Artificial Intelligence

*Code*

This simple document contains the PDF transcript of the code along with the related documentation how to use the program.

Davide Maurilio Morello

Artificial Intelligence

**The repository for this code, can be found at:** [**https://github.com/Eloh666/AICoursework**](https://github.com/Eloh666/AICoursework)

# Parser

**import** numpy  
**import** networkx **as** nx  
**import** math  
**import** ntpath  
  
**def euclideanDistance**(a, b):  
 x = pow(a[0] - b[0], 2)  
 y = pow(a[1] - b[1], 2)  
 **return** math.sqrt(x+y)  
  
**class CavernsNetwork**:  
  
 **def** \_\_init\_\_(self, fileName):  
 self.fileName = ntpath.basename(fileName)  
  
 # Parses basic file into a graph  
 self.coordinates = []  
 self.\_\_cavernData = self.\_\_readFile(fileName).split(',')  
 self.\_\_numberCoordinats = int(self.\_\_cavernData[0], 10)  
  
 # Selects source and destination  
 self.source = 0  
 self.target = self.\_\_numberCoordinats - 1  
  
 # Generates advanced structures from the data  
 self.graph = self.\_\_parseGraph()  
 self.matrix = self.\_\_parseMatrix()  
 self.\_\_computeEdges()  
  
 **def \_\_readFile**(self, fileName):  
 **with** open(fileName, 'r') **as** cavernsFile:  
 **return** cavernsFile.read().replace('\n', '')  
  
 **def \_\_parseGraph**(self):  
 coordsData = self.\_\_cavernData[1: (2 \* self.\_\_numberCoordinats) + 1]  
 parsedCoordsData = list(map(**lambda** x: int(x, 10), coordsData))  
 self.coordinates = list(zip(parsedCoordsData, parsedCoordsData[1:]))[::2]  
  
 graph = nx.DiGraph()  
 **for** index, value **in** enumerate(self.coordinates):  
 graph.add\_node(index, coords=tuple(numpy.squeeze(numpy.asarray(value))))  
 **return** graph  
  
 **def \_\_computeEdges**(self):  
 **for** (x, y), value **in** numpy.ndenumerate(self.matrix):  
 **if** int(value) == 1:  
 source = self.coordinates[x]  
 dest = self.coordinates[y]  
 self.graph.add\_edge(  
 y,  
 x,  
 weight=round(euclideanDistance(source, dest), 2)  
 )  
  
 **def \_\_parseMatrix**(self):  
 matrixData = self.\_\_cavernData[(2 \* self.\_\_numberCoordinats) + 1:]  
 **return** numpy.matrix(matrixData).reshape(self.\_\_numberCoordinats, self.\_\_numberCoordinats)

# Renderer

**from** matplotlib.backends.backend\_qt5agg **import** FigureCanvasQTAgg **as** FigureCanvas  
**from** matplotlib.figure **import** Figure  
**from** PyQt5.QtWidgets **import** QSizePolicy  
**import** networkx **as** nx  
**import** math  
  
  
**class MatPlotLibRenderer**(FigureCanvas):  
  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(Figure(tight\_layout=**True**))  
 self.axes = self.figure.add\_subplot(111)  
  
 FigureCanvas.setSizePolicy(self,  
 QSizePolicy.Expanding,  
 QSizePolicy.Expanding)  
 FigureCanvas.updateGeometry(self)  
 self.pos = **None** self.name = "Cavern"  
  
 **def clearFigure**(self, hard=**False**):  
 **if** hard:  
 self.pos = **None** self.axes.relim()  
 self.axes.autoscale\_view(**True**, **True**, **True**)  
 self.draw\_idle()  
  
 **def plotGraph**(self,  
 network,  
 finalPath=[],  
 primaryVisited=[],  
 secondaryVisisted=[],  
 noPath=**False**,  
 final=**False**):  
  
 graph = network.graph  
 source = network.source  
 target = network.target  
  
 nodesNum = len(graph.nodes())  
 **if not** self.pos:  
 **if** nodesNum < 10:  
 self.pos = nx.fruchterman\_reingold\_layout(graph, scale=100, k=2 / math.sqrt(nodesNum))  
 **else**:  
 self.pos = nx.circular\_layout(graph, scale=100)  
  
 self.axes.clear()  
 self.axes.relim()  
 self.draw\_idle()  
  
