main.py

```
import os
from FileInput import FileInput
from GraphMaker import GraphMaker
from Renderer import Renderer
# run program
# print("Hello world.")
this_dir = os.path.dirname(os.path.realpath(__file__))
FileInput.do_file_input(this_dir, 4)
GraphMaker.build_nodes()
GraphMaker.build_connections()
# for n in GraphMaker.nodes:
#     print(str(n))
# n = Node()
# n.set_position((1, 2))
# print(str(n))
# n2 = Node()
# n2.set_position((4, 3))
# print(str(n2))
# Solver.initialise_solver(GraphMaker.nodes)
# Solver.generate_path()
#
# Solver.initialise_solver(GraphMaker.nodes)
Renderer.render_graph()
```

Renderer.py

```
from tkinter import *
import tkinter as tk
from Solver import Solver
class RenderNode:
    def __init__(self):
        self.id = -1
        self.loc = (-1, -1)
        self.oval = -1
        self.start = False
        self.goal = False
        self.selected = False
        self.visited = False
        self.examined = False
    def get_solver_node(self):
        return Solver.solver_nodes[self.id]
    def get_x(self):
        return self.loc[0]
    def get_y(self):
        return self.loc[1]
class Renderer:
    scale = 20
    oval_radius = 10
    render_nodes = []
    inspected = []
    selected = -1
    min_x = 0
    min_y = 0
    width = 0
    height = 0
    window = Tk()
    canvas = -1
    queue_frame = -1
    labels = []
    key_canvas = -1
    label_route = -1
    @staticmethod
    def reset():
        Renderer.scale = 20
        # Reset trackers
        Renderer.inspected = []
        Renderer.selected = 0
        # Reset render nodes
        for rn in Renderer.render_nodes:
            rn.selected = False
            rn.visited = False
            rn.examined = False
        Renderer.render_nodes[0].selected = True
        Renderer.render_nodes[0].visited = True
        Renderer.render_nodes[0].examined = True
        # Reset labels
        for i in range(5):
            Renderer.labels[i].config(text="")
        Renderer.key_canvas.itemconfig(Renderer.label_route, text="Selected")
        Solver.initialise_solver()
        Renderer.draw_nodes()
    @staticmethod
    def render_graph():
        if not Solver.ready:
            Solver.initialise_solver()
        min_x = 9999999
        max_x = -9999999
```

```
        min_y = 9999999
        max_y = -9999999
        for i in range(len(Solver.solver_nodes)):
            # Make the new render node
            rn = RenderNode()
            rn.id = i
            rn.loc = Solver.solver_nodes[i].get_node().get_position()
            # To calculate screen details
            if rn.loc[0] < min_x:
                min_x = rn.loc[0]
            if rn.loc[0] > max_x:
                max_x = rn.loc[0]
            if rn.loc[1] < min_y:
                min_y = rn.loc[1]
            if rn.loc[1] > max_y:
                max_y = rn.loc[1]
            Renderer.render_nodes.append(rn)
        # Save screen details
        Renderer.min_x = 1 - min_x
        Renderer.min_y = 1 - min_y
        Renderer.width = max_x - min_x + 2
        Renderer.height = max_y - min_y + 2
        if Renderer.width < 4 or Renderer.height < 4:
            Renderer.scale = 60
        # Start node
        Renderer.render_nodes[0].start = True
        Renderer.render_nodes[0].selected = True
        Renderer.render_nodes[0].visited = True
        Renderer.render_nodes[0].examined = True
        Renderer.selected = 0
        # Goal node
        Renderer.render_nodes[len(Renderer.render_nodes) - 1].goal = True
        # Window set up
        Renderer.window.title("Graph Solver")
        Renderer.window.geometry(str(Renderer.width * Renderer.scale + 320) + "x" +
                                 str(Renderer.height * Renderer.scale + 100))
        # Queue frame set up
        queue_frame = Frame(Renderer.window)
        queue_frame.grid(column=3, row=0)
        queue_header = Label(queue_frame, text="Dijkstra queue:")
        queue_header.grid(row=0, sticky=N+W)
        l1 = Label(queue_frame)
        l1.grid(row=1, sticky=W)
        Renderer.labels.append(l1)
