CMPT317 Assignment 1 Report

Yuqing Tan (Beatrice) (yut630, 11119129)

1. Problem description

- Basic problem:
  On a **X by Y** grid graph, there are **K** pairs of package sources and destinations, and one garage location that contains **N** vehicles. Each vehicle can load at most **P** packages at a time.
  The goal is to take the shortest total time to get all packages delivered.

  Since the multiple vehicles operate at the same time, in my implementation I measure the performance based on:
  Shortest time when the last package is delivered
  However, there are several alternative interpretations about the goal, such as:
  Shortest average delivery time
  Least total cost on vehicles (gas, parking…) but might hurt delivery time

- Achieved variances:
  Creating obstacles inside the grid graph either by deleting nodes or deleting edges
  Different dimensions of X and Y (the graph can be both square and rectangle)

- Other variances that are not implemented:
  Adding weights to edges
  Attaching priorities to the packages
  Solve the problem on a non-grid graph

2. Solution description

- I broke the problem into 3 parts:

Case	Detail	Algorithm
Multiple vehicles – multiple packages	Distributing the K packages into N clusters. Each vehicle will deliver all packages in one cluster.	Farthest first clustering algorithm, containing greedy algorithm inside
One vehicle – multiple packages	Following the previous case. Searching for the shortest path covering a number of points (package sources and destinations). Travelling salesman problem.	A* search (with heuristics determined by greedy best first search)
One vehicle – one package	Following the previous case. Searching for the shortest path from one point to another.	A* search (with heuristics determined by calculating Manhattan distances)

  Will talk about these algorithms backwards.

- A* search #1 (Searching for the shortest path from one point to another)

  State: a state specifies the followings:
  current location
  last visited location (to avoid falling into infinite loop between current and last)
  destination location
  elapsed distance from source location to current location (**g(x)**)
  estimated distance from current location to destination location (**h(x)**)

  Initial state:
  currently at source location
  last visited location is null
  g(x) is 0
  h(x) is Manhattan distance from source to destination

Actions:
move to an adjacent node that doesn't increase the value of g(x) + h(x)
if all adjacent nodes increase the value of g(x) + h(x), restart the search from the node with the lowest g(x) + h(x) value in state queue

Transition model: a state plus an action return a new state where:
current location becomes visited location
destination location unchanged
g(x) increased by 1
h(x) being recalculated

Goal test: checks whether the current location is the destination location

Path cost: each step costs 1 which is the edge weight

- A* search #2 (Searching for the shortest path covering a number of points)

State: a state specifies the followings:
current location
a list of unvisited locations (for packages already picked up, destinations only; for unpicked up, both sources and destinations)
a list of loaded packages
elapsed distance from garage to current location after visiting a number places (**g(x)**)
estimated distance from current location to garage after visiting a number places (**h(x)**)

Initial state:
currently at garage
unvisited locations contain all package sources and destinations
load is empty
g(x) is 0
h(x) is calculated from greedy best first search

Action: move to next available location with the lowest g(x) + h(x) value, remove it from the list of uncovered locations, and modify the list of loaded packages

Transition model: a state plus an action return a new state where:
current location being removed from unvisited locations
if current location is a package source, add the package to loaded packages
if current location is a package destination, remove the package from loaded packages
g(x) is increased by shortest path length from last visited to current location
h(x) is recalculated from greedy best first search (recursively picking the closest available location and continuing search from it, until it covers all package locations and returns back to the garage)

Goal test: checks whether all package locations are covered

Path cost: each step costs the shortest path length from last visited to current location (referring to the previous A* search)

- Farthest first clustering algorithm (a simpler description)

Converting source-destination distances between any 2 of the **K** packages into a weighted graph problem: each package is a node, the estimated cost when delivering 2 packages in one carrier is the edge weight between the 2 nodes.

Picking **N** nodes from the weighted graph with the heaviest possible subgraph (which means these **N** nodes are farthest away from each other and shouldn't be delivered by one carrier), and putting each node into a cluster.

Assigning the un-clustered nodes to one of the clusters with the least weight cost.

3. Implementation Description

- A* search #1 (Searching for the shortest path from one point to another)

```
# A* recursive function for shortest path problem
# return -1 if search fails; otherwise 0
# path information is stored in 'tree' and passed back by reference
Spp_astar_rec(graph, tree, queue, curLoc, dstLoc, g(curLoc)):
        # check if curLoc is already explored
        If tree.exist(curLoc):
                Return -1
        # compare with depth limit, in case the search is stuck in infinite loop
        If g(curLoc) > numNodes(graph):
                Return -1
        # goal test
        If curLoc == dstLoc:
                Return 0
        # add all adjacent nodes to state queue
        adjList = adjNode(curLoc)
        For each Node in adjList:
                Spp_addstate(queue, Node, curLoc, dstLoc, g(node))
        # iterate over state queue
        While queue is not empty:
                State = queue.pop()
                Tree.push(State)
                # successor call
                Res = Spp_astar_rec(graph, tree, queue, State[0], dstLoc, State[2])
                If Res != -1:
                        Return 0
        # will reach here if didn't find a path after exhausting the queue
        Return -1

# adding to state queue
Spp_addstate(queue, curLoc, lastLoc, dstLoc, g(curLoc)):
        # elapsed distance increased by 1
        g(curLoc) += 1
        # heuristics
        h(curLoc) = Manhattan(curLoc, dstLoc)
        newState = (curLoc, lastLoc, g(curLoc), h(curLoc))
        For each State in queue:
                If (g(curLoc) + h(curLoc)) > State[2] + State[3]:
                        Continue
                Else
                        Insert newState into queue before State
                        Break
        Return

# Manhattan distance between 2 points in a grid graph
Manhattan(node1, node2):
        Return |node1(x) – node2(x)| + |node1(y) – node2(y)|
```

State space: state space depends on how many obstacles a graph contains.

Case	Number of possible states
When the graph is perfectly grid (no obstacles)	A = (|srcLoc(x) – dstLoc(x)| + |srcLoc(y) – dstLoc(y)|) * 3 3 times of all points along the path (Manhattan distance)
When the graph has some obstacles but doesn't block all routes within Manhattan distance	B = |srcLoc(x) – dstLoc(x)| * |srcLoc(y) – dstLoc(y)| All points within the Manhattan square