 # sets how to mark edges  
 source = [source]  
 target = [target]  
  
 regularNodes = [node **for** node **in** graph.nodes() **if** node **not in** source + target]  
  
 **if** final:  
 redEdges = [(i, j) **for** i, j **in** zip(finalPath[:len(finalPath)], finalPath[1:])]  
 print(redEdges)  
 finalPath = [node **for** node **in** finalPath **if** node **not in** source + target]  
 regularNodes = [node **for** node **in** regularNodes **if** node **not in** finalPath]  
 **else**:  
 redEdges = list(  
 filter(**lambda** x:  
 x[0] **in** primaryVisited **and** x[1] **in** secondaryVisisted  
 **or** x[1] **in** primaryVisited **and** x[0] **in** secondaryVisisted, graph.edges())  
 )  
 regularNodes = [node **for** node **in** regularNodes **if** node **not in** primaryVisited + secondaryVisisted]  
 source = [node **for** node **in** source **if** node **not in** primaryVisited + secondaryVisisted]  
 target = [node **for** node **in** target **if** node **not in** primaryVisited + secondaryVisisted]  
  
 regularEdges = [node **for** node **in** graph.edges() **if** node **not in** redEdges]  
  
 # renders source, destination and general nodes  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=source,  
 ax=self.axes,  
 node\_color='green',  
 alpha=0.5,  
 node\_size=20000/nodesNum,  
 label='Source Node'  
 )  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=target,  
 ax=self.axes,  
 node\_color='orange',  
 alpha=0.5,  
 node\_size=20000/nodesNum,  
 label='Destination Node'  
 )  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=regularNodes,  
 ax=self.axes,  
 node\_color='grey' **if** noPath **else** 'blue',  
 alpha= 0.25,  
 node\_size=20000/nodesNum,  
 label='Others'  
 )  
 # renders the final path  
 **if** final:  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=finalPath,  
 ax=self.axes,  
 node\_color='red',  
 alpha=0.5,  
 node\_size=20000 / nodesNum,  
 label='Selected Path' **if** final **else None** )  
 # otherwise renders the nodes currently being visited  
 **else**:  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=secondaryVisisted,  
 ax=self.axes,  
 node\_color='yellow',  
 alpha=0.5 **if not** final **else** 0.1,  
 node\_size=20000 / nodesNum,  
 label='Secondary Visisted'  
 )  
 nx.draw\_networkx\_nodes(graph,  
 self.pos,  
 nodelist=primaryVisited,  
 ax=self.axes,  
 node\_color='purple',  
 alpha=0.5 **if not** final **else** 0.1,  
 node\_size=20000 / nodesNum,  
 label='Primary Visited'  
 )  
  
 # Renderes red edges  
 nx.draw\_networkx\_edges(graph,  
 self.pos,  
 edgelist=regularEdges,  
 ax=self.axes,  
 arrows=**True**,  
 alpha=0.25,  
 edge\_color='red' **if** noPath **else** 'black'  
 )  
  
 # Renders black edges  
 nx.draw\_networkx\_edges(graph,  
 self.pos,  
 edgelist=redEdges,  
 ax=self.axes,  
 arrows=**True**,  
 alpha=0.5,  
 edge\_color='red'  
 )  
 nx.draw\_networkx\_labels(graph, self.pos, ax=self.axes, labels=nx.get\_node\_attributes(graph, 'coords'), font\_size=16)  
 nx.draw\_networkx\_edge\_labels(graph, self.pos, ax=self.axes, alpha=0.1)  
 self.axes.legend(markerscale=1/5, title='Node types')

# App/Main

**import** sys  
**import** matplotlib  
matplotlib.use("Qt5Agg")  
**from** Parser **import** Parser  
**from** Rendering.Renderer **import** MatPlotLibRenderer  
**from** Algorithms.Dijkstra **import** Dijkstra  
**from** matplotlib.backends.backend\_qt5agg **import** NavigationToolbar2QT **as** NavigationToolbar  
**from** PyQt5 **import** uic, QtCore, QtGui  
**from** PyQt5.QtWidgets **import** QApplication, QMainWindow, QVBoxLayout, QFileDialog  
**from** Utils.UndoRedoStack **import** UndoRedoStack  
**import** appGui  
  
