Maze Solving AI Using the Genetic Algorithm

December 9, 2022

[83]:
```python
from enum import IntEnum
import random
```

[84]:
```python
class Directions(IntEnum):
    NORTH = 0
    SOUTH = 1
    EAST = 2
    WEST = 3

direction = {'n': Directions.NORTH, 's': Directions.SOUTH, 'e': Directions.
 ↪EAST, 'w': Directions.WEST}
directionsSet = (Directions.NORTH, Directions.SOUTH, Directions.EAST,␣
 ↪Directions.WEST)
directionChar = {Directions.NORTH : 'N', Directions.SOUTH : 'S', Directions.
 ↪EAST : 'E', Directions.WEST : 'W'}

def DistanceForm(p1, p2):
    x1 = p1[0]
    y1 = p1[1]

    x2 = p2[0]
    y2 = p2[1]

    return (((x2 - x1)**2) + ((y2 - y1)**2))**.5

def pathStr(path):
    res = ""
    res += directionChar[Directions(path[0])]
    for i in range(1, len(path)):
        res += " "
        res += directionChar[Directions(path[i])]
    return res
```

[85]:
```python
class Cell:
    def __init__(self, cellNum = 0, isStart = False, isFinish = False):
        self.north = None
        self.south = None
        self.east = None
        self.west = None
        self.isStart = isStart
        self.isFinish = isFinish
        self.cellNum = cellNum

    def setDirections(self, north = None, south = None, east = None, west = None):
        self.north = north
        self.south = south
        self.east = east
        self.west = west

    def getCellNum(self):
        return self.cellNum

    def finishStatus(self):
        return self.isFinish

    def startStatus(self):
        return self.isStart

    def makeStart(self):
        self.isStart = True

    def makeFinish(self):
        self.isFinish = True

    def getNorth(self):
        return self.north

    def getSouth(self):
        return self.south

    def getEast(self):
        return self.east

    def getWest(self):
        return self.west

    def setNorth(self, cell):
        self.north = cell

    def setSouth(self, cell):
        self.south = cell

    def setEast(self, cell):
        self.east = cell

    def setWest(self, cell):
        self.west = cell

    def connectCells(self, other, direction):
        match direction:
            case Directions.NORTH:
                self.setNorth(other)
                other.setSouth(self)
            case Directions.SOUTH:
                self.setSouth(other)
                other.setNorth(self)
            case Directions.EAST:
                self.setEast(other)
                other.setWest(self)
            case Directions.WEST:
                self.setWest(other)
                other.setEast(self)

    def hasNorth(self):
        return self.north is not None

    def hasSouth(self):
        return self.south is not None

    def hasEast(self):
        return self.east is not None

    def hasWest(self):
        return self.west is not None

    def getCoords(self, nrows, ncols):
        row = self.cellNum % nrows
        col = ncols - (self.cellNum // ncols)
        return (row, col)

    def hasDir(self, direction):
        match direction:
            case Directions.NORTH:
                return self.hasNorth()
            case Directions.SOUTH:
                return self.hasSouth()
            case Directions.EAST:
                return self.hasEast()
            case Directions.WEST:
                return self.hasWest()
            case _:
                raise ValueError

    def atDeadEnd(self, prevDir):
        match prevDir:
            case Directions.NORTH:
                return not(self.hasSouth()) and not(self.hasEast()) and␣
↪not(self.hasWest())
            case Directions.SOUTH:
                return not(self.hasNorth()) and not(self.hasEast()) and␣
↪not(self.hasWest())
            case Directions.EAST:
                return not(self.hasNorth()) and not(self.hasSouth()) and␣
↪not(self.hasWest())
            case Directions.WEST:
                return not(self.hasNorth()) and not(self.hasSouth()) and␣
↪not(self.hasEast())
            case _:
                raise ValueError
```

