver/CrosswordSovlerMedium.py

# # A L F A L F A # # #

# # B # # # # # B # # A

# # A # # # # # B # # F

# # C # # A B B E Y # I

# # K # # # # A # # # E

# # # # A # # B A B E L

# # # # F # # B # A # D

# B # A F O U L # B # #

# A # # A # # E # O # #

A B A T I S # # # O # #

# E # # R # # # # N # #
```

**---Time taken for code execution 0.018278837203979492 seconds ---**

Hard Puzzle

```python
import time
import nltk
from collections import deque

# Load words from a file and filter for English words
def LoadDictionary(file_path):
    with open(file_path, 'r') as file:
        all_words = set(word.strip().lower() for word in
file.readlines())

    english_words = set(w.lower() for w in
nltk.corpus.words.words())

    return all_words.intersection(english_words)

# Check if a given word is valid based on the loaded dictionary
def IsValidWord(word, dictionary):
    return word.lower() in dictionary

# Check if a word can be placed on the board at a specific
location and direction
def IsValidLocation(board, row, col, word, direction,
dictionary):
    if direction == 'horizontal':
        for i in range(len(word)):
            if (
                col + i >= 13 or
                (board[row][col + i] != 0 and board[row][col +
i] != word[i]) or
                not IsValidWord(word[i], dictionary)
            ):
                return False
    elif direction == 'vertical':
        for i in range(len(word)):
            if (
                row + i >= 12 or
                (board[row + i][col] != 0 and board[row +
i][col] != word[i]) or
                not IsValidWord(word[i], dictionary)
            ):
                return False
    else:
        return False

    return True

# Solve the puzzle using backtracking
def SolvePuzzleBacktracking(board, dictionary,
remaining_words):
    empty = EmptyLocations(board)
    if not empty:
        return True

    row, col = empty

    for word in remaining_words[row][col].copy():
        for direction in ['horizontal', 'vertical']:
            if IsValidLocation(board, row, col, word,
direction, dictionary):
                PlaceWords(board, row, col, word, direction)
                # Update the remaining words after placing a
word
                UpdateWords(board, remaining_words, dictionary)

                if SolvePuzzleBacktracking(board, dictionary,
remaining_words):
                    return True

                RemoveWords(board, row, col, word, direction)

    return False

# Update the set of remaining words for each empty cell on the
board
def UpdateWords(board, remaining_words, dictionary):
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == 0:
                remaining_words[i][j] = GetValidWords(board, i,
j, dictionary)

# Get the set of valid words for a specific location on the
board
def GetValidWords(board, row, col, dictionary):
    valid_words = set()

    for word in dictionary:
        for direction in ['horizontal', 'vertical']:
            if IsValidLocation(board, row, col, word,
direction, dictionary):
                valid_words.add(word)

    return valid_words

# Find the first empty location on the board
def EmptyLocations(board):
```

```python
    for i in range(len(board)):
        for j in range(len(board[0])):
            if board[i][j] == 0:
                return (i, j)
    return None

# Place a word on the board at a specific location and
direction
def PlaceWords(board, row, col, word, direction):
    if direction == 'horizontal':
        for i in range(len(word)):
            board[row][col + i] = word[i]
    elif direction == 'vertical':
        for i in range(len(word)):
            board[row + i][col] = word[i]

# Remove a word from the board at a specific location and
direction
def RemoveWords(board, row, col, word, direction):
    if direction == 'horizontal':
        for i in range(len(word)):
            board[row][col + i] = 0
    elif direction == 'vertical':
        for i in range(len(word)):
            board[row + i][col] = 0

# Load the puzzle from a file
def LoadGrid(file_path):
    with open(file_path, 'r') as file:
        return [list(line.strip()) for line in
file.readlines()]

if __name__ == "__main__":
    start_time = time.time()

    # Load puzzle and dictionary
    pattern = LoadGrid("grid_hard.txt")
    dictionary = LoadDictionary("Words.txt")

    # Initialize the puzzle, replacing '-' with 0
    puzzle = [[0 if cell == '-' else cell for cell in row] for
row in pattern]

    # Initialize the remaining_words variable
    remaining_words = [
        [GetValidWords(puzzle, i, j, dictionary) if
puzzle[i][j] == 0 else set() for j in range(13)] for i in
range(12)
    ]

    # Solve the puzzle and print the result
    if SolvePuzzleBacktracking(puzzle, dictionary,
remaining_words):
        for row in puzzle:
            print(" ".join(map(str, row)))
        print("\n---Time taken for code execution %s seconds --
-" % (time.time() - start_time))
    else:
        print("No solution exists.")
```

## Results of the above code execution.

```
(msds)
C:\Users\Ashish\PycharmProjects\CrossWordSolver>c:/Users/Ashish/msds/Scripts/python.exe
c:/Users/Ashish/PycharmProjects/CrossWordSolver/CrosswordSolverHard.py

# # g n g # # # g n g # #

# g o e a b # b a e o g #

# o b e n o # o n e b o #

n b b # g r n r g # b b n
```

```
e b l b # a e a # b l b e

e l e o # x e x # o e l e

# e o r g # # # g r o e #

# # x a a n # n a a x # #

# # # x n e o e n x # # #

# # # # g e x e g # # # #

# # # # # a s h # # # # #

# # # # # # g # # # # # #
```

**---Time taken for code execution 23.934542894363403 seconds ---**