When the graph has blocked all routes within Manhattan distance	Bigger than B

- A* search #2 (Searching for the shortest path covering a number of points)

```
# A* recursive function for traveling salesman problem
# return -1 if search fails; otherwise 0
# path information is stored in 'tree' and passed back by reference
Tsp_astar_rec(graph, tree, queue, curLoc, garage, g(curLoc), loads, capacity, unvisitedLocs):
        # check if curLoc is already explored
        If tree.exist(curLoc):
                Return -1
        # this search doesn't have a depth limit
        # goal test
        If unvisitedLocs is empty:
                Return 0
        # add all available locations to state queue
        For each Location in unvisitedLocs:
                If Location is srcLoc for pkgX && lengthOf(loads) < capacity:
                        newLoads = duplicate(loads)
                        newLoads.add(pkgX)
                        Tsp_addstate(queue, Location, curLoc, garage, g(curLoc), newLoads, capacity, unvisitedLocs)
                If Location is dstLoc for pkgX && loads.exists(pkgX)
                        newLoads = duplicate(loads)
                        newLoads.remove(pkgX)
                        Tsp_addstate(queue, Location, curLoc, garage, g(curLoc), newLoads, capacity, unvisitedLocs)
        # iterate over state queue
        While queue is not empty:
                State = queue.pop()
                Tree.push(State)
                # successor call
                Res = Tsp_astar_rec(graph, tree, queue, State[0], garage, State[2], State[4], capacity, State[5])
                If Res != -1:
                        Return 0
        # will reach here if didn't find a path after exhausting the queue
        Return -1

# adding to state queue
Tsp_addstate(queue, curLoc, lastLoc, garage, g(curLoc), loads, capacity, unvisitedLocs):
        # using A* search #1 to determine shortest path length from lastLoc to curLoc
        # add that to elapsed distance
        g(curLoc) += Spp_astar(lastLoc, curLoc)
        # using greedy best first search to determine the heuristics
        h(curLoc) = Tsp_greedy(curLoc, garage, loads, capacity, unvisitedLocs)
        newState = (curLoc, lastLoc, g(curLoc), h(curLoc), loads, unvisitedLocs)
        For each State in queue:
                If (g(curLoc) + h(curLoc)) > State[2] + State[3]:
                        Continue
                Else
                        Insert newState into queue before State
                        Break
        Return

# greedy best first search
Tsp_greedy(curLoc, garage, loads, capacity, unvisitedLocs)
        totalPath = 0
        While unvisitedLocs is not empty:
                nextLocList = newList()
```

```
        For nextLoc in unvisitedLocs:
            If nextLoc is srcLoc of pkgX && lengthOf(loads) == capacity:
                Continue
            Else
                nextLocDist = Manhattan(curLoc, nextLoc)
                nextLocList.add( [nextLoc, nextLocDist] )
        Find the nextLoc with smallest nextLocDist in nextLocList:
            If nextLoc is srcLoc for pkgX:
                Loads.add(pkgX)
            If nextLoc is dstLoc for pkgX:
                Loads.remove(pkgX)
            totalPath += nextLocDist
            curLoc = nextLoc
    # after visiting all locations, go back to garage
    totalPath += Manhattan(curLoc, garage)
    Return totalPath
```

State space: number of package source + destination locations

4. Results

- A* search #1 (Searching for the shortest path from one point to another)

  Complete: Yes. If srcLoc and dstLoc are connected, a path will be found and the algorithm will return 0; otherwise, the algorithm will return -1.

  Time: Exponential.

  Space: Linear to number of nodes.

  Optimal: Yes. Since it will always pick the node with smallest g(node)+h(node) to explore, the cost won't be bigger than necessary. h(node) is admissible since it's not overestimated (i.e. reaching destination within Manhattan distance is achievable).

- A* search #2 (Searching for the shortest path covering a number of points)

  Complete: Yes. It can always find a solution and halt.

  Time: Exponential.

  Space: Linear to number of packages to deliver.

  Optimal: No. The heuristic is determined from greedy algorithm.

- Farthest first clustering algorithm

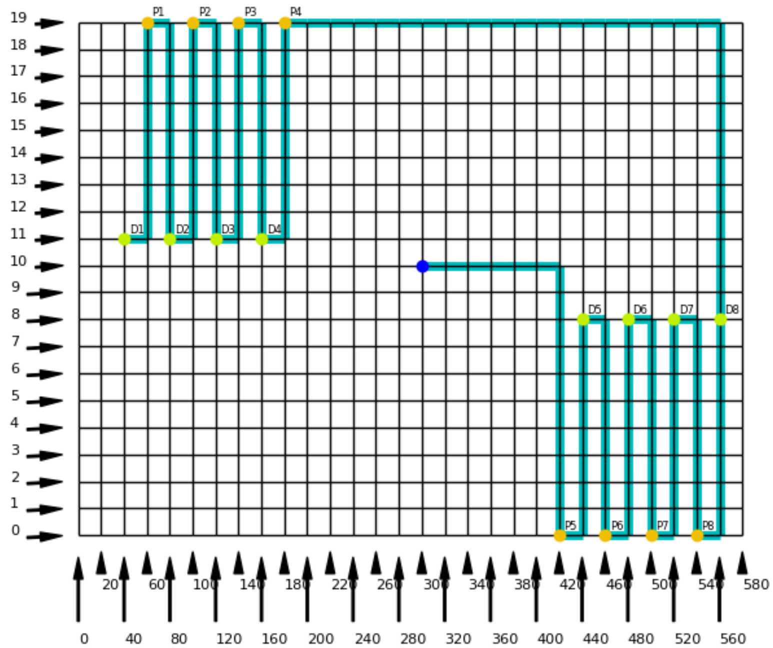
  Complete: No. Sometimes the algorithm returns before each package is assigned to at least one cluster.

  Time: Exponential. It uses combination when determining the farthest nodes and when calculating total cost for each cluster.

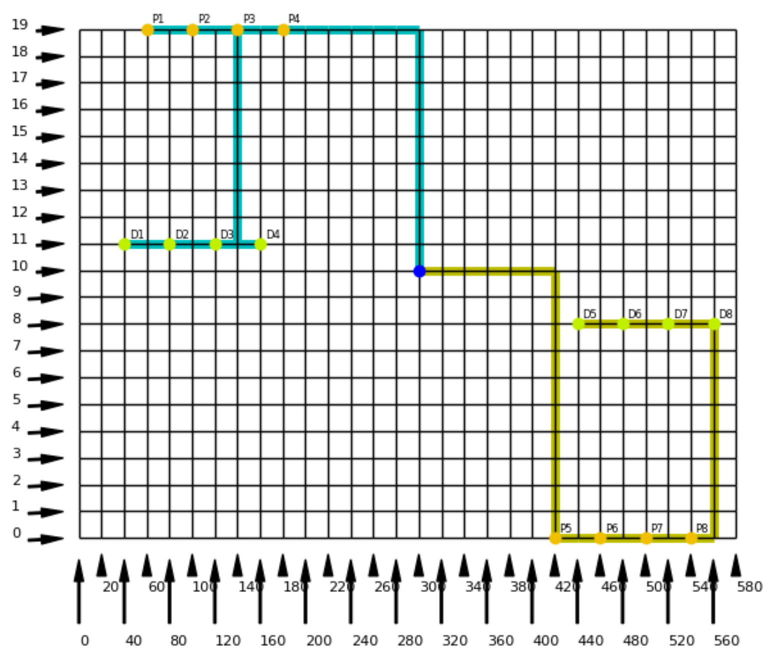
  Space: O(K^2) (need to store weight between any 2 packages)

  Optimal: No. The process of assigning packages to cluster uses greedy algorithm. Somehow the optimality of clustering result is depending on the location of garage.