[92]:
```
class Maze:
    def __init__(self, nrows = 0, ncols = 0):
        self.startCell = None
        self.finishCell = None
        self.cells = []
        self.runningumber = 0
        self.nrows = nrows
        self.ncols = ncols

    def getStartCell(self):
        return self.startCell

    def setRowsCols(self, nrows, ncols):
        self.nrows = nrows
        self.ncols = ncols

    def getRowsCols(self):
        return (self.nrows, self.ncols)

    def __len__(self):
        return len(self.cells)

    def __getitem__(self, index):
        return self.cells[index]

    def __setitem__(self, index, value):
        self.cells[index] = value
    def __contains__(self, index):
        return index in self.cells

    def __iter__(self):
        return self.cells.__iter__()

    '''
    10x10 Maze:
    Start Cell: 0
    Finish Cell: 99
    0  1--2  3  4--5  6--7  8  9
    |  |  |  |  |  |  |     |  |
    10-11 12-13-14 15-16-17-18-19
    |  |  |  |        |     |  |
    20-21 22 23-24-25 26-27 28-29
        |  |     |  |  |  |  |  |
    30-31-32-33 34 35 36 37-38 39
        |     |     |     |
    40 41-42 43 44-45 46-47-48-49
    |        |  |     |     |
    50-51 52-53 54-55-56-57 58-59
    |     |  |  |  |        |
    60-61 62 63-64 65-66-67 68 69
    |     |  |     |     |     |
    70-71 72-73 74-75-76 77 78-79
    |  |     |  |     |  |  |  |
    80 81-82 83-84-85 86 87-88 89
        |  |     |  |     |     |
    90-91 92-93 94 95 96-97-98 99
    '''
    def build10x10Maze(self):
        for i in range(100):
            self.cells.append(Cell(cellNum=i))
        with open("maze_file.txt", "r+") as f:
            for line in f:
                l = line.strip().split()
                for c in l:
                    cleaned = c[1:-1]
                    contents = cleaned.split(",")
                    cellNum = int(contents[0])
                    dirs = contents[1:]
                    for dir in dirs:
                        match dir:
                            case 'N':
                                self[cellNum].connectCells(self[cellNum-10],␣
↪Directions.NORTH)
```

```
                                case 'S':
                                    self[cellNum].connectCells(self[cellNum+10],␣
↪Directions.SOUTH)
                                case 'E':
                                    self[cellNum].connectCells(self[cellNum+1],␣
↪Directions.EAST)
                                case 'W':
                                    self[cellNum].connectCells(self[cellNum-1],␣
↪Directions.WEST)
                                case _:
                                    raise KeyError
        self[0].makeStart()
        self.startCell = self[0]
        self[99].makeFinish()
        self.finishCell = self[99]


    '''
    3x3 Maze:
    Start Cell: 0
    Finish Cell: 8
    0-1-2
    |
    3-4-5
    |   |
    6-7 8
    '''
    def build3x3Maze(self):
        cell0 = Cell(0, isStart=True)
        cell1 = Cell(1)
        cell2 = Cell(2)
        cell3 = Cell(3)
        cell4 = Cell(4)
        cell5 = Cell(5)
        cell6 = Cell(6)
        cell7 = Cell(7)
        cell8 = Cell(8, isFinish=True)
        #cell0.south = cell3
        cell0.east = cell1

        cell1.south = cell4
        cell1.east = cell2
        cell1.west = cell0

        #cell2.south = cell5
        cell2.west = cell1
        #cell3.north = cell0
        cell3.east = cell4
        cell3.south = cell6

        cell4.north = cell1
        #cell4.south = cell7
        cell4.east = cell5
        cell4.west = cell3

        #cell5.north = cell2
        cell5.south = cell8
        cell5.west = cell4

        cell6.north = cell3
        cell6.east = cell7

        #cell7.north = cell4
        #cell7.east = cell8
        cell7.west = cell6

        cell8.north = cell5
        #cell8.west = cell7

        self.cells.append(cell0)
        self.cells.append(cell1)
        self.cells.append(cell2)
        self.cells.append(cell3)
        self.cells.append(cell4)
        self.cells.append(cell5)
        self.cells.append(cell6)
        self.cells.append(cell7)
        self.cells.append(cell8)

        self.startCell = cell0
        cell0.makeStart()
        self.finishCell = cell8
        cell8.makeFinish()

    '''
    5x5 Maze:
    Start Cell: 0
    Finish Cell: 24
    0--1--2  3  4
       |  |  |  |
    5--6  7--8--9
    |  |     |
    10-11 12-13-14
```

```
|        |  |
15-16 17-18 19
|  |  |  |  |
20 21-22 23-24
'''
def build5x5Maze(self):
    cell0 = Cell(0, isStart=True)
    cell1 = Cell(1)
    cell2 = Cell(2)
    cell3 = Cell(3)
    cell4 = Cell(4)
    cell5 = Cell(5)
    cell6 = Cell(6)
    cell7 = Cell(7)
    cell8 = Cell(8)
    cell9 = Cell(9)
    cell10 = Cell(10)
    cell11 = Cell(11)
    cell12 = Cell(12)
    cell13 = Cell(13)
    cell14 = Cell(14)
    cell15 = Cell(15)
    cell16 = Cell(16)
    cell17 = Cell(17)
    cell18 = Cell(18)
    cell19 = Cell(19)
    cell20 = Cell(20)
    cell21 = Cell(21)
    cell22 = Cell(22)
    cell23 = Cell(23)
    cell24 = Cell(24, isFinish=True)

    #cell0.south = cell5
    cell0.east = cell1

    cell1.south = cell6
    cell1.east = cell2
    cell1.west = cell0

    cell2.south = cell7
    #cell2.east = cell3
    cell2.west = cell1

    cell3.south = cell8
    #cell3.east = cell4
    #cell3.west = cell2
```

```
        cell4.south = cell9
        #cell4.west = cell3

        #cell5.north = cell0
        cell5.south = cell10
        cell5.east = cell6

        cell6.north = cell1
        cell6.south = cell11
        #cell6.east = cell7
        cell6.west = cell5

        cell7.north = cell2
        #cell7.south = cell12
        cell7.east = cell8
        #cell7.west = cell6

        cell8.north = cell3
        cell8.south = cell13
        cell8.east = cell9
        cell8.west = cell7

        cell9.north = cell4
        #cell9.south = cell14
        cell9.west = cell8

        cell10.north = cell5
        cell10.south = cell15
        cell10.east = cell11

        cell11.north = cell6
        #cell11.south = cell16
        #cell11.east = cell12
        cell11.west = cell10

        #cell12.north = cell7
        #cell12.south = cell17
        cell12.east = cell13
        #cell12.west = cell11

        cell13.north = cell8
        cell13.south = cell18
        cell13.east = cell14
        cell13.west = cell12

        #cell14.north = cell9
        cell14.south = cell19
        cell14.west = cell13

        cell15.north = cell10
        cell15.south = cell20
        cell15.east = cell16

        #cell16.north = cell11
        cell16.south = cell21
        #cell16.east = cell17
        cell16.west = cell15

        #cell17.north = cell12
        cell17.south = cell22
        cell17.east = cell18
        #cell17.west = cell16

        cell18.north = cell13
        cell18.south = cell23
        #cell18.east = cell19
        cell18.west = cell17

        cell19.north = cell14
        cell19.south = cell24
        #cell19.west = cell18

        cell20.north = cell15
        #cell20.east = cell21

        cell21.north = cell16
        cell21.east = cell22
        #cell21.west = cell20

        cell22.north = cell17
        #cell22.east = cell23
        cell22.west = cell21

        cell23.north = cell18
        cell23.east = cell24
        #cell23.west = cell22

        cell24.north = cell19
        cell24.west = cell23

        self.cells.append(cell0)
        self.cells.append(cell1)
        self.cells.append(cell2)
        self.cells.append(cell3)
        self.cells.append(cell4)
        self.cells.append(cell5)
        self.cells.append(cell6)
        self.cells.append(cell7)
        self.cells.append(cell8)
        self.cells.append(cell9)
        self.cells.append(cell10)
        self.cells.append(cell11)
        self.cells.append(cell12)
        self.cells.append(cell13)
        self.cells.append(cell14)
        self.cells.append(cell15)
        self.cells.append(cell16)
        self.cells.append(cell17)
        self.cells.append(cell18)
        self.cells.append(cell19)
        self.cells.append(cell20)
        self.cells.append(cell21)
        self.cells.append(cell22)
        self.cells.append(cell23)
        self.cells.append(cell24)

        self.startCell = cell0
        cell0.makeStart()
        self.finishCell = cell24
        cell24.makeFinish()

    def defineCellNum(self):
        self.runningumber += 1
        self.cellNumbers.add(self.runningumber)
        return self.runningumber

    def buildMazeCell(self, cell, directions):
        north = None
        south = None
        east = None
        west = None
        for dir in directions:
            cellNum = self.defineCellNum()
            match dir:
                case Directions.NORTH:
                    north = Cell(cellNum)
                    cell.connectCells(north, Directions.NORTH)
                case Directions.SOUTH:
                    south = Cell(cellNum)
                    cell.connectCells(south, Directions.SOUTH)
                case Directions.EAST:
                    east = Cell(cellNum)
                    cell.connectCells(east, Directions.EAST)
                case Directions.WEST:
                    west = Cell(cellNum)
                    cell.connectCells(west, Directions.WEST)
        cell.setDirections(north, south, east, west)

    def enterMaze(self):
        return self.startCell

    def getFitness(self, currentCell):
        curCoords = currentCell.getCoords(self.nrows, self.ncols)
        endCoords = self.finishCell.getCoords(self.nrows, self.ncols)
        return DistanceForm(curCoords, endCoords)

    def testValidPath(self, path):
        cell = self.startCell
        for i in path:
            dir = Directions(i)
            match i:
                case Directions.NORTH:
                    if(cell.hasNorth()):
                        cell = cell.getNorth()
                    else:
                        return False
                case Directions.SOUTH:
                    if(cell.hasSouth()):
                        cell = cell.getSouth()
                    else:
                        return False
                case Directions.EAST:
                    if(cell.hasEast()):
                        cell = cell.getEast()
                    else:
                        return False
                case Directions.WEST:
                    if(cell.hasWest()):
                        cell = cell.getWest()
                    else:
                        return False
        if(cell.getCellNum() == self.finishCell.getCellNum()):
            return True
        else:
            return False
```