  
**class AICourseWork**(QMainWindow, appGui.Ui\_Coursework):  
 **def** \_\_init\_\_(self):  
 super(AICourseWork, self).\_\_init\_\_()  
 self.setupUi(self)  
 self.setAttribute(QtCore.Qt.WA\_DeleteOnClose)  
  
 # Setup general state parameters  
 self.stack = UndoRedoStack()  
 self.currentNetwork = **None** self.algorithmWrapper = **None** # Setup rendering canvas  
 self.layout = QVBoxLayout(self.graphWidget)  
 self.renderingCanvas = MatPlotLibRenderer()  
 self.canvarToolbar = NavigationToolbar(self.renderingCanvas, parent=**None**)  
 self.layout.addWidget(self.canvarToolbar)  
 self.layout.addWidget(self.renderingCanvas)  
  
 # Bindings  
 # open file  
 self.actionOpen.triggered.connect(self.openFile)  
  
 # bind buttons graph  
 self.solveButton.clicked.connect(self.findShortestPath)  
 self.stepButton.clicked.connect(self.stepThrough)  
 self.undoButton.clicked.connect(self.undo)  
 self.redoButton.clicked.connect(self.redo)  
 self.resetButton.clicked.connect(self.reset)  
  
 # buttons disabled until a graph is loaded  
 self.undoButton.setEnabled(**False**)  
 self.redoButton.setEnabled(**False**)  
 self.resetButton.setEnabled(**False**)  
 self.solveButton.setEnabled(**False**)  
 self.stepButton.setEnabled(**False**)  
  
 **def undo**(self):  
 \_, past, present = self.stack.undo()  
 self.undoButton.setEnabled(past)  
 self.redoButton.setEnabled(present)  
  
 **def redo**(self):  
 \_, past, present = self.stack.redo()  
 self.undoButton.setEnabled(past)  
 self.redoButton.setEnabled(present)  
  
 **def handleOutput**(self, text):  
 self.activityLog.moveCursor(QtGui.QTextCursor.End)  
 self.activityLog.ensureCursorVisible()  
 self.activityLog.insertPlainText('\n' + text)  
  
 **def openFile**(self):  
 dialog = QFileDialog()  
 file, \_ = dialog.getOpenFileName(self, directory='./TestFiles', filter='\*.cav')  
 **if** file:  
 self.stack.clearStack()  
 self.currentNetwork = Parser.CavernsNetwork(file)  
 self.renderingCanvas.clearFigure(hard=**True**)  
 self.activityLog.clear()  
 self.algorithmWrapper = Dijkstra(  
 network=self.currentNetwork,  
 log=self.handleOutput,  
 renderer=self.renderingCanvas  
 )  
 self.stack.onChange(self.algorithmWrapper)  
 self.stepButton.setEnabled(**True**)  
 self.resetButton.setEnabled(**True**)  
 self.solveButton.setEnabled(**True**)  
 self.distanceLabel.setText('Total Distance ')  
  
 **def stepThrough**(self):  
 distance = self.algorithmWrapper.bidirectionalDijkstra(stepping=**True**)  
 self.stepButton.setEnabled(**not** self.algorithmWrapper.finished)  
 \_, past, present = self.stack.onChange(self.algorithmWrapper)  
 self.undoButton.setEnabled(past)  
 self.redoButton.setEnabled(present)  
 **if** distance:  
 self.distanceLabel.setText('Total Distance ' + str(distance))  
  
 **def reset**(self):  
 self.stack.clearStack()  
 self.renderingCanvas.clearFigure()  
 self.activityLog.clear()  
 self.stack.clearStack()  
 self.algorithmWrapper.restart()  
 self.stack.onChange(self.algorithmWrapper)  
 self.undoButton.setEnabled(**False**)  
 self.redoButton.setEnabled(**False**)  
 self.resetButton.setEnabled(**True**)  
 self.solveButton.setEnabled(**True**)  
 self.stepButton.setEnabled(**True**)  
  