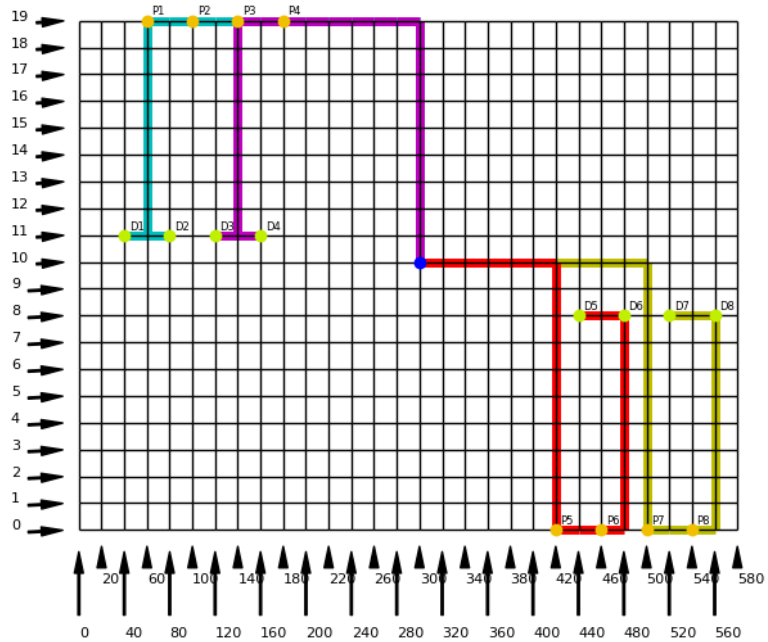
- Sample graphs and solutions
  P = Pickup
  D = Dropoff
  Blue dot is the garage



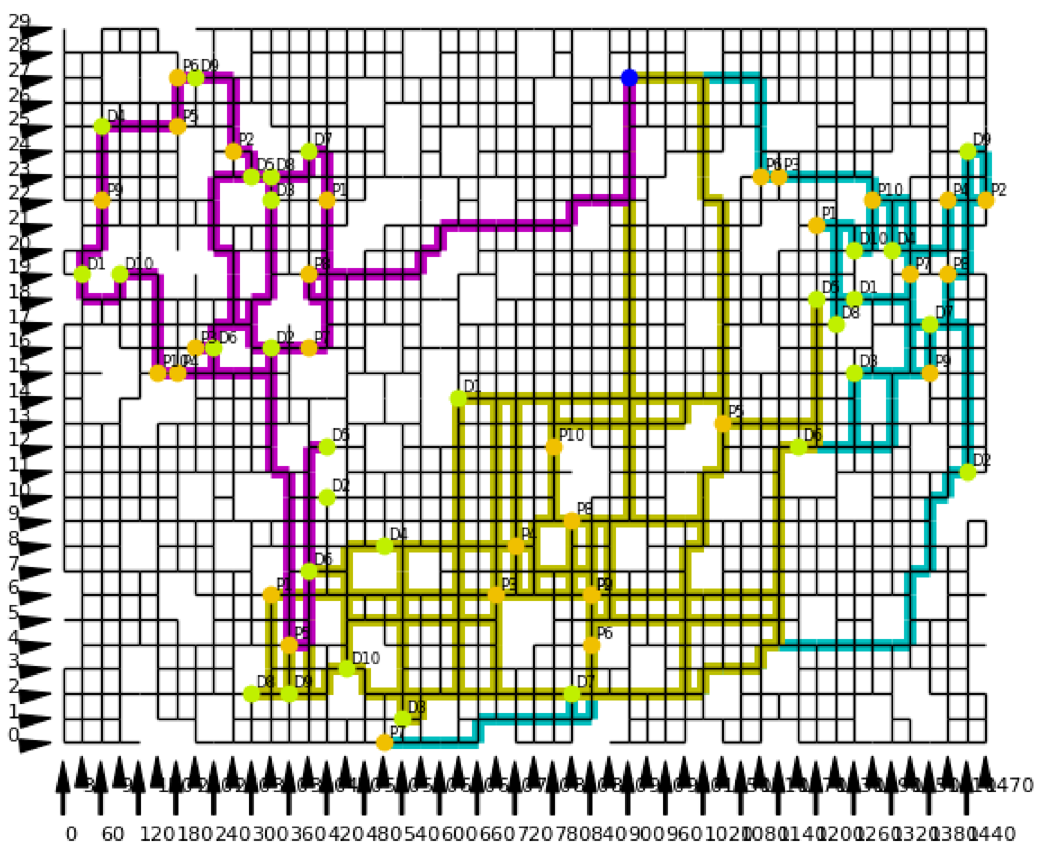
30 * 20 graph (perfect grid)
1 carrier
1 capacity
8 packages
Total route cost: 186



30 * 20 graph (perfect grid)
2 carriers
4 capacities for each carrier
8 packages
Carrier 0 (blue) route cost: 52
Carrier 1 (yellow) route cost: 54



30 * 20 graph (perfect grid)
4 carrier
4 capacities for each carrier
8 packages
Carrier 0 (blue) route cost: 44
Carrier 1 (yellow) route cost: 46
Carrier 2 (magenta) route cost: 36
Carrier 3 (red) route cost: 38



50 * 30 graph (10% nodes being removed)
3 carriers

```
3 capacities for each carrier
30 packages
Carrier 0 (blue) route cost: 178
Carrier 1 (yellow) route cost: 40
Carrier 2 (magenta) route cost: 168
```

5. Conclusions

- Handling large parameters
  My algorithms can handle N and K that are greater than 10, and less than 15% of nodes being removed from graph. However, they can't handle large N and K if more nodes are removed from graph, since they go deep in the stack to search for optimal paths and sometimes hit depth limit of Python recursive calls.

- Known bugs
  Intermittently falling into infinite loop in the 2 A* searches.
  When 2 packages have the same source or destination locations, the A* search for TSP will behave strangely.
  Clustering algorithm is not complete, sometimes some packages are not assigned to clusters or delivered by any carrier.

- Why never use brute force search for traveling salesman problem (search for every possible route), because the time complexity is O(n!)