[87]:
```
class Player:
    def __init__(self, startCell = Cell(isStart = True)):
        self.current_cell = startCell
        self.fitness = 0
        self.path = []
        self.maze = None

    def enterMaze(self, maze):
        self.current_cell = maze.getStartCell()
        self.maze = maze

    def goNorth(self):
        if(self.current_cell.getNorth() is not None):
            self.current_cell = self.current_cell.getNorth()
        else:
            print("Cannot go North")

    def goSouth(self):
        if(self.current_cell.getSouth() is not None):
            self.current_cell = self.current_cell.getSouth()
        else:
            print("Cannot go South")

    def goEast(self):
        if(self.current_cell.getEast() is not None):
            self.current_cell = self.current_cell.getEast()
        else:
            print("Cannot go East")

    def goWest(self):
        if(self.current_cell.getWest() is not None):
            self.current_cell = self.current_cell.getWest()
        else:
            print("Cannot go West")

    def move(self, direction):
        match direction:
            case Directions.NORTH:
                self.goNorth()
            case Directions.SOUTH:
                self.goSouth()
            case Directions.EAST:
                self.goEast()
            case Directions.WEST:
                self.goWest()
            case _:
                raise ValueError

    def checkFinish(self):
        if self.current_cell.finishStatus():
            print("Reached Finish")
            return True
        else:
            return False

    def checkStart(self):
        if self.current_cell.startStatus():
            print("You're at the start")
            return True
        else:
            return False

    def hasNorth(self):
        return self.current_cell.hasNorth()

    def hasSouth(self):
        return self.current_cell.hasSouth()

    def hasEast(self):
        return self.current_cell.hasEast()

    def hasWest(self):
        return self.current_cell.hasWest()

    def lookAround(self):
        msg = "You can go "
        if(self.hasNorth()):
            msg += "North "
        if(self.hasSouth()):
            msg += "South "
        if(self.hasEast()):
            msg += "East "
        if(self.hasWest()):
            msg += "West"
        print(msg)

    def getCurrentCell(self):
        return self.current_cell

    def setFitness(self, fitness):
        self.fitness = fitness

    def getFitness(self):
        return self.fitness

    def setPath(self, path):
        self.path = path.copy()

    def getPath(self):
        return self.path

    def runPath(self, maze):
        for i in range(len(self.path)):
            dir = Directions(self.path[i])
            if(i > 0):
                prev = Directions(self.path[i-1])
                if(self.current_cell.atDeadEnd(prev)):
                    self.fitness = .00000000001
                    return .00000000001
            if(not(self.current_cell.hasDir(dir))):
                self.fitness = .00000000001
                return .00000000001
            self.move(dir)
            if(self.checkFinish()):
                self.fitness = 1
                self.path = self.path[:i+1]
                current_cellnum = self.current_cell.getCellNum()
                return 1
        perf = maze.getFitness(self.current_cell)
        if(perf == 0):
            self.fitness = 1
            return 1
        self.fitness = 1/perf
        return 1/perf
```

