Joseph Yeager
CS 441
11/25/2015

Block Dude Solver

Background:

Block Dude is a puzzle game in which the player utilizes moveable blocks to create staircases to get past obstacles. The end goal is for the player to reach the door. Valid moves are east, west, northwest/northeast to ascend, and falling. Sometimes the game provides only the minimal amount of blocks needed to complete the puzzle, sometimes requiring reuse of blocks. A perfectly legal move can change the solvability of the puzzle. The original goal of this project was to make a program able to solve any level in the game, but the goal was later reduced to solving the first couple levels of the game.

Description:

To solve the problem at hand I used a combination of constraint satisfaction and heuristic search techniques. Python was selected as the implementation language because it had a couple of constraint satisfaction libraries that looked promising, but I ultimately ended up using a homebrewed implementation, which I will discuss later. Another reason for Python's selection was my familiarity with the language, which would prevent me having to wrestle with the language.

Before any problem solving could begin, I had to determine what I thought would be the best representation for the puzzles. In Python, arrays are implemented as lists, so performing some array operations that would be trivial in other languages can be substantially slower in Python. This fact helped me to rule out multidimensional arrays and decide to use a linear integer array as my representation, as they would offer enough performance for what I needed, and quite a bit more than their multidimensional counterparts. Each possible type of object in the game environment was given a unique integer representation; these included: blocks, bricks, empty spaces, a west-facing player, a east-facing player, and the door(the goal). Levels were stored as CSV files with the width and the height as the first two members of the file. I would then load the level into an array and pop the first two data members off, which would give me access to the dimensions making the linearization trivial.

With the level representation out of the way, I was able to move onto the constraint satisfaction. The constraint portion of the project was specialized enough that none of the libraries available seemed to be a great fit, and some of them were downright confusing to use because of the frustratingly low level of documentation. The problem was simple enough that a solution could be implemented without the assistance of outside libraries. The constraints in this problem consist of the legal moves in the game. Legal moves are dictated by the level

environment directly surrounding the player. So the only portion of the environment of interest, as far as constraints are concerned, is a 3x3 matrix of the environment with the player in the middle. All of the legal moves at a given time can be determined by examining the four quadrants of this matrix. A dictionary of 2x2 matrices was used to represent the legal moves of the game, and then each quadrant of the 3x3 matrix was checked against this dictionary to determine what moves were available. If a move was available, the key would be stored as one of the available moves.

Example:

Suppose this is the current 3x3 environment matrix.

EMPY	EMPY	EMPY
BLCK	WEST	EMPY
BRCK	BRCK	BRCK

The the following red sections are the quadrants to be scanned:

EMPY	EMPY	EMPY
BLCK	WEST	EMPY
BRCK	BRCK	BRCK

EMPY	EMPY	EMPY
BLCK	WEST	EMPY
BRCK	BRCK	BRCK

EMPY	EMPY	EMPY
BLCK	WEST	EMPY
BRCK	BRCK	BRCK

EMPY	EMPY	EMPY
BLCK	WEST	EMPY
BRCK	BRCK	BRCK

Each of these sections returns at most one valid move which is able to be performed, and which is added to a movelist to be fed to the heuristic.

The heuristic search portion was without a doubt the largest part of the assignment. Quite a few utility functions were required to be able to solve this problem. The first problem encountered was the need to be able to tell if there is an obstacle between the player and the goal state. Since the player is only able to move east, west, and climb stairs; an obstacle is anything which is two or more blocks higher than what is in the spaces to the east and west of it. The simplest obstacle in the game is a two high stack of bricks with nothing to the east or west of it, but they can also include pits and drop offs. The end result was a function which scans

between the player's location and the goal location to determine if there are any obstacles in between. This function starts at the space adjacent to the player, in the direction of the goal. It recurses through the level and will descend into any pits it encounters and climb any obstacles it finds to determine its height.

