Breadth First Search

```
graph = {
 'A' : ['B','C'],
 'B' : ['D', 'E'],
 'C' : ['F', 'G'],
 'D' : [],
 'E' : [],
 'F' : [],
 'G' : []
}
visited = []
queue = []

def bfs(visited, graph, node):
    visited.append(node)
    queue.append(node)
    while queue:
      s = queue.pop(0)
      if(s=='G'):
        break
      print (s, end = " --> ")
      for neighbour in graph[s]:
            if neighbour not in visited:
                visited.append(neighbour)
                queue.append(neighbour)

bfs(visited, graph, 'A')
```

```
A --> B --> C --> D --> E --> F -->
```

Depth First Search

```
# Using a Python dictionary to act as an adjacency list
GRAPH = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F','G'],
    'D' : ['H'],
    'E' : [],
    'F' : [],
    'G':  []
}

visited = [] # Set to keep track of visited nodes.

def dfs(GRAPH, node, visitedSet = None, path = None):

    if visitedSet is None:
        visitedSet = set()
    if path is None:
        path = []

    visitedSet.add(node)
    path.append(node)
    if node in GRAPH:
        for neighbor in GRAPH[node]:
            if neighbor not in visitedSet:
                dfs(GRAPH, neighbor, visitedSet, path)
    return path

print(dfs(GRAPH, 'A'))
```

```
['A', 'B', 'D', 'H', 'E', 'C', 'F', 'G']
```

Greedy best-first search algorithm

```
graph = {
'A':[('B',12), ('C',4)],
'B':[('D',7), ('E',3)],
'C':[('F',8), ('G',2)],
'D':[],
'E':[('H',0)],
```

```
'F':[('H',0)],
'G':[('H',0)]
}



def bfs(start, target, graph, queue=[], visited=[]):
    if start not in visited:
      print(start)
      visited.append(start)
    queue=queue+[x for x in graph[start] if x[0][0] not in visited]
    queue.sort(key=lambda x:x[1])
    if queue[0][0]==target:
        print(queue[0][0])
    else:
        processing=queue[0]
        queue.remove(processing)
        bfs(processing[0], target, graph, queue, visited)
bfs('A', 'H', graph)
```

```
A
C
G
H
```

A* search algorithm

```
graph=[['A','B',1,3],
       ['A','C',2,4],
       ['A','H',7,0],
       ['B','D',4,2],
       ['B','E',6,6],
       ['C','F',3,3],
       ['C','G',2,1],
       ['D','E',7,6],
       ['D','H',5,0],
       ['F','H',1,0],
       ['G','H',2, 0]]
temp = []
temp1 = []
for i in graph:
    temp.append(i[0])
    temp1.append(i[1])
nodes = set(temp).union(set(temp1))
def A_star(graph, costs, open, closed, cur_node):
  if cur_node in open:
        open.remove(cur_node)
  closed.add(cur_node)
  for i in graph:
        if(i[0] == cur_node and costs[i[0]]+i[2]+i[3] < costs[i[1]]):
            open.add(i[1])
            costs[i[1]] =  costs[i[0]]+i[2]+i[3]
            path[i[1]] = path[i[0]] + ' -> ' + i[1]
  costs[cur_node] = 999999
  small = min(costs, key=costs.get)
  if small not in closed:
        A_star(graph, costs, open,closed, small)
costs = dict()
temp_cost = dict()
path = dict()
for i in nodes:
    costs[i] = 999999
    path[i] = ' '
open = set()
closed = set()
start_node = input("Enter the Start Node: ")
open.add(start_node)
path[start_node] = start_node
costs[start_node] = 0
A_star(graph, costs, open, closed, start_node)
goal_node = input("Enter the Goal Node: ")
print("Path with least cost is: ",path[goal_node])
```

```
Enter the Start Node: A
Enter the Goal Node: H
Path with least cost is:  A -> C -> G -> H
```

Tic-Tac-Toe

```
import sys
import random
class TicTacToeGame:
    def __init__(self, rows:int, columns:int, goal:int, max_depth:int=4):

        self.state = []
        self.tiles = {}
        self.inverted_tiles = {}
        tile = 0
        for y in range(rows):
            row = []
            for x in range(columns):
                row += '.'