[111]:
```
# Agent
def generatePlayers(k, _maze):
    players = []
    for i in range(k):
        newPlayer = Player()
        newPlayer.enterMaze(maze=_maze)
        players.append(newPlayer)
    return players


def generateStarts(_players, lengthOfPath):
    k = len(_players)
    for org in range(k):
        path = []
        for i in range(lengthOfPath):
            path.append(random.randint(0,3))
        _players[org].setPath(path)
```

```python
def runGeneration(_players, maze):
    fitnesses = []
    for player in _players:
        player.enterMaze(maze)
        player.runPath(maze)
        fitnesses.append(player.getFitness())
        if(fitnesses[-1] == 1):
            break
    string = ""
    string += str(fitnesses[0])
    return fitnesses

def selectParents(_players, fitns):
    fitnesses = fitns.copy()
    indices = [i for i in range(len(_players))]
    ip1 = random.choices(indices, weights=fitnesses)[0]
    ip2 = random.choices(indices, weights=fitnesses)[0]
    while(ip1 == ip2):
        index = indices.index(ip2)
        del indices[index]
        del fitnesses[index]
        ip2 = random.choices(indices, weights=fitnesses)[0]
    p1 = _players[ip1]
    p2 = _players[ip2]

    assert(len(p1.getPath()) == len(p2.getPath()))
    slicePoint = random.randint(0, len(p1.getPath()) - 1)

    childPath1 = p1.getPath()[:slicePoint] + p2.getPath()[slicePoint:]
    childPath2 = p2.getPath()[:slicePoint] + p1.getPath()[slicePoint:]

    return (childPath1, childPath2)

def mutate(numMutations, player):
    muts = set()
    while(len(muts) < numMutations):
        muts.add(random.randint(0, len(player.getPath()) - 1))
    for mut in muts:
        player.getPath()[mut] = random.randint(0, 3)

def generateChildren(_players, fitnesses, numberOfMutations):
    k = len(_players)
    children = []
    while(len(children) < k):
        children_i = selectParents(_players, fitnesses)
        children.append(children_i[0])
        children.append(children_i[1])
    for i in range(len(_players)):
        player = _players[i]
        child = children[i]
        player.setPath(child)
    for chld in _players:
        mutate(numberOfMutations, chld)

def geneticAlgorithm(maze, numPlayers, numberOfMutations, lengthOfPath):
    players = generatePlayers(numPlayers, maze)
    generateStarts(players, lengthOfPath)
    i = 0
    while(True):
    #for i in range(numGenerations):
        fitnesses = runGeneration(players, maze)
        if(1 in fitnesses):
            ind = fitnesses.index(1)
            solnPath = players[ind].getPath()
            if(len(solnPath) == 1):
                print("stop here")
            print("Found solution with path: " + pathStr(solnPath) + "\nAfter "␣
 ↪+ str(i) + " generations")
            break
        generateChildren(players, fitnesses, numberOfMutations)
        i += 1


def Solve3x3Maze():
    maze = Maze(3, 3)
    maze.build3x3Maze()
    numberOfStates = 100
    numberOfMutations = 5
    lengthOfPath = 25
    geneticAlgorithm(maze, numberOfStates, numberOfMutations, lengthOfPath)

def Solve5x5Maze():
    maze = Maze(5, 5)
    maze.build5x5Maze()
    numberOfStates = 100
    numberOfMutations = 50
    lengthOfPath = 100
    geneticAlgorithm(maze, numberOfStates, numberOfMutations, lengthOfPath)

def Solve10x10Maze():
    maze = Maze(10, 10)
    maze.build10x10Maze()
    numberOfStates = 1000
    numberOfMutations = 100
    lengthOfPath = 1000
    geneticAlgorithm(maze, numberOfStates, numberOfMutations, lengthOfPath)
```