Another necessary piece of information is where the blocks need to be place to pass an obstacle and how many of them are required. Since the obstacles are overcome by building stairs, the number of blocks can be expressed by the following summation: (where h is height)

$$\sum_{i=1}^{h-1} i$$

But this equation doesn't take into account that some of the spaces might already be occupied by blocks or bricks. To solve this, a function which iterates over the spaces adjacent to the obstacle in a staircase pattern was implemented. It checks if a space is empty of already occupied by a block or brick, it stores the location where the block is needed and increments the quantity required.

With some of the basic functionality in place it was time to begin searching. I noticed in my constraint satisfaction portion there were at most three possible moves at any given state, so a ternary tree was the obvious choice. Each node would store the sequence of moves made to arrive at its current state, the updated level, and some other miscellaneous player state items. The original implementation used a traditional representation of a ternary tree. Some of the algorithms became non-trivial to implement and understand with this design. The next attempt was to use the array representation of an n-ary tree. This made the search algorithms much easier to implement, but both the time and space efficiency were abysmal. This was due to the tree is very sparse at times due to pruning, but it was still storing 3^n nodes, where n is the number of moves. This meant it had to store and iterate over a very large array which had large empty sections. With this implementation the search was taking 16 seconds to complete some of the more difficult test sets. The third time was the charm, I arrived at a data structure which offered significant time and space efficiency increases. I used an ordered dictionary and used the array indexing system to generate the dictionary keys. This allowed easy access to any relatives while only having to store relevant nodes. Since the dictionary was ordered, it was also indexable by the order of the dictionary, allowing for it to be easily iterated. With these optimizations, the search times went from about 16 secs to tens of milliseconds on the more difficult test sets.

The heuristic was pretty simple and for the most part it consisted of eliminating cycles. Any move which immediately undoes the progress of the previous move or move(s) would be pruned from the tree. For example, picking up a block and then immediately dropping it would cause a prune. To prevent these , 2,3, and 4 move sequences with the new potential move included would be examined. If backward progress or oscillating occurred, the branch would be pruned. Here are some example movesets which cause a prune:

["west", "west", "face east", "east"]
["east", "east", "face west", "west"],

["face west", "face east"], ["face east", "face west"],
["drop", "pick up"], ["pick up", "drop"]

Some other heuristics I use consist of:

- Don't drop the block unless it helps to solve an obstacle
- Check periodically for obstacles, if there are none, move in the direction of the goal.
- If a move results in victory, always choose it.

Using these methods I was able to complete the first two levels of the game, see unanticipated obstacles for an explanation.

Where's The AI:

This project uses two different artificial intelligence techniques. The first is constraint satisfaction to define and restrict the moves to only legal moves allowed by the game. The second is the heuristic used the search to prevent solution from having backward progress and keep it from exploring solutions containing long oscillations. These both greatly increase the speed to finding a solution.

Code Statistics:

The total code base is 433 lines of code excluding comments and whitespace. This was initially much higher, but after a few good refactors I was able to greatly reduce the complexity of the program and the LOC count. All of the code has been debugged for the current levels, but there probably are some edge cases exist on untested levels of the game.

Unanticipated Obstacles:

- I had to start from scratch when I coded myself into a corner. I built something which wasn't a heuristic search and more of me telling it what to do. This was fine on the simpler levels, but as the levels became more complex so did my code.
- Data structure problems. The tree had horrible time and space efficiency, so I sunk a lot of time into optimizing it.
- Minor details here and there. Small things building up.

What I learned:

I now have a better understanding of constraint satisfaction and can see it is very useful when you have a problem needing to abide by a certain set of rules. It is a very simple concept to implement and it is very powerful. While researching CSPs for this project I saw examples of constraint satisfaction be used to solve sudoku, n-queens, and Einstein's 5 house riddle. With the use of some of these libraries, these problems are able to be solved in very few lines of code and seem to be fairly efficient.

