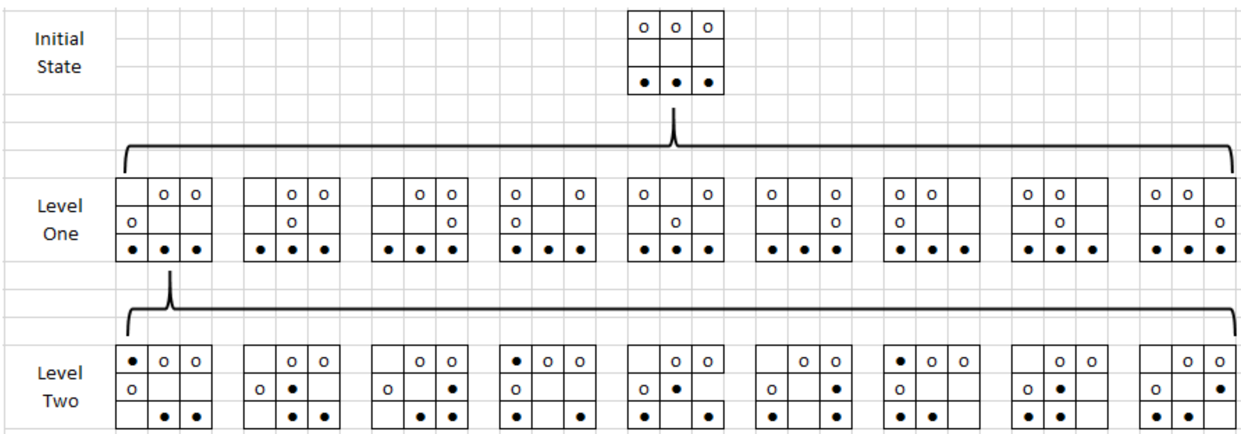
Group Name: Deus Ex Machina (Undergraduate)

Yukun Huang	V00804445	kennethhyk@gmail.com
Cameron Mulgrew-MacFarlane	V00775582	Cammac93@gamil.com
Willy Su Yep	V00795480	Op4977377832@gmail.com
Sam Wheating	V00816465	SamWheating@yahoo.ca
Matt Johnson	V00757521	Matthewj@uvic.ca

CENG 420 Assignment #1

Q1.1) The branching factor would be 9, and it is uniform.

Q1.2)



Q1.3) I think A* search is a good enough approach for this game. Since this is only a 3 by 3 board game, the search space won't be too big. Also, it is likely to enter a loop or cycle, therefore, we cannot use DFS. By giving a good heuristic function, the program should be able to determine when to play defensively (usually only one option) or aggressively (need to find out which move can lead to a goal or closer to a goal).

Q1.4)

Heuristic function:
- each token should have a score, and this score may depend on the following:
  1. is this token connected to another token (1 point)
  2. is this token blocking the opponent's tokens (5 points)
  3. score schema for opponent's tokens will be the same, but negative

- The heuristic function for each board state will return sum of the scores of all tokens (including opponents), higher score -> higher chance to win

The following code is separated into two files: sega1.py contains a framework for command-line sega and sega2.py implements the heuristic function and plays a game against a human player (python 3.4).