[114]:
```
Solve3x3Maze()
```

```
Reached Finish
Found solution with path: E S E S
After 2 generations
```

[110]:
```
Solve5x5Maze()
```

```
Reached Finish
Found solution with path: E E S E S S S E
After 32 generations
```

[ ]:
```
Solve10x10Maze() # takes too long to run
```

Daniel Paliulis
Professor Joseph Johnson
CSE 4705
9 December 2022

Implementing the Genetic Algorithm to Create a Maze Solving AI

Problem Description

The goal of this project is to develop an algorithm that uses the principles of artificial intelligence to find a path to the goal cell in a maze. In this problem, each maze is composed of a collection of cells, with a distinct "start cell" and distinct "goal cell". Each cell contains a unique cell number, and a reference to other maze cells in the north, south, east and west directions. These references serve as the connections between maze cells. Note that for every connection from one cell to an adjacent cell, there is also a connection from the adjacent cell back to the original cell in the opposite direction. So, for instance, if cell 11 has a connection to cell 12 in the east direction, cell 12 also has a connection to cell 11 in the west direction. This allows the agent free movement between connected cells. It is important to point out, however, that not all cells have connections in all directions. Cells will point to "None" in a particular direction, indicating that there is no connection to another cell in that direction. Similarly, if a cell does not have a connection to an adjacent cell in one direction, the adjacent cell also does not have a connection in the opposite direction. So, if cell 11 points to "None" in the west direction, then cell 10 also points to "None" in the east direction.