                tile += 1
                self.tiles[tile] = (y, x)
                self.inverted_tiles[(y, x)] = tile
            self.state.append(row)
        self.goal = goal
        self.vectors = [(1,0), (0,1), (1,1), (-1,1)]
        self.rows = rows
        self.columns = columns
        self.max_row_index = rows - 1
        self.max_columns_index = columns - 1
        self.max_depth = max_depth
        self.winning_positions = []
        self.get_winning_positions()
        self.player = random.choice(['X', 'O'])
    def get_winning_positions(self):
        for y in range(self.rows):
            for x in range(self.columns):
                for vector in self.vectors:
                    sy, sx = (y, x)
                    dy, dx = vector
                    counter = 0
                    positions = []
                    while True:
                        positions.append(self.inverted_tiles.get((sy, sx)))
                        if (len(positions) == self.goal):
                            self.winning_positions.append(positions)
                            break
                        sy += dy
                        sx += dx
                        if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) > self.max_columns_index):
                            break
    def play(self):
        result = None
        print('Starting board')
        while True:
            self.print_state()
            if (self.player == 'X'): # AI
                print('Player X moving (AI) ...')
                max, py, px, depth = self.max(-sys.maxsize, sys.maxsize)
                print('Depth: {0}'.format(depth))
                if(depth > self.max_depth):
                    py, px = self.get_best_move()
                self.state[py][px] = 'X'
                result = self.game_ended()
                if(result != None):
                    break
                self.player = 'O'
            elif (self.player == 'O'):

                print('Player O moving (Human) ...')
                min, py, px, depth = self.min(-sys.maxsize, sys.maxsize)
                print('Depth: {0}'.format(depth))
                if(depth > self.max_depth):
                    py, px = self.get_best_move()
                print('Recommendation: {0}'.format(self.inverted_tiles.get((py, px))))
                number = int(input('Make a move (tile number): '))
                tile = self.tiles.get(number)
                if(tile != None):
                    py, px = tile
                    self.state[py][px] = 'O'
                    result = self.game_ended()
                    if(result != None):
                        break
                    self.player = 'X'
                else:
```

```
                print('Move is not legal, try again.')
        self.print_state()
        print('Winner is player: {0}'.format(result))
    def get_best_move(self):
        heuristics = {}
        empty_cells = []
        for y in range(self.rows):
            for x in range(self.columns):
                if (self.state[y][x] == '.'):
                    empty_cells.append((y, x))
        for empty in empty_cells:
            number = self.inverted_tiles.get(empty)
            for win in self.winning_positions:
                if(number in win):
                    player_x = 0
                    player_o = 0
                    start_score = 1
                    for box in win:
                        y, x = self.tiles[box]
                        if(self.state[y][x] == 'X'):
                            player_x += start_score if self.player == 'X' else start_score * 2
                            start_score *= 10
                        elif (self.state[y][x] == 'O'):
                            player_o += start_score if self.player == 'O' else start_score * 2
                            start_score *= 10
                    if(player_x == 0 or player_o == 0):
                        score = max(player_x, player_o) + start_score
                        if(heuristics.get(number) != None):
                            heuristics[number] += score
                        else:
                            heuristics[number] = score
        best_move = random.choice(empty_cells)
        best_count = -sys.maxsize
        for key, value in heuristics.items():
            if(value > best_count):
                best_move = self.tiles.get(key)
                best_count = value
        return best_move
    def game_ended(self) -> str:
        result = self.player_has_won()
        if(result != None):
            return result
        for y in range(self.rows):
            for x in range(self.columns):
                if (self.state[y][x] == '.'):
                    return None
        return 'It is a tie!'

