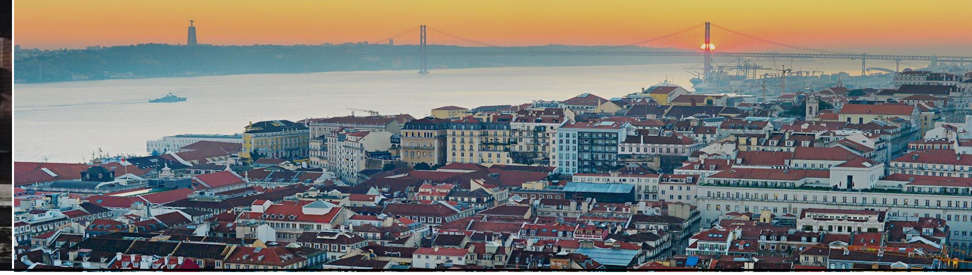



Michael Smurfit Business School
Blackrock, June - July 2020

GRASP & PJS Heuristic TOP

Prof. Dr. Angel A. Juan
ajuanp@gmail.com | http://ajuanp.wordpress.com
IN3 - Computer Science Dept., UOC, Barcelona, Spain

UOC R&I
IN3 Internet Interdisciplinary Institute

UNIVERSIDADE ABERTA
www.uab.pt

UCD DUBLIN
University College Dublin
Ireland's Global University

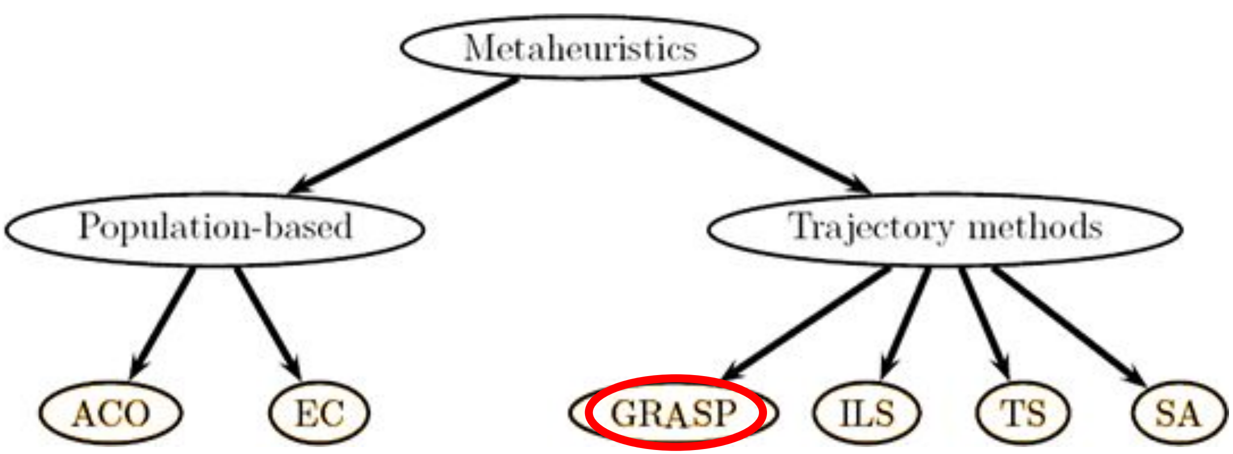
Overview

- Part I: GRASP Basic Concepts
- Part II: GRASP (TSP) w. Python
- Part III: PJ Savings Heuristic (TOP) w. Python
- References