This problem also defines a Player object, who can be given a path to attempt to solve the maze with. A path is a list of integers [0, 3], where 0 corresponds to north, 1 corresponds to south, 2 corresponds to east, and 3 corresponds to west. The player is able to enter a maze and iterate through this path, attempting to use each integer in the path to pass through connections between cells.

Background Analysis

A genetic algorithm has been implemented for this problem to inform the agents on how to achieve a solution path through the environment. The environment in this problem is classified as partially observable, deterministic, sequential, static, continuous, and single agent. The environment is partially observable because, at each step in a single path, the agent is only aware of the immediate directions and whether there are cell connections in those directions. The agent is provided a fitness evaluation after attempting a path, however this does not directly inform the agent on the overall structure of the maze. The environment is deterministic because there is no aspect of randomness or uncertainty affecting the outcome of a path. The maze's structure remains constant for the duration of the program. Additionally, each decision of an agent to go in a particular direction results in going in that direction. There is no case where an agent will attempt to go in one direction and they will not go in that direction. Agent's may hit dead ends or attempt to go in a direction that does not exist, but these are results of the outcomes of their decisions. The environment is sequential because each generation of paths produced by the agent is informed by the results of the previous path. The environment is static because the agent can spend as much time as it needs solving the path without their performance being affected. The performance measure in this problem is binary, success in solving the maze or failure in not solving the maze. The environment is continuous because for any maze, given that an agent can cross through the same connection an infinite number of times, there is technically an infinite set of paths that the agent may take. Finally, the environment is a single agent environment because there is only one agent running the genetic algorithm and attempting paths to find a solution.

The agents in this solution are utility based. Since there is a partially observable environment where the goal node is not known to the agent, a utility has to be provided to the agent to inform its decisions on building the next path. This utility comes in the form of the fitness function. Note that this agent is able to determine what effect their actions will have, given that their actions are represented as paths taken, with the results of those actions being following a path in the maze or hitting a dead end. The agent's sensors receive information on what fitness a path had, and whether a path resulted in a solution. The agent's actuators use players to run a path and achieve an end cell.

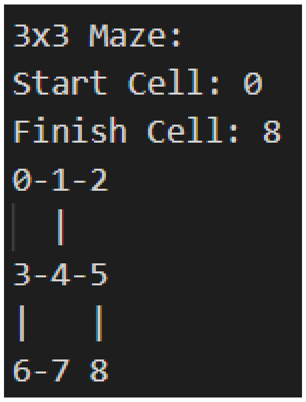
Algorithm

The agent in this problem implements a genetic algorithm. The genetic algorithm is a local search algorithm which employs an objective function value to inform states. The genetic algorithm was suited for this problem because it traditionally takes a linear collection of values as states, and the path of the maze can be represented as a linear collection of directions. The process of this algorithm begins with k randomly generated states. Using the objective function, the fitness of each state is then evaluated. The algorithm then proceeds to iteratively select two random states from our set of states to produce the next k states. The weight of a state to be selected is influenced by its evaluated fitness, where states with higher fitness functions are preferred over those with lower fitness functions. Each pair of states selected, considered the parent states, are partitioned at a random index. Complementary partitions from opposite states are then combined to create the subsequent "child" states from its parents. For instance, if state P_1 is "01231", and state P_2 is "2132", and a partition occurs at index 2, then the child state $P_{1,2}$ would be "012" + "32" = "01232", and child state $P_{2,1}$ would be "213" + "21" = "21321". Finally, mutations are applied to the child states, where single values at random locations are set to a different random value. This process is then repeated on the child states for subsequent generations until a goal state is achieved.