  I have drawn partial search trees for one vehicle with any capacity, and have the following observations:

Value of K	Number of all possible delivery routes
2	6 = (2*2)! / 2^2
3	90 = (3*2)! / 2^3
4	2520 = (4*2)! / 2^4
5	113400 = (5*2)! / 2^5
generalized	(2K)! / 2^K

  It grows too fast when K is over 5

6. Implementation

- Please see below or the attached file 317a1.py (they have the same content).
- Note that there are 2 automated test functions at the end of implementation:
  Automation1 is randomly generating obstacles, garage location and package locations and then calculating the solution;
  Automation2 is based on a perfect grid graph and all fixed locations;
  Please comment out one of them to execute the other.
- The graph is printed to file '317a1.png' without popping up.

```
## CMPT317 A1
## Yuqing Tan (Beatrice) (yut630, 11119129)

import matplotlib
matplotlib.use('Agg')

import math
import networkx as nx
import random as rand
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

############
# Michael's code
```

```
def makeMap(m, n, gapfreq):
	""" Creates a graph in the form of a grid, with mXn nodes.
	The graph has irregular holes poked into it by random deletion.

	:param m: number of nodes on one dimension of the grid
	:param n: number of nodes on the other dimension
	:param gapfreq: the fraction of nodes to delete (see function prune() below)
	:return: a networkx graph with nodes and edges.

	The default edge weight is  (see below).  The edge weights can be changed by
	designing a list that tells the frequency of weights desired.
	  100% edge weights 1:  [(1,100)]
	  50% weight 1; 50% weight 2: [(1,50),(2,100)]
	  33% each of 1,2,5: [(1,33),(2,67),(5,100)]
	  a fancy distribution:  [(1,10),(4,50),(6,90),(10,100)]
	  (10% @ 1, 40% @ 4, 40% @ 6, 10% @ 10)
	"""
	g = nx.grid_2d_graph(m, n)
	weights = [(1,100)]
	prune(g, gapfreq)
	setWeights(g, weights)
	return g

def setWeights(g, weights):
	""" Use the weights list to set weights of graph g
	:param g: a networkx graph
	:param weights: a list of pairs [(w,cf) ... ]
	:return: nothing

	weights are [(w,cf) ... ]
	w is the weight, cf is the cumulative frequency

	This function uses a uniform random number to index into the weights list.
	"""
	for (i, j) in nx.edges(g):
		c = rand.randint(1,100)
		w = [a for (a,b) in weights if b >= c] # drop all pairs whose cf is < c
		g.edge[i][j]['weight'] = w[0]  # take the first weight in w
	return

def draw(g, dimx, dimy, filename):
	""" Draw the graph, just for visualization.  Also creates a jpg in $CWD
	:param g: a networkx graph
	:return:
	"""

	# print out coordinates
	for i in range(0, dimx):
		str = "%d" % (i*dimy)
		if math.floor(i/2.0) == i/2.0:
			y = -4
		else:
			y = -2
		plt.annotate(str, xy=(i, -0.5), xytext=(i, y), size=8, \
			arrowprops=dict(facecolor='black', shrink=0.1, width=1, headwidth=5))
	for j in range(0, dimy):
		str = "%d" % (j)
		plt.annotate(str, xy=(-0.5, j), xytext=(-3, j), size=8, \
```

```
                    arrowprops=dict(facecolor='black', shrink=0.1, width=1, headwidth=5))

        # print nodes and edges
        pos = {n: n for n in nx.nodes(g)}
        #nx.draw_networkx_nodes(g, pos, node_size=3, node_color='k')
        edges = nx.edges(g)
        nx.draw_networkx_edges(g, pos, edgelist=edges, width=1)
        plt.axis('off')
        plt.savefig(filename)  # save as png
        #plt.show()  # display
        return

def prune(g, gapf):
        """ Poke random holes the graph g by deleting random nodes, with probability gapf.
        Then clean up by deleting all but the largest connected component.

        Interesting range (roughly):  0.1 < gapf < 0.3
        values too far above 0.3 lead to lots of pruning, but rather smaller graphs

        :param g: a networkx graph
        :param gapf: a fraction in [0,1]
        :return: nothing
        """

        #there are 2 ways to create obstacles in a map...

        #removing edges (looks like a maze)
        #for edge in nx.edges(g):
        #        if rand.random() < gapf:
        #                g.remove_edge(edge[0],edge[1])


        #removing nodes (looks like a broken window screen)
        for node in nx.nodes(g):
                if rand.random() < gapf:
                        g.remove_node(node)

        # deleting all but the largest connected component...
        comps = sorted(nx.connected_components(g), key=len, reverse=False)
        while len(comps) > 1:
                nodes = comps[0]
                for node in nodes:
                        g.remove_node(node)
                comps.pop(0)

        return

# End of Michael's code
############

############
# Start of my algorithms
############

############
# Section1: building structure and initializing the puzzle

# convert the graph to unified node index
# nodes stored in list; index = x*dim+y; pruned nodes stored as (-1, -1)
```

```
def unifynode(g, dimx, dimy):
	nl = sorted(nx.nodes(g)) #node list
	numnode = nx.number_of_nodes(g);
	ug = []

	for i in range(0,dimx*dimy):
		ug.append((-1,-1))

	for j in range(0,numnode):
		x = nl[j][0]
		y = nl[j][1]
		ug[x*dimy+y] = nl[j]

	return ug

# create connectivity list
# list of 2-5 tuples; [node, adj-node1, adj-node2, ...]
def connect(g, ug, dimy):
	nl = sorted(nx.nodes(g)) #node list
	el = nx.edges(g) #edge list
	cl = [] #connectivity list

	numnode = len(ug)
	#dim = int(math.floor(math.sqrt(len(ug))))
	for i in range(0,numnode):
		curitem = []
		curitem.append(ug[i])
		cl.append(curitem)

	numedge = nx.number_of_edges(g);
	for j in range(0,numedge):
		node1=el[j][0]
		node2=el[j][1]
		n1index = node1[0]*dimy+node1[1]
		n2index = node2[0]*dimy+node2[1]
		cl[n1index].append(n2index)
		cl[n2index].append(n1index)

	return cl

# initialize the puzzle with k pairs of package sources and destinations
def initpack(g, dimy, k):
	nl = sorted(nx.nodes(g)) #node list
	numnode = nx.number_of_nodes(g);
	packlist = [] #store node indices only
	slist = [] #store list of source nodes for printing
	dlist = [] #store list of destination nodes for printing

	for i in range(0,k):
		srcgindex = int(rand.random()*numnode)
		dstgindex = int(rand.random()*numnode)
		srcgnode = nl[srcgindex]
		dstgnode = nl[dstgindex]
		srcugindex = srcgnode[0]*dimy+srcgnode[1]
		dstugindex = dstgnode[0]*dimy+dstgnode[1]
		packlist.append([srcugindex, dstugindex])
		slist.append(srcgnode)
		dlist.append(dstgnode)
```

```
                str = "P%d" % (i+1)