Code: (https://github.com/CamMacFarlane/CENG420/tree/master/Assignment1/Q1.4%20Code)

```
"""
Sega1.py - Basic framework for command-line sega
"""
import random
import time
x_len = 3
y_len = 3
horzLine = "----"*x_len
debug = True
mainBoard = [[' ' for k in range(x_len)] for k in range(y_len)]


#prints the state passed
def printState(state):
    print("Y\X 0   1   2")
    print(" ",horzLine)
    for k in range(x_len):
        print(k ,"| ", end = "")
        for i in range(y_len):
            print(state[k][i], "| ", end = "")
        print()
        print(" ",horzLine)

#returns True if the move was valid
def move(player, curX, curY, newX, newY, state=mainBoard, quiet = False):
    ret  = False
    if(state[curY][curX] != player):
        if(debug):
            print("INVALID MOVE: not your piece")
    elif(state[newY][newX] != ' '):
        if(debug):
            print("INVALID MOVE: space is not free")
    else:
        swap = state[curY][curX]
        state[curY][curX] = state[newY][newX]
        state[newY][newX] = swap
        if not quiet: print("\nPlayer", player, "moves: (",curX, "," , curY
, ")", "to", "(", newX, ",", newY, ")\n")
        ret = True
    return ret

def checkGoalState(player, state=mainBoard, quiet=False):
    #horizontal
    for k in range(y_len):
        if(state[k] == [player]*x_len):
            if not quiet: print("\nhorizontal line for ", player,"!")
            return True

    #vertical

    for i in range(x_len):
        col = ""
        for k in range(y_len):
            col += state[k][i]
```

```
            if(col == player*x_len):
                if not quiet: print("\nvertical line for ", player,"!")
                return True

    #diagonal
    if(state[0][0] == player):
        if(state[1][1] == player):
            if(state[2][2] == player):
                if not quiet: print("\nDiagonal line for ", player,"!")
                return True
    if(state[0][2] == player):
        if(state[1][1] == player):
            if(state[2][0] == player):
                if not quiet: print("\nDiagonal line for ", player,"!")
                return True

#populates board to beginning game state
def populateBoard(state=mainBoard):
    for i in range(x_len):
        state[0][i] = "X"
        state[x_len - 1 ][i] = 'O'


def randomDance(state=mainBoard):
    populateBoard()
    printState(state)
    while(True):
        Attempt = 0
        time.sleep(0.1)

        while(move('O', random.randint(0,2), random.randint(0,2),
random.randint(0,2),random.randint(0,2)) == False):
            Attempt = Attempt +1
            print("Attempt = ", Attempt , "\r", end="")

        printState(state)
        if(checkGoalState('O')):
            time.sleep(2)

        Attempt = 0
        time.sleep(0.1)

        while(move('X', random.randint(0,2), random.randint(0,2),
random.randint(0,2),random.randint(0,2)) == False ):
            Attempt = Attempt +1
            print("Attempt = ", Attempt , "\r", end="")

        printState(state)
        if(checkGoalState('X')):
            time.sleep(2)
```

```
"""
Sega2.py - implementation of heursitic function
"""

import sega1
import time
import random
x_len = sega1.x_len
y_len = sega1.y_len
gameBoard = sega1.mainBoard
debug = False

"""
Blocking detection: horizontal, vertical, diagonal:
Determines if the piece in question is blocking a potential line. (2 out of
3 in a row).
If it is blocking, it adds 5 to the heuristic score (subject to change), as
this is a very important move.
Functions work by iterating through the row/column/diagonal and looking for
1 'player' and n-1 'opponents'
"""

def blocking_col(player, state, x):     #determines if a piece is blocking
opponent's horizontal victory
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    count_player = 0
    count_opponent = 0
    for i in range(y_len):
        if state[i][x] == opponent:
            count_opponent += 1
        if state[i][x] == player:
            count_player += 1
    if((count_opponent == (y_len-1)) & (count_player == 1)):          # if
the piece is blocking
        return 5
    else:
        return 0

def blocking_row(player, state, y):     # determines if player is blocking
opponent's veritcal victory
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    count_player = 0
    count_opponent = 0
    for i in range(x_len):
        if state[y][i] == opponent:
            count_opponent += 1
        if state[y][i] == player:
            count_player += 1
```