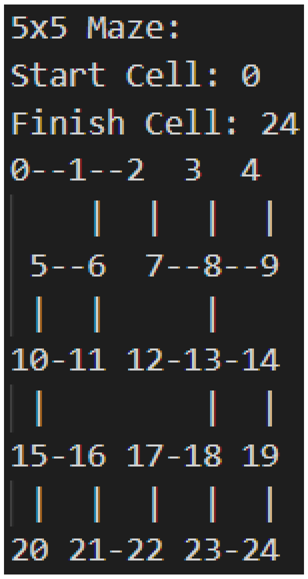
The agent is generated by specifying how many players it will maintain for each generation, how many mutations will be applied at each generation, and the length of the path for each player to generate before trying to walk through a maze. A list of directions, defined as integers, is assigned to a single player to represent a single state in a single generation. For instance, a possible path for a player may be [1, 3, 2, 0, 1, 1, 3], which corresponds to [South, West, East, North, South, South, West]. To determine a path's fitness function, cells are treated like coordinates with a unique x and y position on a grid. The distance formula, $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, is then used to find the distance from the path's final cell to the goal cell in the maze. Using this distance, the fitness is evaluated as 1/(distance to goal cell). However, there are special conditions which override this fitness value. One such condition is if the path leads to a dead end, then its fitness is set to .00000000001. This way, any path that leads to a dead end has a low fitness and is less likely to produce a child for the next generation. If a path manages to reach the goal cell at any point in its execution, then that path receives a fitness of 1 and is displayed as a solution, ending the program. At each generation, if the goal state is not reached, then random.choices() from python's "random" library is used to continuously choose pairs of parent paths, with the fitness evaluation corresponding to the weight of each path to be chosen. Pairs of parent paths are then partitioned to and combined to create child paths for the next generation. The code is implemented to also allow for a custom number of mutations for any path as it produces a child for the next generation. In this case, for an input of n mutations, the code will choose a random index in the path to change to a random direction [0, 3] n number of times. Adding mutations and partitioning is done to increase the diversity of each subsequent generation. This allows for exploration of unique paths that don't converge to a single local maxima, which may maximize the fitness evaluation but not reach the goal cell. The output of this program shows the number of generations it took to achieve the solution path, and the solution path taken from the start node to the goal node.

The Mazes

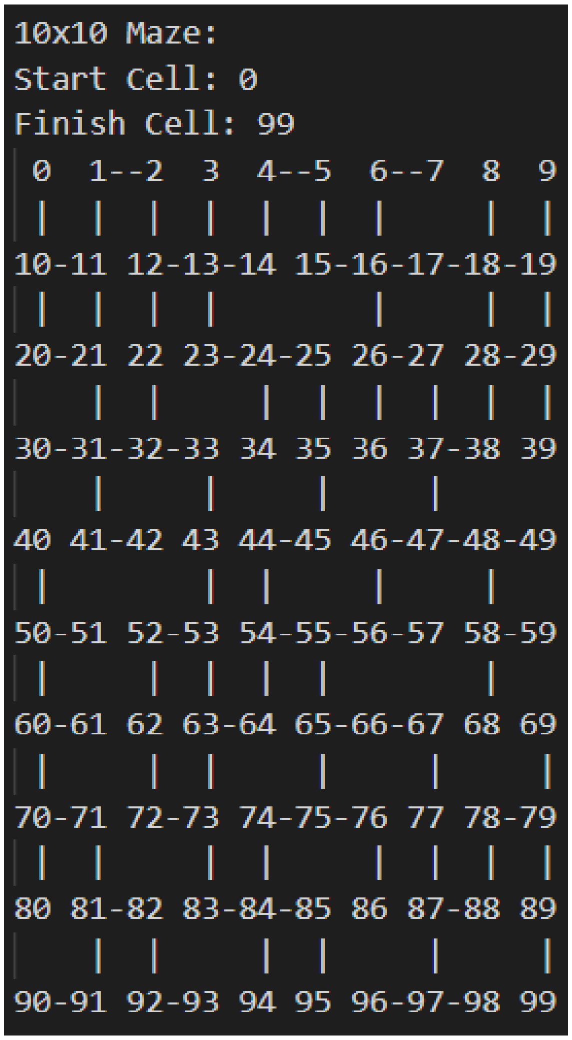
The following are visual representations of the three mazes designed to test the AI. An integer represents the cell (defined by its cell number) and a '-' or a '|' represents a connection between cells:



```
3x3 Maze:
Start Cell: 0
Finish Cell: 8
0-1-2
  |
3-4-5
|   |
6-7 8
```



```
5x5 Maze:
Start Cell: 0
Finish Cell: 24
0--1--2  3  4
    |  |  |  |
 5--6  7--8--9
 |  |     |
10-11 12-13-14
 |        |  |
15-16 17-18 19
 |  |  |  |  |
20 21-22 23-24
```



```
10x10 Maze:
Start Cell: 0
Finish Cell: 99
 0  1--2  3  4--5  6--7  8  9
 |  |  |  |  |  |  |     |  |
10-11 12-13-14 15-16-17-18-19
 |  |  |  |        |     |  |
20-21 22 23-24-25 26-27 28-29
    |  |     |  |  |  |  |  |
30-31-32-33 34 35 36 37-38 39
    |     |     |     |
40 41-42 43 44-45 46-47-48-49
 |        |  |     |     |
50-51 52-53 54-55-56-57 58-59
 |     |  |  |  |        |
60-61 62 63-64 65-66-67 68 69
 |     |  |     |     |     |
70-71 72-73 74-75-76 77 78-79
 |  |     |  |     |  |  |  |
80 81-82 83-84-85 86 87-88 89
    |  |     |  |     |     |
90-91 92-93 94 95 96-97-98 99
```