I also have a much better understanding of how to go about implementing a breadth-first heuristic search. I learned pruning is a very simple yet very powerful concept. It can prevent you from performing an enormous number of unnecessary operations. Every time I would add a blacklisted move sequence which resulted in more pruning, the time for my search to complete would drop significantly. I also learned over pruning is a very easy mistake to make. I had a test set which kept stalling, and I later realized it was because I had a blacklisted move sequence that seemed nonsensical, but was required for completion in this case. This project helped to prove to me why these searches are necessary and just how immense of a difference they can make.

This project also reinforced the importance of using the correct data structure. If I would have stuck with my array implementation for a ternary tree, it would have been a list with 3^n items where n=24. Which means it would have had 282,429,536,481 items in it even with pruning. My optimized data structure has 165 nodes in it for the same level.

Build Instructions:

Install the graphics libraries:
sudo apt-get install python-tk
sudo apt-get install python-imaging-tk

Run the program:

Test sets:
python solver.py test

Game:
python solver.py game

Both:
python solver.py test game

Both with pauses in between levels:
python solver.py test game pause

The program requires python 2.7 to run due to dependencies, so if any of the above commands fail try replacing python with python2.7

Source Code:

Student Written	Borrowed

```
import sys
from Tkinter import Tk, Frame, Canvas
```

```
from PIL import ImageTk
import csv
import time
import math
from collections import OrderedDict
EMPY, BRCK, BLCK, WEST, EAST, DOOR = 0,1,2,3,4,5
width, height = 0,0

########################################################################
#####################       Data Types       ###########################
########################################################################
class Coordinate:
    def __init__(s, x, y):
        s.x, s.y = x, y

class Level:
    def __init__(s, width,height,layout):
        s.width, s.height, s.layout = width, height, layout

    def copy(s, level):
        s.width = level.width
        s.height = level.height
        s.layout = list(level.layout)

class Node:
    def __init__(s):
        s.move = None
        s.level = Level(0,0,0)
        s.player = Player()
        s.moveList, s.children, s.blockGoals = [],[],[]

#  The app class is resonsible for loading and parsing the CSV files that
#  contain the levels.  It is also responsible for displaying the GUI and
#  updating it when a change has been made.
class App:
    def __init__(s, root):
        s.root = root
        s.root.title("Block Dude Solver")
        s.frame = Frame(s.root)
        s.frame.pack()
        s.canvas = Canvas(s.frame, bg="white", width=(width*24)+100,
height=(height*24)+100)
        s.canvas.pack()
        s.levels, s.imageArray = [],[]
        filenames = ["brick.png","block.png","dudeLeft.png","dudeRight.png","door.png"]

        s.imageArray.append("")
        for i in range(0,len(filenames)):
            s.imageArray.append(ImageTk.PhotoImage(file="./assets/" + filenames[i]))

    def displayLevel(s,level):
        s.canvas.delete("all")
        s.updateCanvasDems(level.width,level.height)
        length = len(level.layout)
        row = 0
        for i in range(0,length):
            if i % (level.width) == 0:
                row += 1
            if level.layout[i] != EMPY:
```

```
                x = ((i%(level.width))*24) + 65
                y = (row*24) + 40
                s.canvas.create_image(x,y, image=s.imageArray[level.layout[i]])

    def loadLevels(s,path, fileArray):
        length = len(fileArray)
        for i in range(0,length):
            with open(path + fileArray[i], 'rb') as f:
                reader = csv.reader(f)
                reader = list(reader).pop(0)
                level = map(int,reader)
                width = level.pop(0)
                height = level.pop(0)
                newLevel = Level(width,height,level)
                s.levels.append(newLevel)

    def updateCanvasDems(s,width, height):