                plt.annotate(str, xy=srcgnode, xytext=(srcgnode[0]+0.2, srcgnode[1]+0.2), size=6)
                str = "D%d" % (i+1)
                plt.annotate(str, xy=dstgnode, xytext=(dstgnode[0]+0.2, dstgnode[1]+0.2), size=6)

        pos = {n: n for n in nx.nodes(g)}
        nx.draw_networkx_nodes(g, pos, nodelist=slist, node_size=50, node_color='#f0c000')
        nx.draw_networkx_nodes(g, pos, nodelist=dlist, node_size=50, node_color='#c0f000')
        return packlist

# initialize package locations with area bounds
def initpack_bound(g, cl, dimy, k, xrange, yrange):
        packlist = [] #store node indices only
        slist = [] #store list of source nodes for printing
        dlist = [] #store list of destination nodes for printing
        numpack = 0

        #for i in range(0,k):
        while numpack < k:
                src_x = int(rand.random() * (xrange[1]-xrange[0]) + xrange[0])
                src_y = int(rand.random() * (yrange[1]-yrange[0]) + yrange[0])
                dst_x = int(rand.random() * (xrange[1]-xrange[0]) + xrange[0])
                dst_y = int(rand.random() * (yrange[1]-yrange[0]) + yrange[0])
                srcindex = src_x*dimy + src_y
                dstindex = dst_x*dimy + dst_y
                if cl[srcindex][0][0] == -1 or cl[dstindex][0][0] == -1:
                        continue

                packlist.append([srcindex, dstindex])
                slist.append((src_x, src_y))
                dlist.append((dst_x, dst_y))
                numpack += 1

                str = "P%d" % (numpack)
                plt.annotate(str, xy=(src_x, src_y), xytext=(src_x+0.2, src_y+0.2), size=6)
                str = "D%d" % (numpack)
                plt.annotate(str, xy=(dst_x, dst_y), xytext=(dst_x+0.2, dst_y+0.2), size=6)

        pos = {n: n for n in nx.nodes(g)}
        nx.draw_networkx_nodes(g, pos, nodelist=slist, node_size=50, node_color='#f0c000')
        nx.draw_networkx_nodes(g, pos, nodelist=dlist, node_size=50, node_color='#c0f000')
        return packlist

# dummy function for testing
def initpack_dummy(g, cl, packlist):
        slist = [] #store list of source nodes for printing
        dlist = [] #store list of destination nodes for printing

        for i in range(0,len(packlist)):
                srcgnode = cl[packlist[i][0]][0]
                dstgnode = cl[packlist[i][1]][0]
                slist.append(srcgnode)
                dlist.append(dstgnode)

                str = "P%d" % (i+1)
                plt.annotate(str, xy=srcgnode, xytext=(srcgnode[0]+0.2, srcgnode[1]+0.2), size=6)
                str = "D%d" % (i+1)
                plt.annotate(str, xy=dstgnode, xytext=(dstgnode[0]+0.2, dstgnode[1]+0.2), size=6)
```

```
	pos = {n: n for n in nx.nodes(g)}
	nx.draw_networkx_nodes(g, pos, nodelist=slist, node_size=50, node_color='#f0c000')
	nx.draw_networkx_nodes(g, pos, nodelist=dlist, node_size=50, node_color='#c0f000')
	return packlist

# initialize the puzzle with a vehicle garage
def initcar(g, dimy):
	nl = sorted(nx.nodes(g)) #node list
	numnode = nx.number_of_nodes(g);
	nlist = []

	vehigindex = int(rand.random()*numnode)
	vehignode = nl[vehigindex]
	vehiugindex = vehignode[0]*dimy+vehignode[1]
	nlist.append(vehignode)

	pos = {n: n for n in nx.nodes(g)}
	nx.draw_networkx_nodes(g, pos, nodelist=nlist, node_size=50, node_color='b')
	return vehiugindex

# End of Section1
############

############
# Section2: declaring common routines

# distance calculation mode
def distmode(g, cl, i1, i2):
	return manhattan(cl, i1, i2)
	#return len(spp_astar_main(g, cl, i1, i2)) #very slow, takes >10 mins

# return manhattan distance given 2 nodes
def manhattan(cl, i1, i2):
	n1 = cl[i1][0]
	n2 = cl[i2][0]
	dist = 0
	if n1[0] > n2[0]:
		dist += n1[0] - n2[0]
	else:
		dist += n2[0] - n1[0]
	if n1[1] > n2[1]:
		dist += n1[1] - n2[1]
	else:
		dist += n2[1] - n1[1]
	return dist

def min(val1, val2):
	if val1 < val2:
		return val1
	else:
		return val2

def exist(list, elem):
	ret = 0
	for i in range(0,len(list)):
		if list[i]==elem:
			ret = 1
	return ret
```

```
"""
def safeinsert(list, elem):
	e = exist(list, elem)
	if e==0:
		list.insert(0,elem)
	return

def saferemove(list, elem):
	e = exist(list, elem)
	if e==1:
		list.remove(elem)
	return
"""

# duplicate linear list
def duplist(list):
	newlist = []
	for i in range(0, len(list)):
		newlist.append(list[i])
	return newlist

# duplicate 2 dimensional list
def duplist2(list):
	newlist = emptylist(len(list))
	for i in range(0, len(list)):
		newlist[i] = duplist(list[i])
	return newlist

# pop from 2 dimensional list
def poplist(list, index):
	list[index].pop(0)
	if len(list[index]) == 0:
		list.pop(index)
	return

# create list of natural numbers
def numberlist(n):
	list = []
	for i in range(0, n):
		list.append(i)
	return list

# create empty list
def emptylist(n):
	list = []
	for i in range(0, n):
		list.append([])
	return list

# translate indices into nodes
def index2node(cl, indexlist):
	nodelist = emptylist(len(indexlist))
	for i in range(0, len(indexlist)):
		for j in range(0, len(indexlist[i])):
			nodelist[i].append(cl[indexlist[i][j]][0])
	return nodelist

# End of Section2
############
```

```
############
# Section3: SPP (shortest path problem)
#           case: one vehicle - one package
#           determining optimal path between 2 points on a grid graph (with obstacles)
#           A* search, heuristics decided with Manhattan distance

# recursively remove terminal nodes to accelerate path search
def spp_removenode(g, cl, keeplist):
    scc = cl #strongly connected components
    todraw = []
    qualify = 0
    skipcheck = 0
    while qualify == 0:
        qualify = 1
        for i in range(0,len(scc)):
            skipcheck == 0
            if len(scc[i])==2:
                for j in range(0,len(keeplist)):
                    if keeplist[j] == i:
                        skipcheck = 1

            if len(scc[i])==2 and skipcheck == 0:
                conn = scc[i][1]
                scc[conn].remove(i)
                todraw.append(scc[i][0])
                scc[i] = [(-1,-1)]
                qualify = qualify*0
    pos = {n: n for n in nx.nodes(g)}
    #nx.draw_networkx_nodes(g, pos, nodelist=todraw, node_size=50, node_color='y')
    return scc

def spp_addstate(cl, list, tonode, fromnode, dst, elapdist):
    if tonode<0 or tonode>=len(cl):
        return
    if cl[tonode][0][0] == -1:
        return
    e = exist(cl[tonode], fromnode)
    if e==0:
        return
    elapdist += 1