        l2 = Label(queue_frame)
        l2.grid(row=2, sticky=W)
        Renderer.labels.append(l2)
        l3 = Label(queue_frame)
        l3.grid(row=3, sticky=W)
        Renderer.labels.append(l3)
        l4 = Label(queue_frame)
        l4.grid(row=4, sticky=W)
        Renderer.labels.append(l4)
        l5 = Label(queue_frame)
        l5.grid(row=5, sticky=W)
        Renderer.labels.append(l5)
        # Button frame set up
        bottomframe = Frame(Renderer.window)
        bottomframe.grid(column=2, row=2)
        # Button for reset
        reset_button = Button(Renderer.window, text="Reset", fg="black",
```

```
command=Renderer.reset)
        reset_button.grid(column=3, row=2, sticky=N+W)
        # Button for all
        all_button = Button(bottomframe, text="Jump to end", fg="black",
command=Renderer.do_all)
        all_button.pack(side=RIGHT, fill=X)
        # Button for stepping
        step_button = Button(bottomframe, text="Step forward", fg="black",
command=Renderer.do_step)
        step_button.pack(side=LEFT, fill=X)
        # Canvas set up
        Renderer.canvas = Canvas(Renderer.window, width=Renderer.width * Renderer.scale,
                                 height=Renderer.height * Renderer.scale)
        Renderer.canvas.grid(column=1, row=0, columnspan=2, rowspan=2)
        # Key set up
        key = Canvas(Renderer.window, width=100, height=Renderer.height *
Renderer.scale)
        key.grid(column=0, row=0, rowspan=2)
        Renderer.key_canvas = key
        key.create_text(4, 13, anchor=W, text="Key:")
        y_val = 25
        space = 5
        key.create_oval(3, y_val, 23, y_val + 20, fill="#2eb82e")
        key.create_text(28, y_val + 10, anchor=W, text="Start")
        y_val += 20 + space
        key.create_oval(3, y_val, 23, y_val + 20, fill="#cc0000")
        key.create_text(28, y_val + 10, anchor=W, text="Goal")
        y_val += 20 + space
        key.create_oval(3, y_val, 23, y_val + 20, fill="#ff8c1a")
        Renderer.label_route = key.create_text(28, y_val + 10, anchor=W,
text="Selected")
        y_val += 20 + space
        key.create_oval(3, y_val, 23, y_val + 20, fill="#0044cc")
        key.create_text(28, y_val + 10, anchor=W, text="Visited")
        y_val += 20 + space
        key.create_oval(3, y_val, 23, y_val + 20, fill="#66c2ff")
        key.create_text(28, y_val + 10, anchor=W, text="Examined")
        # Draw on canvas
        for node in Renderer.render_nodes:
            for link in node.get_solver_node().get_node().get_links():
                Renderer.draw_links(Renderer.canvas, link)
        for node in Renderer.render_nodes:
            Renderer.draw_node(Renderer.canvas, node)
        # Main loop
        Renderer.window.mainloop()
    @staticmethod
    def do_all():
        while not Solver.solved and not Solver.unsolvable:
            Renderer.do_step()
        Renderer.do_step()
    @staticmethod
    def do_step():
        Solver.generate_by_step()
        if Renderer.selected != -1:
            Renderer.render_nodes[Renderer.selected].selected = False
            Renderer.selected = -1
        if not Solver.solved:
            if not Solver.unsolvable:
                if len(Renderer.inspected) > 0:
                    sel = Renderer.inspected.pop(0)
                    Renderer.selected = sel
                    Renderer.render_nodes[sel].visited = True
```

```
                    Renderer.render_nodes[sel].selected = True
                queue = Solver.visit_queue
                new_inspects = []
                for pair in queue:
                    new_inspects.append(pair[0])
                    if pair[0] not in Renderer.inspected:
                        Renderer.render_nodes[pair[0]].examined = True
                        Renderer.inspected.append(pair[0])
                Renderer.print_labels(queue)
                Renderer.inspected = new_inspects
            else:
                Renderer.labels[0].config(text="FINISHED")
                Renderer.labels[1].config(text="Graph is")
                Renderer.labels[2].config(text=" unsolvable.")