        newWidth = (width*24)+100
        newHeight = (height*24)+100
        s.canvas.config(width=newWidth,height=newHeight)

    def run(s):
        s.root.mainloop()


# The player class is responsible for saving the state of a player, this
# includes: the players position or index, the direction the player is facing,
# if the player is falling, etc. This class is also responsible for performing a players
moves.
class Player:
    def __init__(s):
        s.pos = Coordinate(0,0)
        s.dir, s.index,s.index2 = 0, 0, 0
        s.isHoldingBlock, s.falling = False, False

    def copy(s, player):
        s.setPos(player.pos.x, player.pos.y)
        s.dir = player.dir
        s.index = player.index
        s.isHoldingBlock = player.isHoldingBlock
        s.falling = player.falling

    def setPos(s,x,y):
        s.pos.x, s.pos.y = x, y

    def setDirection(s, playerValue):
        s.dir = playerValue

    def moveEast(s):
        s.index += 1

    def moveWest(s):
        s.index -= 1

    def moveNEast(s, width):
        s.moveEast()
        s.index -= (width)

    def moveNWest(s,width):
```

```
        s.moveWest()
        s.index -= (width)

    def fall(s, width):
        s.index += (width)

    def pickupBlock(s):
        s.isHoldingBlock = True
        return s.getAdj()

    def dropBlock(s):
        s.isHoldingBlock = False
        return s.getAdj()

    def setIndex(s, index):
        s.index = index

    def getAdj(s):
        if s.dir == WEST:
            return s.index - 1
        else:
            return s.index + 1

# The solver class contains all of the logic that is needed to solve
# the first two levels of Block Dude as well as all the test sets.
class Solver:
    def __init__(s):

        # Set up the search tree root node
        s.dt = OrderedDict()
        s.dt[0] = Node()
        s.root = s.dt[0]
        s.root.moveList = []
        s.root.children = s.getChildren(0)

        s.obstacleFlag, s.victory = False, False
        s.spacesBelow, s.blockGoals, s.moveQuadrants, s.moveList, s.quadMoves =
[],[],[],[],[]
        s.obstacles = {}

        # validMoves contains all of the legal moves that can be made
        # The moves are expressed as 2x2 matrices that have been linearized
        s.validMoves = {
            "e":  [[0,0,4,0]],
            "w":  [[0,0,0,3]],
            "fe": [[0,0,3,0],[0,0,3,1],[0,0,3,2],[0,1,3,1],[0,2,3,2],[0,2,3,1],[0,1,3,2]],
            "fw": [[0,0,0,4],[0,0,1,4],[0,0,2,4],[1,0,1,4],[2,0,2,4],[2,0,1,4],[2,0,1,4]],
            "nw": [[0,0,1,3],[0,0,2,3]],
            "ne": [[0,0,4,1],[0,0,4,2]],
            "pu": [[0,0,4,2],[0,0,2,3]],
            "dr": [[0,0,4,0], [0,0,0,3],[0,0,4,1]],
            "fa":
[[3,1,0,1],[3,1,0,0],[3,0,0,0],[3,2,0,1],[3,2,0,0],[3,1,0,2],[3,2,0,2],[3,0,0,1],[3,0,0,2]
,

[1,4,1,0],[1,4,0,0],[0,4,0,0],[2,4,1,0],[2,4,0,0],[1,4,2,0],[2,4,2,0],[0,4,1,0],[0,4,2,0]]
        }

        # Victory moves contains all of the legal moves that result in victory.
```

```
        s.victoryMoves = {
            "e":    [[0,0,4,5]],
            "w":    [[0,0,5,3]],
            "nw":   [[5,0,1,3],[5,0,2,3]],
            "ne":   [[0,5,4,1],[0,5,4,2]],
            "fa":
[[3,1,5,1],[3,1,5,0],[3,0,5,0],[3,2,5,1],[3,2,5,0],[3,1,5,2],[3,2,5,2],[3,0,5,1],[3,0,5,2]
,

[1,4,1,5],[1,4,0,5],[0,4,0,5],[2,4,1,5],[2,4,0,5],[1,4,2,5],[2,4,2,5],[0,4,1,5],[0,4,2,5]]
        }

        # List of moves that result in cycles or backwards progress
        s.cyclicalMoves = [
            ["w", "fa", "fe", "ne"], ["e", "fa", "fw", "nw"],
            ["fw", "w", "fe", "e"],  ["fe", "e", "fw", "w"],
            ["w", "w", "w", "fe"],   ["e", "e", "e", "fw"],
            ['pu', 'e', 'fw', 'dr'], ['pu', 'w', 'fe', 'dr'],

            ["w", "w", "fe"],  ["e", "e", "fw"],
            ["w", "fe", "e"],  ["e", "fw", "w"],
            ["ne", "fw", "w"], ["nw", "fe", "e"],
            ["w", "w", "fe"],  ["e", "e", "fw"],

            ["fw", "fe"], ["fe", "fw"],
            ["dr", "pu"], ["pu", "dr"],
            ["w", "fe"],  ["e", "fw"]
        ]

    # Takes a level as an argument and sets it as the current working level in the solver
    def setLevel(s,level):
        s.root.level = Level(level.width,level.height,list(level.layout) )