    estmdist = manhattan(cl, tonode, dst)
    tuple = (tonode, fromnode, elapdist, estmdist)
    if len(list) == 0:
        list.insert(0,tuple)
        return
    for i in range(0, len(list)):
        if elapdist + estmdist > list[i][2] + list[i][3]:
            continue
        else:
            list.insert(i,tuple)
            return
    list.append(tuple)
    return

# A* search recursive function
# state: 4-tuple (to-node, from-node, elapdist, estmdist)
#                   from-node and to-node represent an edge
#                   elapdist: elapsed distance from src to to-node
```

```
#			estmdist: manhattan distance from to-node to dst
# tree: stores edges (4-tuples) already explored
#			sequence: later -> former explored
# stack: stores edges (4-tuples) not explored
#			primary sequence: elapdist+estmdist low -> high
#			secondary sequence: later -> former added
# return: -1 for failing to reach destination; 0 for success
def spp_astar_rec(g, cl, src, dst, tree, stack, elapdist):
	fail = -1
	succ = 0
	if cl[src][0][0]==-1 or len(tree)>len(cl):
		return fail
	if src == dst:
		tree.insert(0, (dst,dst,elapdist,0))
		return succ
	for j in range(0, len(tree)):
		if tree[j][1]==src or tree[j][1]==dst:
			return fail

	#adjacency list
	adjlist = []
	for i in range(1, len(cl[src])):
		adjlist.append(cl[src][i])
	adjlist = sorted(adjlist)
	for j in range(0, len(tree)):
		for k in range(0, len(adjlist)):
			if tree[j][1]==adjlist[k]:
				adjlist[k]=-1
	for k in range(0, len(adjlist)):
		spp_addstate(cl, stack, adjlist[k], src, dst, elapdist)

	pos = {n: n for n in nx.nodes(g)}
	todraw = []
	while len(stack) > 0:
		#tuple = (tonode, fromnode, elapdist, estmdist)
		tuple = stack.pop(0)
		tree.insert(0, tuple)
		d = spp_astar_rec(g, cl, tuple[0], dst, tree, stack, tuple[2])
		#todraw.insert(0,(cl[tuple[0]][0],cl[tuple[1]][0]))
		#nx.draw_networkx_edges(g, pos, edgelist=todraw, width=5, edge_color='m')
		if d != fail:
			return succ

	return fail

def spp_astar_main(g, cl, i1, i2):
	scc=spp_removenode(g, cl, [i1, i2])
	path = []
	tree = []
	stack = []
	elapdist = 0
	d = spp_astar_rec(g, scc, i1, i2, tree, stack, elapdist)

	#translate tree into path
	if len(tree) > 1:
		carry = 0
		for i in range(1, len(tree)):
			if tree[i][0] == tree[carry][1] and tree[i][2] <= tree[carry][2]:
				path.append((cl[tree[i][0]][0], cl[tree[i][1]][0]))
```

```
                                        carry = i

        return path

# End of Section3
############

############
# Section4: TSP (traveling salesman problem)
#       case: one vehicle - multiple packages
#       determining optimal path covering a number of points
#       A* search, heuristics decided with greedy algorithm

# using greedy algorithm to determine heuristics
def tsp_greedy(g, cl, curindex, garage, packlist, load, capa):
        totalpath = 0
        adjlist = []
        while len(packlist) > 0:
                for i in range(0, len(packlist)):
                        if len(load) >= capa and len(packlist[i]) == 2:
                                continue
                        pathlen = distmode(g, cl, curindex, packlist[i][0])
                        adjlist.append((i, packlist[i][0], pathlen))

                nextindex = -1
                min_i = -1
                min_dist = len(cl)
                for j in range(0, len(adjlist)):
                        if adjlist[j][2] < min_dist:
                                min_i = adjlist[j][0]
                                nextindex = adjlist[j][1]
                                min_dist = adjlist[j][2]
                totalpath += min_dist

                if len(packlist[min_i]) == 2:
                        load.append(packlist[min_i][1])
                if len(packlist[min_i]) == 1:
                        load.remove(packlist[min_i][0])
                poplist(packlist, min_i)
                curindex = nextindex
                adjlist = []

        goback = distmode(g, cl, curindex, garage)
        totalpath += goback
        return totalpath

# tonode is the index inside packlist
# fromnode is the index inside cl
def tsp_addstate(g, cl, list, tonode, fromnode, garage, elapdist, packlist, load, capa):
        if len(load) >= capa and len(packlist[tonode]) == 2:
                return
        if len(load) < capa and len(packlist[tonode]) == 2:
                load.append(packlist[tonode][1])
        if len(packlist[tonode]) == 1:
                load.remove(packlist[tonode][0])

        #pathlen = manhattan(cl, fromnode, packlist[tonode][0])
        pathlen = len(spp_astar_main(g, cl, fromnode, packlist[tonode][0]))
        elapdist += pathlen
```

```
    packlist2 = duplist2(packlist)
    poplist(packlist2, tonode)
    load2 = duplist(load)
    estmdist = tsp_greedy(g, cl, packlist[tonode][0], garage, packlist2, load2, capa)

    packlist4 = duplist2(packlist)
    load4 = duplist(load)
    tuple = (tonode, fromnode, elapdist, estmdist, packlist4, load4)

    if len(list) == 0:
        list.insert(0,tuple)
        return
    for i in range(0, len(list)):
        if elapdist + estmdist > list[i][2] + list[i][3]:
            continue
        else:
            list.insert(i,tuple)
            return
    list.append(tuple)
    return

# A* search combined with greedy
def tsp_astar_rec(g, cl, tree, stack, curindex, garage, elapdist, packlist, load, capa, color):
    fail = -1
    succ = 0
    pos = {n: n for n in nx.nodes(g)}

    if len(packlist) == 0:
        goback = len(spp_astar_main(g, cl, curindex, garage))
        tree.insert(0, (-1, curindex, elapdist+goback, 0, [], []))
        return succ

    for i in range(0, len(tree)):
        if tree[i][1] == curindex:
            return fail

    tempload = duplist(load)
    for i in range(0, len(packlist)):
        load = duplist(tempload)
        if len(load) < capa:
            tsp_addstate(g, cl, stack, i, curindex, garage, elapdist, packlist, load, capa)
        elif len(packlist[i]) < 2:
            tsp_addstate(g, cl, stack, i, curindex, garage, elapdist, packlist, load, capa)

    while len(stack) > 0:
        #tuple = (tonode, fromnode, elapdist, estmdist, packlist, loads)
        tuple = stack.pop(0)

        tonode = tuple[4][tuple[0]][0]
        path = spp_astar_main(g, cl, tuple[1], tonode)
        nx.draw_networkx_edges(g, pos, edgelist=path, width=5, edge_color=color)
        poplist(tuple[4], tuple[0])
        tree.insert(0, tuple)

        d = tsp_astar_rec(g, cl, tree, stack, tonode, garage, tuple[2], tuple[4], tuple[5], capa, color)