                Renderer.labels[3].config(text="")
                Renderer.labels[4].config(text="")
        else:
            Renderer.render_nodes[len(Renderer.render_nodes) - 1].visited = True
            Renderer.render_nodes[len(Renderer.render_nodes) - 1].selected = True
            for node_id in Solver.get_path():
                Renderer.render_nodes[node_id].selected = True
            Renderer.labels[0].config(text="FINISHED")
            Renderer.labels[1].config(text="Total distance:")
            Renderer.labels[2].config(text=str(round(Solver.distance, 2)))
            Renderer.labels[3].config(text="")
            Renderer.labels[4].config(text="")
            Renderer.key_canvas.itemconfig(Renderer.label_route, text="Route")
        Renderer.draw_nodes()
    @staticmethod
    def draw_nodes():
        for node in Renderer.render_nodes:
            Renderer.draw_node(Renderer.canvas, node)
    @staticmethod
    def draw_node(canvas, render_node):
        rad = Renderer.oval_radius
        if render_node.oval == -1:
            x = (render_node.loc[0] + Renderer.min_x) * Renderer.scale
            y = (Renderer.height - (render_node.loc[1] + Renderer.min_y)) *
Renderer.scale
            render_node.oval = canvas.create_oval(x - rad, y - rad, x + rad, y + rad,
fill="#ffffff")
            canvas.create_text(x, y, text=str(render_node.id + 1))
        if render_node.goal:
            if render_node.selected:
                canvas.itemconfig(render_node.oval, fill="#ff3311")
            else:
                canvas.itemconfig(render_node.oval, fill="#cc0000")
        elif render_node.start:
            if render_node.selected:
                canvas.itemconfig(render_node.oval, fill="#66ff66")
            else:
                canvas.itemconfig(render_node.oval, fill="#2eb82e")
        elif render_node.selected:
            canvas.itemconfig(render_node.oval, fill="#ff8c1a")
        elif render_node.visited:
            canvas.itemconfig(render_node.oval, fill="#0044cc")
        elif render_node.examined:
            canvas.itemconfig(render_node.oval, fill="#66c2ff")
        else:
            canvas.itemconfig(render_node.oval, fill="#ffffff")
    @staticmethod
```

```
    def draw_links(canvas, link):
        start = list(Solver.solver_nodes[link.start].get_node().get_position())
        start[0] = (start[0] + Renderer.min_x) * Renderer.scale
        start[1] = (Renderer.height - (start[1] + Renderer.min_y)) * Renderer.scale
        end = list(Solver.solver_nodes[link.end].get_node().get_position())
        end[0] = (end[0] + Renderer.min_x) * Renderer.scale
        end[1] = (Renderer.height - (end[1] + Renderer.min_y)) * Renderer.scale
        start, end = Renderer.arrow_reduction(start, end)
        canvas.create_line(start[0], start[1], end[0], end[1], arrow=LAST)
    @staticmethod
    def print_labels(queue):
        list_length = 5
        for i in range(list_length):
            if i >= len(queue):
                Renderer.labels[i].config(text="")
            else:
                lab = "Node: " + str(queue[i][0] + 1) + ", From: " + \
                      str(Solver.solver_nodes[queue[i][0]].visited_from + 1) + \
                      ", Distance: " + str(round(queue[i][1], 2))
                Renderer.labels[i].config(text=lab)
        if len(queue) > list_length:
            Renderer.labels[list_length - 1].config(text="+" + str(len(queue) -
(list_length - 1)) + " more...")
    @staticmethod
    def arrow_reduction(start, end):
        direction = [end[0] - start[0], end[1] - start[1]]
        direction = Renderer.normalise(direction)
        rad = Renderer.oval_radius
        start = [start[0] + direction[0] * rad, start[1] + direction[1] * rad]
        end = [end[0] - direction[0] * rad, end[1] - direction[1] * rad]
        return start, end
    @staticmethod
    def normalise(vec):
        length = (vec[0] ** 2 + vec[1] ** 2)**(1/2)
        return [vec[0]/length, vec[1]/length]
```

Solver.py

```
from Node import Node
from GraphMaker import GraphMaker
class SolverNode:
    def __init__(self):
        self.visited_from = -1
        self.visited = False
        self.shortest_dist = -1
        self.id = -1
        self.goal = False
    def set_node(self, id_no):
        self.id = id_no
    def previous(self):
        return self.visited_from
    def visited(self):
        return self.visited
    def get_node(self):
        return GraphMaker.nodes[self.id]
class Solver:
    ready = False
    solver_nodes = []
    visited_count = 0
    visit_queue = []
    solved = False
    unsolvable = False
    saved_path = []
    distance = -1
    @staticmethod
    def reset_solver():
        Solver.solver_nodes.clear()
        Solver.visited_count = 0
        Solver.visit_queue = [(0, 0)]
        Solver.solved = False
        Solver.unsolvable = False
        Solver.ready = False
        Solver.saved_path.clear()
        Solver.distance = -1
    @staticmethod
    def initialise_solver():
        nodes = GraphMaker.nodes
        Solver.reset_solver()
        for i in range(len(nodes)):
            sn = SolverNode()
            sn.set_node(i)
            Solver.solver_nodes.append(sn)
        Solver.solver_nodes[0].shortest_dist = 0
        Solver.solver_nodes[len(nodes) - 1].goal = True
        Solver.ready = True
    @staticmethod
    # Dijkstra all in one
    def generate_path():
        if not Solver.ready:
            Solver.initialise_solver()
        if len(Solver.solver_nodes) == 0:
            print("No nodes. Aborting pathfinding.")