        s.level = Level(level.width,level.height,list(level.layout) )
        s.currentLevel = s.root.level
        s.length = len(s.level.layout)
        s.root.player = Player()

    # Locates the player and the door in the level
    # Sets the x,y coords of the player and the goal
    def locateStartAndGoalState(s):
        s.goalPos = Coordinate(-1,-1)
        s.root.player.setPos(-1,-1)
        for i in range(0,s.length):
            if s.level.layout[i] == DOOR:
                x, y = i % s.level.width, (i - (i%s.level.width))/s.level.width
                s.goalPos = Coordinate(x,y)
            elif s.level.layout[i] == WEST or s.level.layout[i] == EAST:
                x, y = i % s.level.width, (i - (i%s.level.width))/s.level.width
                s.root.player.setPos(x,y)
                s.root.player.setIndex(i)
                s.root.player.setDirection(s.level.layout[i])

    # Computes the taxi cab distance between the player and the door
    # Sets s.modifier to -1 if the value is negative and 1 otherwise
    # The modifer is used quite often in other computations
    def taxiCabDistance(s, player):
        s.taxiCab = Coordinate(s.goalPos.x - player.pos.x, s.goalPos.y - player.pos.y)
        s.modifier = s.taxiCab.x / abs(s.taxiCab.x)
```

```
    # Uses the list of obstacles computed by checkObstacles to determine
    # where blocks need to be placed to complete an obstacle.  It will check
    # to both the east and the west of the obstacle
    def findBlockGoals(s):
        s.blockGoals = []
        numOfObs = len(s.obstacles)
        nonEmpty = [BRCK, BLCK]
        for index,height in s.obstacles.iteritems():
            reset = 0
            west = index - 1
            east = index + 1
            tempIR = index + (s.level.width)
            tempIL = tempIR
            for i in range (1, height):
                for j in range (1, i+1):
                    reset += s.modifier
                    tempIR -= s.modifier
                    tempIL += s.modifier
                    # Check to the east and the west,  if the block adjacent in either
direction is nonEmpty
                    # don't add  blocks in that direction of the obstacle to the
blockGoals list
                    if s.level.layout[east] not in nonEmpty and s.level.layout[tempIR] not
in nonEmpty:
                        s.blockGoals.append(tempIR)
                    if s.level.layout[west] not in nonEmpty and s.level.layout[tempIL] not
in nonEmpty:
                        s.blockGoals.append(tempIL)
                tempIR += s.level.width + reset
                tempIL += s.level.width - reset

    # Checks for obstacles only between the player and the goal
    # If obstacles are found, call findBlockGoals
    def checkObstaclesFindBlocks(s, index):
        cur = s.dt[index]
        s.taxiCabDistance(s.dt[index].player)
        s.obstacleFlag = False
        s.checkObstaclesHelper(0,0,0, cur.player.index + s.modifier, cur.level,
s.modifier)
        if s.obstacleFlag:
            s.findBlockGoals()

    # checkFor obstacles in both directions.  This will clear the obstacle flag
    # if nothing is found.
    def checkObstacles(s, index):
        cur = s.dt[index]
        s.obstacles = {}
        s.obstacleFlag = False
        mod = 1
        s.checkObstaclesHelper(0,0,0, cur.player.index + mod, cur.level, mod)
        s.checkObstaclesHelper(0,0,0, cur.player.index - mod, cur.level, -mod)

    # Recursive helper function for finding obstacles.  It will scan along the ground and
descend into pits
    # until an obstacle is found.  When an obstacle is found it will check its height, if
its height is more that
    # 1 higher than obstacle adjacent to it, it will set the obstacle flag, and store the
index of the highest point
    # of the obstacle as well as it's height.  If a pit of depth 2 or greater is found, it
```

```
will store the index of
    # the dropoff point.
    def checkObstaclesHelper(s, prevHeight, height, depth, index, level, modifier):
        spaceAbove = level.layout[index - level.width]
        if (height + 1) - prevHeight > 1 and (spaceAbove == EMPY or spaceAbove == DOOR):
            s.obstacleFlag = True
            s.obstacles[index] = height + 1
        if depth >= 2:
            s.obstacleFlag = True