 # solve with selected algorithm  
 **def findShortestPath**(self):  
 self.redoButton.setEnabled(**False**)  
 self.undoButton.setEnabled(**False**)  
 self.stepButton.setEnabled(**False**)  
 self.activityLog.clear()  
 self.activityLog.insertPlainText('Processing ' + self.currentNetwork.fileName + '\n')  
 distance = self.algorithmWrapper.bidirectionalDijkstra()  
 self.distanceLabel.setText('Total Distance ' + str(distance))  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication(sys.argv)  
 window = AICourseWork()  
 window.show()  
 sys.exit(app.exec\_())

## Algorithm Wrapper

**from** heapq **import** heappush  
**from** itertools **import** count  
  
**class AlgorithmWrapper**:  
 **def** \_\_init\_\_(self, network, log, weight='weight', renderer=**None**):  
 self.renderer = renderer  
 self.weight = weight  
 self.log = log  
 self.network = network  
 self.graph = network.graph  
 self.source = network.source  
 self.target = network.target  
 self.coordinates = network.coordinates  
 self.\_\_setupInitialParams()  
  
 self.dirEnum = {  
 0: 'forward',  
 1: 'backward',  
 }  
  
 **def \_\_setupInitialParams**(self):  
 self.dists = {  
 'forward': {},  
 'backward': {},  
 }  
  
 # stores the available paths  
 self.paths = {  
 'forward': {self.source: [self.source]},  
 'backward': {self.target: [self.target]},  
 }  
  
 # heap of (distance, node) tuples for extracting next node to expand  
 self.fringe = {  
 'forward': [],  
 'backward': [],  
 }  
  
 # nodes who have already been seen and investigated  
 self.visited = {  
 'forward': {self.source: 0},  
 'backward': {self.target: 0},  
 }  
  
 # neighs for extracting correct neighbor information  
 # the method unwraps the successors to a node  
 self.neighs = {  
 'forward': self.graph.successors\_iter,  
 'backward': self.graph.predecessors\_iter,  
 }  
  
 self.c = count()  
  
 # setup the self.fringe heap  
 heappush(self.fringe['forward'], (0, next(self.c), self.source))  
 heappush(self.fringe['backward'], (0, next(self.c), self.target))  
  
 # variables to hold shortest discovered path  
 self.finalPath = []  
 self.directionNum = 1  
 self.finalDist = 0  
  
 self.primaryVisited = []  
 self.secondaryVisited = []  
 self.noPath = **False** self.finished = **False  
  
 def restart**(self):  
 self.\_\_setupInitialParams()  
  
 **def reRender**(self):  
 self.renderer.plotGraph(  
 network=self.network,  
 primaryVisited=self.primaryVisited,  
 secondaryVisisted=self.secondaryVisited,  
 noPath=self.noPath,  
 finalPath=self.finalPath,  
 final=self.finished  
 )

# Dijkstra

**from** heapq **import** heappush, heappop  
**from** Algorithms.AlgorithmWrapper **import** AlgorithmWrapper  
  
  
**class Dijkstra**(AlgorithmWrapper):  
  
 **def bidirectionalDijkstra**(self, stepping=**False**):  
  
 **if** self.finished:  
 self.log('\n\nComputation already complete')  
 **return  
  
 if** self.source == self.target:  
 self.log('Matching Source and Destination')  
 self.log('The distance is zero.')  
 **return** 0, [self.source]  
  
 **while** self.fringe['forward'] **and** self.fringe['backward']:  
 # choose direction  
 # direction == 0 is forward direction and direction == 1 is back  
 self.directionNum = 1 - self.directionNum  
  
 direction = self.dirEnum[self.directionNum]  
 self.log('\n\n--- Visiting direction set to ' + direction)  
  