    def player_has_won(self) -> str:

        for y in range(self.rows):
            for x in range(self.columns):

                for vector in self.vectors:

                    sy, sx = (y, x)
                    dy, dx = vector
                    steps = 0
                    player_x = 0
                    player_o = 0
                    while steps < self.goal:
                        steps += 1
                        if(self.state[sy][sx] == 'X'):
                            player_x += 1
                        elif(self.state[sy][sx] == 'O'):
                            player_o += 1
                        sy += dy
                        sx += dx

                        if(sy < 0 or abs(sy) > self.max_row_index or sx < 0 or abs(sx) > self.max_columns_index):
                            break
                    if(player_x >= self.goal):
                        return 'X'
                    elif(player_o >= self.goal):
                        return 'O'
        return None
    def min(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):

        min_value = sys.maxsize
        by = None
        bx = None
        result = self.game_ended()
        if(result != None):
```

```
        if(result != None):
            if result == 'X':
                return 1, 0, 0, depth
            elif result == 'O':
                return -1, 0, 0, depth
            elif result == 'It is a tie!':
                return 0, 0, 0, depth
        elif(depth > self.max_depth):
            return 0, 0, 0, depth
        for y in range(self.rows):
            for x in range(self.columns):
                if (self.state[y][x] == '.'):
                    self.state[y][x] = 'O'
                    max, max_y, max_x, depth = self.max(alpha, beta, depth + 1)

                    if (max < min_value):
                        min_value = max
                        by = y
                        bx = x

                    self.state[y][x] = '.'
                    if (min_value <= alpha):
                        return min_value, bx, by, depth
                    if (min_value < beta):
                        beta = min_value
        return min_value, by, bx, depth
    def max(self, alpha:int=-sys.maxsize, beta:int=sys.maxsize, depth:int=0):
        max_value = -sys.maxsize
        by = None
        bx = None
        result = self.game_ended()
        if(result != None):
            if result == 'X':
                return 1, 0, 0, depth
            elif result == 'O':
                return -1, 0, 0, depth
            elif result == 'It is a tie!':
                return 0, 0, 0, depth
        elif(depth > self.max_depth):
            return 0, 0, 0, depth
        for y in range(self.rows):
            for x in range(self.columns):
                if (self.state[y][x] == '.'):

                    self.state[y][x] = 'X'
                    min, min_y, min_x, depth = self.min(alpha, beta, depth + 1)
                    if (min > max_value):
                        max_value = min
                        by = y
                        bx = x
                    self.state[y][x] = '.'
                    if (max_value >= beta):
                        return max_value, bx, by, depth
                    if (max_value > alpha):
                        alpha = max_value
        return max_value, by, bx, depth
    def print_state(self):
        for y in range(self.rows):
            print('| ', end='')
            for x in range(self.columns):
                if (self.state[y][x] != '.'):
                    print(' {0}  | '.format(self.state[y][x]), end='')
                else:
                    digit = str(self.inverted_tiles.get((y,x))) if len(str(self.inverted_tiles.get((y,x)))) > 1 else ' ' + str(self.inver
                    print('{0}  | '.format(digit), end='')
            print()
        print()
def main():
    game = TicTacToeGame(3, 3, 3, 1000)
    game.play()
if __name__ == "__main__": main()
```

```
 Player O moving (Human) ...
```

```
|  4  |  X  |  6  |
|  7  |  8  |  9  |

Player O moving (Human) ...
Depth: 268
Recommendation: 8
Make a move (tile number): 8
|  O  |  X  |  3  |
|  4  |  X  |  6  |
|  7  |  O  |  9  |

Player X moving (AI) ...
Depth: 104
|  O  |  X  |  3  |
|  X  |  X  |  6  |
|  7  |  O  |  9  |

Player O moving (Human) ...
Depth: 32
Recommendation: 6
Make a move (tile number): 6
|  O  |  X  |  3  |
|  X  |  X  |  O  |
|  7  |  O  |  9  |

Player X moving (AI) ...
Depth: 11
|  O  |  X  |  X  |
|  X  |  X  |  O  |
|  7  |  O  |  9  |

Player O moving (Human) ...
Depth: 4
Recommendation: 7
Make a move (tile number): 7
|  O  |  X  |  X  |
|  X  |  X  |  O  |
|  O  |  O  |  9  |

Player X moving (AI) ...
Depth: 1
|  O  |  X  |  X  |
|  X  |  X  |  O  |
|  O  |  O  |  X  |

Winner is player: It is a tie!
```