            return
        print("Search results:")
        while not Solver.solved:
            node_id = Solver.visit_queue[0][0]
            Solver.search_from_node(node_id)
            print(Solver.visit_queue)
```

```
            queue_length = len(Solver.visit_queue)
            if queue_length == 0:
                print("Didn't find a path.")
                break
        print(str(Solver.visited_count) + " visited")
        path = Solver.get_path()
        print(path)
        coords = ""
        for n in path:
            coords += str(Solver.solver_nodes[n].get_node()) + "\n"
        print(coords)
    @staticmethod
    # Step by step method for dijkstra
    def generate_by_step():
        if not Solver.ready:
            Solver.initialise_solver()
        if not Solver.solved:
            if len(Solver.visit_queue) == 0:
                Solver.unsolvable = True
                print("No possible route.")
            else:
                Solver.search_from_node(Solver.visit_queue[0][0])
        else:
            print(Solver.get_path())
    @staticmethod
    # Dijkstra method of node searching
    def search_from_node(this_id):
        Solver.visit_queue.pop(0)
        this_node = Solver.solver_nodes[this_id]
        Solver.visited_count += 1
        this_node.visited = True
        if this_node.goal:
            Solver.solved = True
            Solver.distance = this_node.shortest_dist
            return
        dests = Solver.order_closest_neighbours(this_id)
        for d in dests:
            weight = this_node.get_node().get_connection_to(d).length()
            test_val = this_node.shortest_dist + weight
            if Solver.solver_nodes[d].shortest_dist == -1 \
                    or test_val < Solver.solver_nodes[d].shortest_dist:
                if Solver.solver_nodes[d].shortest_dist != -1:
                    Solver.visit_queue.remove((d,
Solver.solver_nodes[d].shortest_dist))
                Solver.solver_nodes[d].shortest_dist = test_val
                Solver.solver_nodes[d].visited_from = this_id
                Solver.visit_queue.append((d, test_val))
                Solver.visit_queue.sort(key=Solver.sort_by_second)
                # Solver.search(d)
        Solver.solved = False
    @staticmethod
    def sort_by_second(pair):
        return pair[1]
    @staticmethod
    # Dijkstra method of neighbour sorting
    def order_closest_neighbours(root):
        cons = GraphMaker.nodes[root].get_destinations()
        for c in cons:
            if Solver.solver_nodes[c].visited:
                cons.remove(c)
        pairs = []
```

```
        while len(cons) > 0:
            key = cons.pop()
            pairs.append((key, GraphMaker.nodes[root].get_connection_to(key).length()))
        pairs.sort(key=Solver.sort_by_second)
        dests = []
        while len(pairs) > 0:
            dests.append(pairs.pop(0)[0])
        # while len(cons) > 0:
        #     shortest = 9999999
        #     short_id = -1
        #     for i in range(len(cons)):
        #         test_len = GraphMaker.nodes[root].get_connection_to(cons[i]).length()
        #         if test_len < shortest:
        #             shortest = test_len
        #             short_id = i
        #     dests.append(cons[short_id])
        #     cons.pop(short_id)
        return dests
    @staticmethod
    # return the list of nodes from start to finish
    def get_path():
        if len(Solver.saved_path) == 0:
            nodes = []
            flag = True
            current_id = len(Solver.solver_nodes) - 1
            if Solver.solver_nodes[current_id].previous() == -1:
                return "No valid solution found."