 # extract closest to expand  
 (dist, \_, v) = heappop(self.fringe[direction])  
 self.log('\nExpanding main node ===> ' + str(self.coordinates[v]))  
 **if** v **in** self.dists[direction]:  
 # Shortest path to v has already been found  
 self.log('Shortest path already been found ')  
 **continue** # update distance  
 self.dists[direction][v] = dist  
 **if** v **in** self.dists[self.dirEnum[1 - self.directionNum]]:  
 self.log('Node ' + str(self.coordinates[v]) + ' expanded bidirectionally')  
 # the shorted path is found  
 self.log('\n\nBidirectional scan complete.')  
 self.log('\Shortest path found ' + str(self.finalPath))  
 self.log('Global distance ' + str(self.finalDist))  
 self.finished = **True  
 if not** stepping:  
 self.reRender()  
 **return** self.finalDist  
  
 self.log('Checking node ' + str(self.coordinates[v]) + ' neighbors')  
 **if** stepping:  
 self.primaryVisited = [v]  
 self.secondaryVisited = list(self.neighs[direction](v))  
 self.path = []  
 **for** w **in** self.neighs[direction](v):  
 self.log('\nExpanding node ---> ' + str(self.coordinates[w]))  
 **if** direction == 'forward':  
 minWeight = self.graph[v][w].get(self.weight, 1)  
 **else**: # back, must remember to change v,w->w,v  
 minWeight = self.graph[w][v].get(self.weight, 1)  
 vwLength = self.dists[direction][v] + minWeight # self.graph[w][v].get(weight,1)  
 self.log('Calculating distance ' + str(vwLength))  
  
 # catches the exception caused by negative paths in dijkstras  
 **if** w **in** self.dists[direction]:  
 **if** vwLength < self.dists[direction][w]:  
 self.log('\n\n\nWarning: Dijkstra does NOT support negative cycles. Please check the input.')  
 **return** -1, []  
  
 **elif** w **not in** self.visited[direction] **or** vwLength < self.visited[direction][w]:  
 **if** direction == 'forward':  
 self.log('Adding new distance from source: ' + str(vwLength))  
 **else**:  
 self.log('Adding new distance from target: ' + str(vwLength))  
 self.log('Mapping updated')  
 # relaxing  
 self.visited[direction][w] = vwLength  
 heappush(self.fringe[direction], (vwLength, next(self.c), w))  
 self.paths[direction][w] = self.paths[direction][v] + [w]  
 **if** w **in** self.visited['forward'] **and** w **in** self.visited['backward']:  
 totalDistance = self.visited['forward'][w] + self.visited['backward'][w]  
 self.log('Source --> Destination path found')  
 self.log('The distance is: ' + str(totalDistance))  
 **if** self.finalPath:  
 self.log('Comparing new path to previous ones')  
 self.log('Older option: ' + str(self.finalDist))  
 **if** self.finalPath == [] **or** self.finalDist > totalDistance:  
 **if** self.finalDist > totalDistance:  
 self.log('The new path is better, selecting')  
 **else**:  
 self.log('Selecting the path as current best')  
 self.finalDist = totalDistance  
 reversePath = self.paths['backward'][w]  
 reversePath.reverse()  
 self.finalPath = self.paths['forward'][w] + reversePath[1:]  
 self.log('New temporary best path: ')  
 self.log(str(self.finalPath))  
 **else**:  
 self.log('The previous options are better')  
 self.log('Discarding')  
  
 **if** stepping:  
 self.log('\n')  
 **return None** self.log("No path between %s and %s." % (self.source, self.target))  
 self.finished = **True** self.noPath = **True  
 return** -1, []

# Undo/Redo Stack

**import** copy  
  
**class UndoRedoStack**:  
 **def** \_\_init\_\_(self):  
 self.past = []  
 self.future = []  
 self.present = **None  
  
 def undo**(self):  
 **if not** self.past **or not** self.present:  
 **return  
 if** self.present:  
 self.future.append(self.present)  
 self.present = self.past.pop()  
 self.reRender()  
 **return** self.present, self.past != [], self.future != []  
  
 **def redo**(self):  
 **if not** self.future **or not** self.present:  
 **return  
 if** self.present:  
 self.past.append(self.present)  
 self.present = self.future.pop()  
 self.reRender()  
 **return** self.present, self.past != [], self.future != []  
  
 **def onChange**(self, state):  
 **if** self.present:  
 self.past.append(self.present)  
 self.present = copy.copy(state)  
 self.reRender()  
 **return** self.present, self.past != [], self.future != []  
  
 **def clearStack**(self):  
 self.past = []  
 self.future = []  
 self.present = **None  
  
 def reRender**(self):  
 **if** self.present.reRender:  
 self.present.reRender()