        if d != fail:
            return succ
```

```
        return fail

def tsp_astar_main(g, cl, cluster, garage, capa, paths, costs):
        colors = ['c', 'y', 'm', 'r', 'g', 'b']
        for i in range(0, len(cluster)):
                tree = []
                stack = []
                load = []
                elapdist = 0
                k = i
                while k > 6:
                        k -= 6
                tsp_astar_rec(g, cl, tree, stack, garage, garage, elapdist, cluster[i], load, capa, colors[k])

                #for j in range(0, len(tree)):
                #       print(tree[j])
                #print("")

                #paths and costs are passed back by reference
                costs.append(tree[0][2])
                paths.append([])
                for h in range(0, len(tree)):
                        paths[i].append(cl[tree[h][1]][0])

        return

# End of Section4
############

############
# Section5: FFC (farthest first clustering)
#       case: multiple vehicles - multiple packages
#       converting distance relation of K packages into weighted graph problem
#       clustering K points into N groups (to be delivered by N vehicles)
#       clustering uses greedy algorithm

# G-Garage, P-Pickup(source), D-Dropoff(destination)
# let (P1, D1, P2, D2, P3, D3 ...) represent pickup/dropoff locations
# PGsum: sum of (P1-G, P2-G ...) and (D1-G, D2-G ...) distances
# PDsum: sum of (P1-D1, P2-D2 ...) distances
# PPsum: sum of (P1-P2, P2-P3 ...) and (D1-D2, D2-D3 ...) distances
# PDcsum: sum of (P1-D2, P2-D1 ...) distances
# ecost: estimated lowest cost based on calculations of the above 4 values

# structure 6-tuple (ecost, [pkg-list], (PGsum,nPG), (PDsum,nPD), (PPsum,nPP), (PDcsum,nPDc))
# numpkg = 1                    (ecost, [pkg1], (PGsum,2), (PDsum,1), (PPsum,0), (PDcsum,0))
# numpkg = 2            (ecost, [pkg1, pkg2], (PGsum,4), (PDsum,2), (PPsum,2), (PDcsum,2))
# numpkg = 3    (ecost, [pkg1, pkg2, pkg3], (PGsum,6), (PDsum,3), (PPsum,6), (PDcsum,6))
# numpkg = 4 (ecost, [pkg1, pkg2, pkg3, pkg4], (PGsum,8), (PDsum,4), (PPsum,12), (PDcsum,12))

"""
# estimated cost to deliver 1 package
def primedist(g, cl, packlist, garage, pkgindex):
        pkg = packlist[pkgindex]
        PGsum = distmode(g, cl, garage, pkg[0]) + \
                        distmode(g, cl, garage, pkg[1])
        PDsum = distmode(g, cl, pkg[0], pkg[1])
        return (PDsum, [pkgindex], (PGsum,2), (PDsum,1), (0,0), (0,0))
"""
```

```
# estimated cost to deliver 2 packages
def seconddist(g, cl, packlist, garage, pkgindex1, pkgindex2, capa):
	pkg1 = packlist[pkgindex1]
	pkg2 = packlist[pkgindex2]
	PGsum = distmode(g, cl, garage, pkg1[0]) + \
				distmode(g, cl, garage, pkg1[1]) + \
				distmode(g, cl, garage, pkg2[0]) + \
				distmode(g, cl, garage, pkg2[1])
	PDsum = distmode(g, cl, pkg1[0], pkg1[1]) + \
				distmode(g, cl, pkg2[0], pkg2[1])
	PPsum = distmode(g, cl, pkg1[0], pkg2[0]) + \
				distmode(g, cl, pkg1[1], pkg2[1])
	PDcsum = distmode(g, cl, pkg1[0], pkg2[1]) + \
				 distmode(g, cl, pkg1[1], pkg2[0])

	npack = 2.0
	ncomb = npack*(npack-1) #number of combinations
	if capa > 2:
		capa = 2.0
	#total number of connections
	totalconn = npack*2-1
	#max number of connections that are inter-sources or inter-destinations
	#(pickup A, pickup B, dropoff A, dropoff B) count as 2
	interconn = int(math.floor( npack/capa*(capa-1) )) * 2

	#estimated cost when using one capacity (pickup A, dropoff A, pickupB, dropoffB ...)
	capa1 = PDsum + PDcsum/npack
	#estimated cost when using up n capacity
	capan = PPsum/ncomb * interconn + (PDsum+PDcsum)/(npack*npack) * (totalconn-interconn)
	#print("seconddist capa1=%d capan=%d" % (capa1, capan))

	return (min(capa1, capan), [pkgindex1, pkgindex2], (PGsum,4), (PDsum,2), (PPsum,2), (PDcsum,2))

# estimated cost to deliver more than 2 packages
# twodist is a list generated from seconddist()
def multidist(g, cl, twodist, nodelist, capa):
	templist = [] #store information for later calculation
	npack = float(len(nodelist))
	ncomb = float(npack*(npack-1)) #number of combinations
	if capa > npack:
		capa = npack
	i = 0
	while len(templist) < ncomb/2 and i < len(twodist):
		elem1 = twodist[i][1][0]
		elem2 = twodist[i][1][1]
		e1 = exist(nodelist, elem1)
		e2 = exist(nodelist, elem2)
		if e1==1 and e2==1:
			templist.append(twodist[i])
		i+=1
	if len(templist) != ncomb/2:
		return -1;
	PGsum = 0 #sum of 2n values
	PDsum = 0 #sum of n values
	PPsum = 0 #sum of n(n-1) values
	PDcsum = 0 #sum of n(n-1) values
	for i in range(0, len(templist)):
		PGsum += templist[i][2][0]
```

```
            PDsum += templist[i][3][0]
            PPsum += templist[i][4][0]
            PDcsum += templist[i][5][0]
        PGsum = PGsum/(npack-1)
        PDsum = PDsum/(npack-1)

        #total number of connections
        totalconn = npack*2-1
        #max number of connections that are inter-sources or inter-destinations
        #(pickup A, pickup B, dropoff A, dropoff B) count as 2
        interconn = int(math.floor( npack/capa*(capa-1) )) * 2

        #estimated cost when using one capacity (pickup A, dropoff A, pickupB, dropoffB ...)
        capa1 = PDsum + PDcsum/npack
        #estimated cost when using up n capacity
        capan = PPsum/ncomb * interconn + (PDsum+PDcsum)/(npack*npack) * (totalconn-interconn)
        #print("multidist capa1=%d capan=%d" % (capa1, capan))

        return min(capa1, capan)

# combination (not permutation)
# choose N (num) items from list, no repetition
def combination(prefix, list, count, result):
        if count == 0:
                result.append(prefix)
                return
        for i in range(0, len(list)):
                pprefix = duplist(prefix)
                pprefix.append(list[i])
                combination(pprefix, list[i+1:], count - 1, result)
        return

def deletion(combinelist, edge):
        rmlist = []
        for i in range(0, len(combinelist)):
                e1 = exist(combinelist[i], edge[0])
                e2 = exist(combinelist[i], edge[1])
                if e1==1 and e2==1:
                        rmlist.append(i)
        for i in range(0, len(rmlist)):
                if rmlist[i] < len(combinelist):
                        combinelist.pop(rmlist[i])
        return

# farthest first clustering algorithm
# K (packlist) list of package pickup-dropoff points
# N (ncar) number of carriers
# P (capa) capacity of each carrier
def cluster(g, cl, packlist, garage, ncar, capa):