Data

Number of Generations Required to Solve 3 Types of Mazes Using the Genetic Algorithm

Input				Output					
Maze	Agent			Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average Number of Generations
	Number of States per Generation	Number of Mutations per Generation	Length of Path per State						
3x3 Maze	100	5	25	2	0	1	2	10	3
5x5 Maze	100	50	100	108	135	23	86	32	76.8
10x10 Maze	1000	100	1000	Inconclusive					

Result Analysis

As is shown in the table above, the simple 3 by 3 maze (9 cells) finds a solution fairly efficiently, with an average number of generations of 3. Any number of generations above 3 when running this algorithm tends to be an outlying result, with most solutions being found within 3 generations.

Increasing the maze size to 5 by 5 (25 cells) significantly increases the required number of generations to find a solution. As can be seen in the data, the average number of generations increases to 76.8. It is also important to note that in other runs of this test, the number of generations fluctuated significantly, with some runs taking as few as 15 generations, and others taking as many as 250. It is possible that the recipe of parameters used to define the agents for this maze could be better optimized for faster problem solving.

Increasing the maze size to 10 by 10 (100 cells) takes longer than 90 minutes to run, and so a result from that agent has never been observed. It is assumed that, given the nature of the algorithm, there will eventually be a particular partition paired with a particular number of mutations that will allow for a solution to be produced. However, such a solution may take far longer than reasonable time to appear, and therefore no results can be concluded for gathering data on the data of the number of generations it takes for the algorithm to solve a 10 by 10 maze.

Potential Improvements

This genetic algorithm may be improved to result in more efficient problem solving. One potential improvement may be to take an iterative deepening approach. That is, starting at a path with a length of two, increase the path length by a degree of one for every generation. For instance, at generation 0, every state consists of two directions taken in the maze. Then, after partitioning and mutating, the path length increases to a length of three for generation 2, where one random direction is appended to the end at each path before testing the generation. This way, time is not wasted in earlier generations trying nonsensical paths before any form of selection has occurred. This may provide a significant improvement given that many of the early paths lead to dead ends. Such a conclusion is made given that testing fixed length paths from the start of the program, with no selection informing the agent of a better path, relies only on randomness to reach a solution path.

One other improvement may be to implement the evaluation function to not promote paths which perform backtracking. That is, the algorithm would not favor paths where the agent passes through the same connections more than once. This would be a potential improvement because there is nothing stopping the algorithm from continuously passing across the same connections multiple times, passing this behavior onto subsequent generations. This may especially become an issue if there are cycles in the maze. Implementing the fitness function to prefer paths which do not cross the same locations more than once would reduce the set of possible paths taken, and thus result in finding a solution faster.

One final improvement would be to make the fitness function favor paths that have similarity to a possible solution path, rather than paths that result in cells that are locationally closer to the goal cell. At the moment, the fitness function acts as more of a heuristic function, where being closer to the goal state is preferred for subsequent generations. The reason this may be an issue is that a solution may require the agent to go in a path that temporarily moves away from the goal state, before eventually moving towards the goal state. This may be especially prevalent given that mazes, in having directions cut at certain cells, do not guarantee locational proximity to the goal cell as a correct path. An improvement on this fitness function would be for the problem statement to privately define all of the possible solution paths for the maze, and compare directions in the agent's paths to look for similarities with the solution paths. Paths with more similar directions at particular locations will have a higher fitness and thus be more preferred for subsequent generations in the maze.