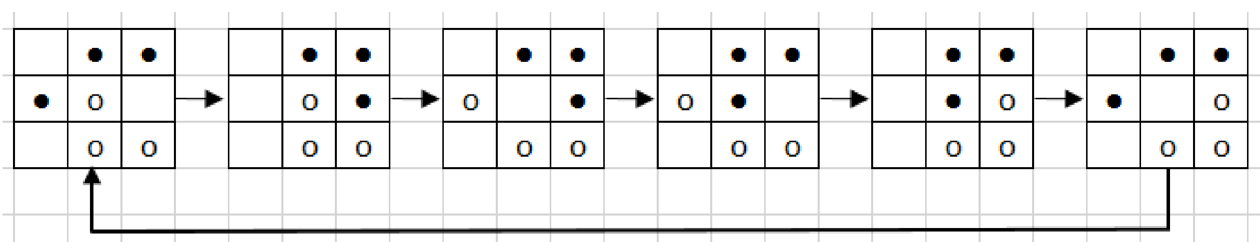
```
    if((count_opponent == (x_len-1)) & (count_player == 1)):
        return 5
    else:
        return 0

def blocking_diag(player, state, x, y):     #assumes a square board
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    #diagonal uppper left to lower right
    if(x==y):
        count_player = 0
        count_opponent = 0
        for i in range(x_len):
            if state[i][i] == opponent:
                count_opponent += 1
            if state[i][i] == player:
                count_player += 1
        if((count_opponent == (x_len-1)) & (count_player == 1)):          #
if the piece is blocking a row
            return 5
        else:
            return 0

    # diagonal lower left to upper right
    if((x+y) == (x_len-1)):
        count_player = 0
        count_opponent = 0
        for i in range(x_len):
            if state[y_len-i-1][i] == opponent:
                count_opponent += 1
            if state[y_len-i-1][i] == player:
                count_player += 1
        if((count_opponent == (x_len-1)) & (count_player == 1)):          #
if the piece is blocking
            return 5
        else:
            return 0
    return 0

"""
num_inrow / num_incol:
Counts the number of other pieces in the same row or column.
Adds 2 to the heuristic score for each pair of pieces in the same
row/column.
"""

def num_inrow(player, state, x, y):    # returns the number of other pieces
in the same column
    if state[y][x] != player:
        print("error, player not in spot")
    else:
        count = 0
        for i in range(x_len):
            if state[y][i] == player:
                count += 1
```

```
        return count-1

def num_incol(player, state, x, y):         # returns number of other pieces
in the same column
    if state[y][x] != player:
        print("error, player not in spot")
        return 0
    else:
        count = 0
        for i in range(y_len):
            if state[i][x] == player:
                count += 1
        return count-1

def num_diag(player, state, x, y):
    if((x != y) & ((x+y) != (x_len-1))):       # skip if the piece is not
along a diagonal line
        return 0
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    count = 0
    if(x==y):
        count -= 1
        for i in range(x_len):
            if state[i][i] == player:
                count += 1
    if((x+y) == (x_len-1)):
        count -= 1
        for i in range(x_len):
            if state[y_len-i-1][i] == player:
                count += 1
    return count

"""
Score:
Calls the different functions and totals the score.
Adds 1000 if the state is a goal state, to ensure the heuristic search
algorithm favours this state.
"""

def score(state, player):
    score = 0
    for x in range(x_len):
        for y in range(y_len):
            if state[y][x] == player:
                score += ( num_inrow(player, state, x, y) +
num_incol(player, state, x, y) + num_diag(player, state, x, y) +
blocking_row(player, state, y) + blocking_col(player, state, x) +
blocking_diag(player, state, x, y))
    if(sega1.checkGoalState(player, state, quiet=True) == True):
        score += 1000
    return score

"""
Heuristic function.
```

```
Calculates score(player) - score(opponent) as a final heuristic score.
This encourages that the selection of the next state to benefit the player
as much as possible, without setting up the opponent for a victory.
"""

def heuristic(state, player):
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    return(score(state, player) - score(state, opponent))


#PERMUTATIONS AND MOVE EVALUATION

def buildPermutations(player, state):  # returns a list of the possible next
states
    spaces = []
    filled = []
    boards = []
    if player=='X':
        opponent='O'
    else:
        opponent = 'X'
    for x in range(x_len):              # generates lists of coordinates for
empty spaces and player-occupied spaces
        for y in range(y_len):
            if state[y][x] == ' ':
                spaces.append([x,y])
            elif state[y][x] == player:
                filled.append([x,y])

    for piece in range(len(filled)):      # builds every possible next-state
and stores boards in an array
        for space in range(len(spaces)):
            tmp_board = [x[:] for x in state]
            sega1.move(player, filled[piece][0], filled[piece][1],
spaces[space][0], spaces[space][1], state=tmp_board, quiet=True)
            boards.append(tmp_board)
    return boards

def analyzePermutations(player, states, board):                 # takes the
heuristic score of each optional state and moves to the highest scoring
    scores = [' ' for k in range(len(states))]
    for state in range(len(states)):
        scores[state] = heuristic(states[state], player)
    selected_state = scores.index(max(scores))
    print("\nMy turn!\n")
    if(debug):
        print("\n\nAnalyzing Options:\n\n")
        for i in range(len(states)):
            if(debug):
                sega1.printState(states[i])
                print("\n score: ", scores[i], "\n")
        print("\n\nSelected State #%i\n\n" % selected_state)
    return states[selected_state]                               # returns
new gameboard
```

```
# start game
# loops and alternates human and computer moves until there is a winner.

sega1.populateBoard()
print("\nStarting Game! 3x3 Sega!\n")
sega1.printState(gameBoard)
firstmove = True
while(not sega1.checkGoalState('X', state=gameBoard, quiet=True) and not
sega1.checkGoalState('O', state=gameBoard, quiet=True) or firstmove):
    firstmove = False
    # take and perform user's move
    moves = input("\nenter to and from coordinates for X, 4 numbers,
separated by spaces: \n\n")
    move = moves.split()
    sega1.move('X', int(move[0]), int(move[1]), int(move[2]), int(move[3]),
gameBoard)
    sega1.printState(gameBoard)
    if(sega1.checkGoalState('X', state=gameBoard)):
        print("\n\nX wins!\n")
        break
    #computer's turn
    boards = buildPermutations('O', gameBoard)
    new_board = analyzePermutations('O', boards, gameBoard)
    gameBoard = [x[:] for x in new_board]
    sega1.printState(gameBoard)
    if(sega1.checkGoalState('O', state=gameBoard)):
        print("\n\nO wins!\n")
        break
```