            newIndex = index - ((depth -1) * level.width) - modifier
            if newIndex not in s.obstacles:
                s.obstacles[newIndex] = depth
        if index % level.width ==  0 or index < 0 or index > s.length or
level.layout[index] == DOOR:
            return

        elif level.layout[index] == EMPY:
            spaceBelow = level.layout[index + level.width]
            if spaceBelow == BLCK or spaceBelow == BRCK: # If space below is brick/block
go forward
                newIndex = index + modifier
                s.checkObstaclesHelper(0, 0, 0, newIndex, level, modifier)
            elif spaceBelow == EMPY: # If space below is empty go down
                newIndex = index  +  s.level.width
                s.checkObstaclesHelper(prevHeight, height, depth+1, newIndex, level,
modifier)
            elif spaceBelow == DOOR:
                return

        else:
            spaceAbove = level.layout[index - level.width]
            if spaceAbove == BRCK: # move up when the block above is a block
                newIndex = index -  level.width
                s.checkObstaclesHelper(prevHeight, height+1, 0, newIndex, level, modifier)
            elif spaceAbove == BLCK  or spaceAbove == EMPY: # move forward when the block
above is empty
                newIndex = index  +  modifier
                if level.layout[newIndex] == EMPY:
                    depth += 1
                s.checkObstaclesHelper(0, 0, depth, newIndex, level, modifier)
            elif spaceAbove == DOOR:
                return

    # Check if any of the current moves are a victory move.  If one of them is,
    # add it to the move list and set the victory flag.
    def checkVictory(s, move, moveList):
        for k,v in s.victoryMoves.iteritems():
            if move in v:
                moveList.append(k)
                s.moveList = list(moveList)
                s.victory = True
                break

    # Takes a linear version of the 3x3 matrix surrounding the player and chops it
    # up into quadrants that can be scanned for moves.
    def generateMoveQuads(s, i, l, w, level, moveList):
        pg = [ l[i-w-1], l[i-w], l[i-w+1], l[i-1], l[i], l[i+1], l[i+w-1], l[i+w],l[i+w+1]
]
        s.moveQuadrants = []
```

```
        for i in range(0,5):
            if i != 2:
                s.moveQuadrants.append([ pg[i], pg[i+1], pg[i+3], pg[i+4] ])

    # analyze the move quadrants generated, and check them for legal moves.
    # Add legal moves into list
    def analyzeMoveQuads(s, move, index):
        cur = s.dt[index]
        for k,v in s.validMoves.iteritems():
            if move in v:
                if k == "pu" and not cur.player.isHoldingBlock:
                    s.addMove(k)
                elif k == "dr" and cur.player.isHoldingBlock:
                    s.addMove(k)
                elif k != "dr" and k != "pu":
                    s.addMove(k)

    # Add a move to the currently available moves list
    def addMove(s, move):
        if move not in s.quadMoves:
            s.quadMoves.append(move)

    # Recusively scans down until a non empty space is found
    def checkDown(s,index, level):
        if level.layout[index] != EMPY:
            index -= level.width
            return index
        return s.checkDown(index + level.width, level)

    # Have the player perform the move, and update the level
    def performMove(s,level,data, player, move):
        oldIndex = player.index

        if move == "fa":
            player.fall(s.level.width)
        elif move == "w":
            player.moveWest()
        elif move == "e":
            player.moveEast()
        elif move == "nw":
            player.moveNWest(s.level.width)
        elif move == "ne":
            player.moveNEast(s.level.width)
        elif move == "fw":
            player.setDirection(WEST)
        elif move == "fe":
            player.setDirection(EAST)
        elif move == "pu":
            playerAdj = player.pickupBlock()
            level.layout[playerAdj] = EMPY
        elif move == "dr":
            playerAdj = player.dropBlock()
            playerAdj = s.checkDown(playerAdj,level)
            level.layout[playerAdj] = BLCK

        level.layout[oldIndex] = 0
        level.layout[player.index] = player.dir
```