        if ncar == 1:
                return [packlist]

        clusters = emptylist(ncar)
        npack = len(packlist)

        #FIRST, convert relationship between K packages into weighted graph problem
        """
        onedist = [] #estimated distance delivering one package
```

```
for i in range(0, npack):
		distinfo = primedist(g, cl, packlist, garage, i)
		onedist.append(distinfo)
"""
#estimated distance delivering any two packages in one carrier
twodist = []
for j in range(0, npack):
		for k in range(j+1, npack):
				distinfo = seconddist(g, cl, packlist, garage, j, k, capa)
				twodist.append(distinfo)
twodist = sorted(twodist)
#print("twodist")
#for g in range(0, len(twodist)):
#		print(twodist[g])
#print("")

distqueue = emptylist(npack)
for l in range(0, len(twodist)):
		list1 = distqueue[twodist[l][1][0]]
		list2 = distqueue[twodist[l][1][1]]
		list1.append(l);
		list2.append(l);
for h in range(0, npack):
		distqueue[h].append(h)
distqueue = sorted(distqueue)
#print("distqueue")
#print(distqueue)

#SECOND, determine farthest N nodes, put each as a cluster
#ncar must be >=2 to get to this step
elem1 = twodist[len(twodist)-1][1][0]
elem2 = twodist[len(twodist)-1][1][1]
clusters[0].append(elem1)
clusters[1].append(elem2)
unclustered = numberlist(npack)
unclustered.remove(elem1)
unclustered.remove(elem2)
if ncar > 2:
		prefix = []
		farlist = []
		combination(prefix, unclustered, ncar-2, farlist)
		for m in range(0, len(farlist)):
				farlist[m].append(elem1)
				farlist[m].append(elem2)
		n = 0
		while n < len(twodist) and len(farlist) > 1:
				deletion(farlist, twodist[n][1])
				n+=1
		for o in range(2, ncar):
				clusters[o].append(farlist[0][o-2])
				unclustered.remove(farlist[0][o-2])
#print("clusters")
#print(clusters)
#print("unclustered")
#print(unclustered)

#THIRD, distribute the rest nodes to these clusters with the least cost
s = 0 # to prevent infinite loop
while len(unclustered) > 0 and s < npack-1:
```

```
			s += 1
			p = npack-1
			while p > -1:
				tmplist0 = distqueue[p]
				e = exist(unclustered, tmplist0[len(tmplist0)-1])
				if e==0: #already in cluster
					nextnode = -1
					elem1 = twodist[tmplist0[s]][1][0]
					elem2 = twodist[tmplist0[s]][1][1]
					e1 = exist(unclustered, elem1)
					e2 = exist(unclustered, elem2)
					if e1==0 and e2==1:
						nextnode = elem2
					if e1==1 and e2==0:
						nextnode = elem1
					if nextnode==-1:
						p-=1
						continue

					tmplist1 = []
					for q in range(0, ncar):
						tmplist2 = duplist(clusters[q])
						tmplist2.insert(0, nextnode)
						#this value is estimated for comparison
						ecost = multidist(g, cl, twodist, tmplist2, capa)
						tmplist2.insert(0, ecost)
						tmplist1.append(tmplist2)
					tmplist1 = sorted(tmplist1)
					#print(tmplist1)
					if tmplist1[0][0] == tmplist1[1][0]:
						p-=1
						continue
					else:
						for r in range(0, ncar):
							if clusters[r][len(clusters[r])-1] == tmplist1[0][len(tmplist1[0])-1]:
								clusters[r].insert(0, nextnode)
								unclustered.remove(nextnode)
								break
				p-=1
				#print("clusters")
				#print(clusters)
				#print("unclustered")
				#print(unclustered)
				#print("\n")
				if len(unclustered) == 0:
					break

		#clusters translation: from package # to graph index
		result = emptylist(ncar)
		for u in range(0, ncar):
			for v in range(0, len(clusters[u])):
				result[u].append([packlist[clusters[u][v]][0], packlist[clusters[u][v]][1]])

		return result

# End of Section5
############

############
```

```
# Section6: automated tests

# random test
def automation1():
	# define the graph
	dimx = 50
	dimy = 30
	gapfreq = 0.1
	graph = makeMap(dimx, dimy, gapfreq) # a rectangle graph
	ugraph = unifynode(graph, dimx, dimy)
	cl=connect(graph, ugraph, dimy)
	print("number of nodes: %d" % nx.number_of_nodes(graph))
	print("")

	garage = initcar(graph, dimy)
	print("vehicle garage:")
	print(cl[garage][0])
	print("")

	# define 30 packages
	packlist1 = initpack_bound(graph, cl, dimy, 10, [0,15], [15,30])
	packlist2 = initpack_bound(graph, cl, dimy, 10, [35,50], [10,25])
	packlist3 = initpack_bound(graph, cl, dimy, 10, [10,30], [0,15])
	packlist = []
	packlist.extend(packlist1)
	packlist.extend(packlist2)
	packlist.extend(packlist3)

	# cluster packages
	ncar = 3   # N
	capa = 3   # P
	clus = cluster(graph, cl, packlist, garage, ncar, capa)
	clus_node = emptylist(ncar)
	print("package assigned to clusters:")
	for i in range(0, ncar):
		clus_node[i] = index2node(cl, clus[i])
		print("cluster %d" % i)
		for j in range(0, len(clus_node[i])):
			print(clus_node[i][j])
	print("")

	# assign each cluster to a carriage
	paths = []
	costs = []
	tsp_astar_main(graph, cl, clus, garage, capa, paths, costs)
	for i in range(0, ncar):
		print("carrier %d total cost: %d" % (i, costs[i]))
		print(paths[i])
		print("")

	draw(graph, dimx, dimy, "317a1.png")
	return

# fixed points test
def automation2():
	# define the graph
	dimx = 30
	dimy = 20
	gapfreq = 0
```

```
	graph = makeMap(dimx, dimy, gapfreq) # a rectangle graph
	ugraph = unifynode(graph, dimx, dimy)
	cl=connect(graph, ugraph, dimy)
	print("number of nodes: %d" % nx.number_of_nodes(graph))
	print("")

	# define one garage
	garage = 310
	nlist = [(15,10)]
	pos = {n: n for n in nx.nodes(graph)}
	nx.draw_networkx_nodes(graph, pos, nodelist=nlist, node_size=50, node_color='b')
	print("vehicle garage:")
	print(cl[garage][0])
	print("")

	# define 8 packages
	packlist = [[79, 51], [119, 91], [159, 131], [199, 171], [420, 448], [460, 488], [500, 528], [540, 568]]
	initpack_dummy(graph, cl, packlist)
	packlist_node = index2node(cl, packlist)
	print("package sources/destinations:")
	for i in range(0, len(packlist_node)):
		print(packlist_node[i])
	print("")

	# cluster packages
	ncar = 2   # N
	capa = 2   # P
	clus = cluster(graph, cl, packlist, garage, ncar, capa)
	clus_node = emptylist(ncar)
	print("package assigned to clusters:")
	for i in range(0, ncar):
		clus_node[i] = index2node(cl, clus[i])
		print("cluster %d" % i)
		for j in range(0, len(clus_node[i])):
			print(clus_node[i][j])
	print("")

	# assign each cluster to a carriage
	paths = []
	costs = []
	tsp_astar_main(graph, cl, clus, garage, capa, paths, costs)
	for i in range(0, ncar):
		print("carrier %d total cost: %d" % (i, costs[i]))
		print(paths[i])
		print("")

	draw(graph, dimx, dimy, "317a1.png")
	return

# End of Section6
############

#automation1()

automation2()
```