Q1.5) It is possible to have cycles, one example would be:



Q 2.1) Suggested algorithm: Traverse the tree in a DFS fashion storeing each node in a stack as well as the sum of the edges. When a leaf node is reached record the queue and the sum as an attack scenario and the cost respectively. Backtracking pops nodes from the queue and subtracts the poppped node's edges from the sum of the cost.

Q 2.2) Psuedocode:

```
#is this a node a viable option?
is_node_valid(node):
   return ((weight of the next node * 2) < our remaining funds)

#returns a value indicating the proximity to the asset, if the value is less than 1, the closer the value to
1 the closer we may be to the asset
proximity_to_asset(asset, path):
   proximity = (path.sum/asset.value)

proximity_to_asset(asset, path, node):
   proximity = (path.sum + node /asset.value)

if we cannot see an asset:

   for node in visible_nodes:
      if(is_node_valid(node)):
         list_of_viable_nodes += node

   max_prox = 0
   best_node = ''

   for node in list_of_viable_nodes:
      for asset in assests:
         prox = proximity_to_asset(asset, path, node)
         if prox > max_prox:
            max_prox = prox
            best_node = node

   got_to_node(best_node)

else:
   go_to_node(asset)
```

Q4) Choose one of the following algorithms that we learned, state its time complexity, space complexity, explain how it works, and when to use this algorithm:

- greedy algorithm
- A*
- DFS
- BFS
- etc.