These blocks were borrowed from

https://en.wikipedia.org/wiki/K-ary_tree#Properties_of_k-ary_trees

```
    # Get the parent index of the current working index.
    # Uses the same indexing scheme as n-ary array representation of trees
    def getParentIndex(s):
        s.par =  int ( math.floor( (s.i - 1) / 3 ) )
        if s.par < 0:
            s.par = 0

    # Takes an index and uses it to generate the index of the nth child
    # Uses the same indexing scheme as n-ary array representation of trees
    def getNthChild(s,index, nth):
        return 3 * index + 1 + nth

    # Returns a list of the children indices of index
    def getChildren(s,index):
        return [ s.getNthChild(index,0), s.getNthChild(index, 1), s.getNthChild(index, 2)]

    # Pops a child index of the parents list of children
    # Creates a new node and deep copies the parents data
    # Generates the list of children for the new node
    # Takes the move that caused the branch and adds it to the new
    # nodes move list, then performs using the new nodes player and level
    # adds the node to the tree
    def addToTree(s, tree, parent, move):
        s.i = s.dt[s.par].children.pop(0)
        newChild = Node()
        newChild.move = move
        newChild.moveList = list(parent.moveList)
        newChild.moveList.append(move)
        newChild.player.copy(parent.player)
        newChild.level.copy(parent.level)
        newChild.children = s.getChildren(s.i)
        newChild.blockGoals = list(s.blockGoals)
        s.performMove(newChild.level, newChild, newChild.player, move)
        tree[s.i] = newChild

    # Syntactic sugar used for prioritizing moves
    def prioritizeMoves(s, move, moves):
        if move in moves:
            s.addToTree(s.dt, s.dt[s.par], move)

    # Checks the move list in combination with the new
    # potential move to see if it would generate a cyclical
    # move and sets a flag if it does.
    def checkCycles(s, move, moveList):
        l = 3
        s.isNotACycle = True
        moveSeq = moveList[-l:]
        moveSeq.append(move)
        for i in range(0, l):
            if moveSeq[i:] in s.cyclicalMoves:
                s.isNotACycle = False
                break

    # Generates the spaces that are adjacent to the player
    # given the current direction they are facing.  If there
    # is a dropoff in front of the player, it decends the dropoff
    # and adds all of those spaces aswell.
```

```
    def generateAdjacent(s, level):
        s.playerAdj = []
        tempI = s.dt[s.par].player.getAdj()
        s.playerAdj.append(tempI)
        tempI += level.width
        while tempI < s.length:
            if level.layout[tempI] == EMPY:
                s.playerAdj.append(tempI)
            tempI += level.width

    # Checks if falling is a available move at the time, and
    # if it is, falling is chosen due to the physics of the game.
    # Other wise it checks to see if there are still obstacles
    # and then loops through the list of available moves and
    def pickMoves(s):
        cur = s.dt[s.par]
        if "fa" in s.quadMoves:  # If fall is a choice, it is the only choice.
            s.addToTree(s.dt, s.dt[s.par], "fa")
            s.counter += 1
            return

        if s.obstacleFlag: # If check to see if obstacles have been cleared
            s.checkObstacles(s.par)
        for move in s.quadMoves:
            s.checkCycles(move, s.dt[s.par].moveList)
            if not s.obstacleFlag:  # If there are no obstacles, run for the goal
                if s.modifier < 0:
                    s.prioritizeMoves(move,["w","nw","fw"])
                else:
                    s.prioritizeMoves(move,["e","ne","fe"])
            elif s.isNotACycle:  # Only pick moves that will not generate cycles
                if move == "dr":
                    s.generateAdjacent(cur.level)
                    #  Only drop a block if it is in one of the block goals.
                    if any(i in s.playerAdj for i in s.blockGoals):
                        s.addToTree(s.dt, cur, move)
                else:
                    s.addToTree(s.dt, cur, move)
        s.counter += 1

    def solve(s):
        s.locateStartAndGoalState()
        s.i = 1
        s.checkObstaclesFindBlocks(0)

        bottom, s.counter = 0, 0
        startTime = time.clock()
        while not s.victory:
            lenDt = len(s.dt)
            if bottom == lenDt:
                print "\nFailed to Solve"
                return
            for i in range(bottom, lenDt):
                s.par = s.dt.keys()[s.counter]
                cur = s.dt[s.par]
                s.generateMoveQuads(cur.player.index, cur.level.layout,
s.root.level.width, cur.level, cur.moveList)
                for move in s.moveQuadrants:
                    s.checkVictory(move, cur.moveList)
```

```
                    if s.victory:
                        break
                    s.analyzeMoveQuads(move,s.par)
                s.pickMoves()
                s.quadMoves = []
                bottom = s.counter

        print "\nTime taken(secs): ", time.clock() - startTime
        moveListWorstCase = (3**len(s.moveList))
        treeReduction = (moveListWorstCase - len(s.dt))
        print "Solved!!! Uses", len(s.moveList), "moves and a tree with", len(s.dt),
"nodes, a reduction of %.5g nodes" %treeReduction

    def stepThroughSolution(s):
        currentMove = s.moveList.pop(0)
        s.performMove(s.root.level, s,s.root.player, currentMove)

########################################################################
####################         Program Loop        ######################
########################################################################
if __name__=='__main__':
    root = Tk()
    app = App(root)
    setPause = False
    testFiles = ["level1.csv",
"level2.csv","level3.csv","level4.csv","level5.csv","level6.csv","level7.csv"]
    gameFiles = ["level1.csv","level2.csv"]

    if len(sys.argv) >= 2:
        if "test" in sys.argv:
            print "Loading test sets..."
            app.loadLevels("./testLevels/", testFiles)
        if "game" in sys.argv:
            print "Loading game levels..."
            app.loadLevels("./gameLevels/", gameFiles)
        if "pause" in sys.argv:
            setPause = True
    else:
        print "Please provide an arugment:\n\ttest: runs the test sets\n\tgame: runs the
levels"
        sys.exit(0)

    def startFunction():
        for i in range(0, len(app.levels)):
            solver = Solver()
            solver.setLevel(app.levels[i])
            app.displayLevel(app.levels[i])
            root.update()
            solver.solve()

            if setPause:
                raw_input("Press Enter to view solution...")
            while(len(solver.moveList) > 0):
                root.update()
                solver.stepThroughSolution()
                app.displayLevel(solver.currentLevel)
                time.sleep(0.13)
            if setPause:
                if i != len(app.levels) -1:
```

```
                    raw_input("Press Enter to begin solving next level")
                else:
                    raw_input("Done!  Press enter to exit.")

        sys.exit(0)

    root.after(500, startFunction)
    app.run()
```