            while flag:
                if current_id == -1:
                    flag = False
                else:
                    nodes.insert(0, current_id)
                    current_id = Solver.solver_nodes[current_id].previous()
            Solver.saved_path = nodes
        return Solver.saved_path
```

GraphMaker.py

```
from FileInput import FileInput
from Node import Node
from Link import Link
class GraphMaker:
    nodes = []
    @staticmethod
    def build_nodes():
        nodes = []
        nums = FileInput.numbers.copy()
        nums = nums[1:nums[0] * 2 + 1]
        while len(nums) > 0:
            n = Node()
            n.set_position((nums.pop(0), nums.pop(0)))
            nodes.append(n)
            # print(str(n))
        GraphMaker.nodes = nodes.copy()
    @staticmethod
    def build_connections():
        count = FileInput.count
        nums = FileInput.numbers.copy()
        cons = nums[FileInput.count * 2 + 1:]
        for f in range(count):
            for t in range(count):
                if cons[count * t + f] == 1:
                    link = Link()
                    link.set_ends(f, t)
                    link.set_length(GraphMaker.calculate_link_length(
                        Node.nodes[f].get_position(), Node.nodes[t].get_position()))
                    GraphMaker.nodes[f].add_connection(link)
    @staticmethod
    def calculate_link_length(s, e):
        return ((e[0] - s[0])**2 + (e[1] - s[1])**2)**(1/2.0)
```

FileInput.py

```
import os
class FileInput:
    file_data = ""
    count = 0
    numbers = []
    @staticmethod
    def do_file_input(this_dir, file_no):
        FileInput.read_file(this_dir, file_no)
        FileInput.process_data()
        FileInput.print_file()
    @staticmethod
    def read_file(this_dir, file_number):
        rel_path = ""
        if file_number == 1:
            rel_path += "..\Caverns\input1.cav"
        elif file_number == 2:
            rel_path += "..\Caverns\input2.cav"
        elif file_number == 3:
            rel_path += "..\Caverns\input3.cav"
        else:
            rel_path += "..\Caverns\input.cav"
        abs_path = os.path.join(this_dir, rel_path)
        print("Looking for file at: " + abs_path)
        with open(abs_path, 'r') as f:
            FileInput.file_data = f.read()
    @staticmethod
    def process_data():
        if len(FileInput.numbers) > 0:
            FileInput.numbers.clear()
        sep = FileInput.file_data.split(',')
        FileInput.count = int(sep[0])
        for val in sep:
            FileInput.numbers.append(int(val))
    @staticmethod
    def print_file():
        print(FileInput.numbers[:1])
        print(FileInput.numbers[1:1 + FileInput.count * 2])
        for i in range(FileInput.count):
            print(FileInput.numbers[1 + FileInput.count * 2 + FileInput.count * i:
                                    1 + FileInput.count * 2 + FileInput.count *
(i+1)])
```

Node.py

```
from Link import Link
from NodeData import NodeData
class Node:
    unique_id = 0
    nodes = []  # list of node data
    def __init__(self):
        Node.nodes.append(NodeData())
        self.id = Node.unique_id
        Node.unique_id += 1
    def __str__(self):
        return "ID:" + str(self.id + 1) + ", " + str(Node.nodes[self.id])
    def set_position(self, pos):
        Node.nodes[self.id].set_position(pos)
    def get_position(self):
        return Node.nodes[self.id].get_position()
    def add_connection(self, con):
        Node.nodes[self.id].add_connection(con)

    def get_links(self):
        return Node.nodes[self.id].get_connections()
    def get_destinations(self):
        dests = []
        for con in Node.nodes[self.id].get_connections():
            dests.append(con.end_node())
        return dests
    def get_connection_to(self, val):
        for con in Node.nodes[self.id].get_connections():
            if con.end_node() == val:
                return con
        return Link()
    def clear_connections(self):
        Node.nodes[self.id].clear_connections()
```

NodeData.py

```
class NodeData:
    def __init__(self):
        self.location = (0, 0)
        self.connections = []  # List of links
    def __str__(self):
        return "Node[loc:" + str(self.location) + ",cons:" + str(len(self.connections))
+ "]"
    def set_position(self, pos):
        self.location = pos
    def get_position(self):
        return self.location
    def add_connection(self, con):
        self.connections.append(con)
    def get_connections(self):
        return self.connections
    def clear_connections(self):
        self.connections.clear()
```

Link.py

```
class Link:
    def __init__(self):
        self.start = 0
        self.end = 0
        self.weight = 0
    def start_node(self):
        return self.start
    def end_node(self):
        return self.end
    def length(self):
        return self.weight
    def set_start(self, s):
        self.start = s
    def set_end(self, e):
        self.end = e
    def set_length(self, l):
        self.weight = l
    def set_ends(self, s, e):
        self.start = s
        self.end = e
```