python™

SPYDER

The Scientific Python Development Environment

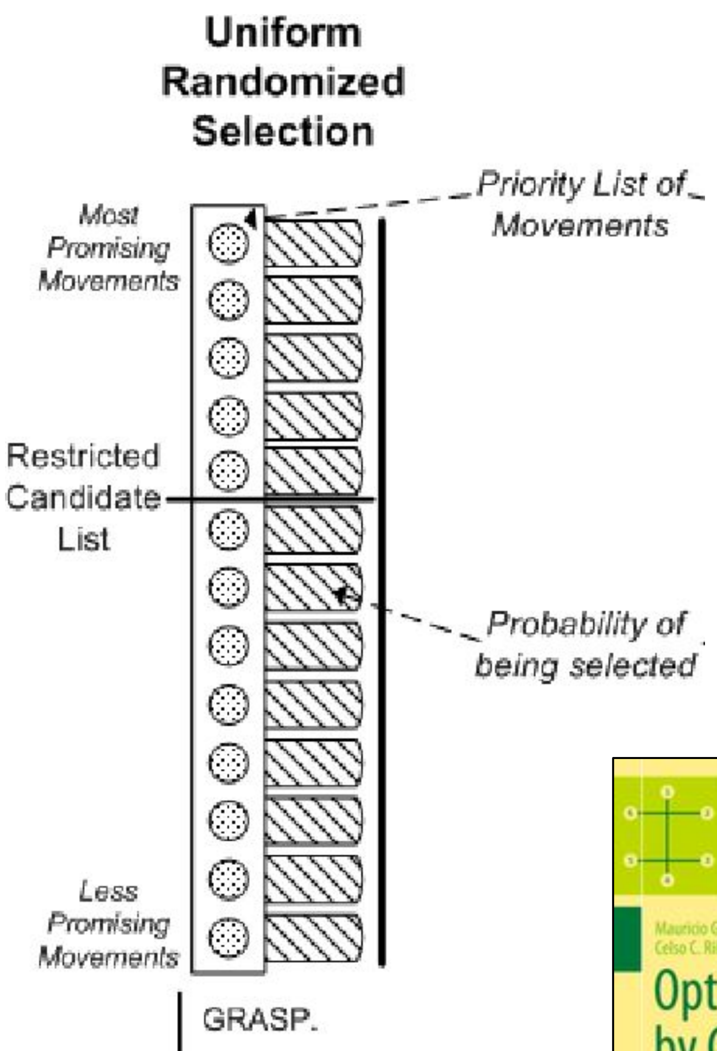


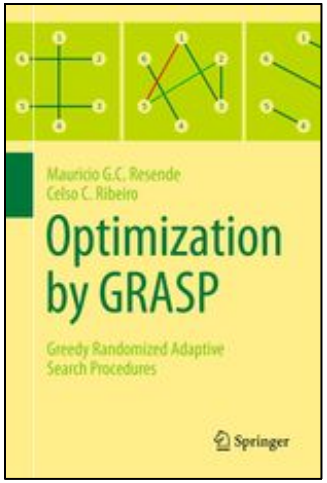
Part I:

GRASP Basic Concepts

GRASP Basic Concepts

- Greedy Randomized Adaptive Search or GRASP is a metaheuristic and a global optimization algorithm.
- The strategy is to iteratively sample stochastically greedy solutions and then use a local search heuristic to refine them to a local optima (Festa 2002).
- It builds a Restricted Candidate List (RCL) that constrains the features of a solution that may be selected from each cycle.
- The RCL may be constrained by an explicit size, or by using a factor [0, 1] on the cost of adding each feature to the current candidate solution.





Pseudocode of a Generic GRASP (1/3)

```
Procedure GRASP (MAX_ITERATIONS, SEED)

Best_Solution = 0;
Read_Input ( );
for k = 1,2,…, MAX_ITERATIONS do
    Solution = GreedyRandomizedConstruction (SEED);
    Solution = LocalSearch (Solution);
    if (Solution is better than Best_Solution) then
        UpdateSolution (Solution, Best_Solution);
    endif
endfor
return (Best_Solution);
end GRASP
```

Figure 1. Pseudocode of a generic GRASP

Greedy Randomized Construction + Local Search

Source: Festa (2002)

Pseudocode of a Generic GRASP (2/3)

Restricted Candidate List (RCL)

Procedure GreedyRandomizedConstruction (SEED)

Solution = 0;
Sort the candidate elements according to their incremantal costs;
while Solution is not complete **do**
    Build the Restricted Candidate List (RCL);
    Select from RCL an element v at random;
    Solution = Solution $\cup$ {v};
    Resort the candidate elements according to their incremental costs;
endwhile
return (Solution);
end GreedyRandomizedConstruction

Figure 2. Pseudocode of a generic GRASP construction phase

Source: Festa (2002)

Pseudocode of a Generic GRASP (3/3)

Procedure LocalSearch (Solution)

while Solution is not locally optimal **do**
    Find $s' \in N$ such that $f(s') \leq f(\text{Solution})$;
    Solution = s';
endwhile
return (Solution);
end LocalSearch

Figure 3. Pseudocode of a generic local search phase

Generic Local Search Process

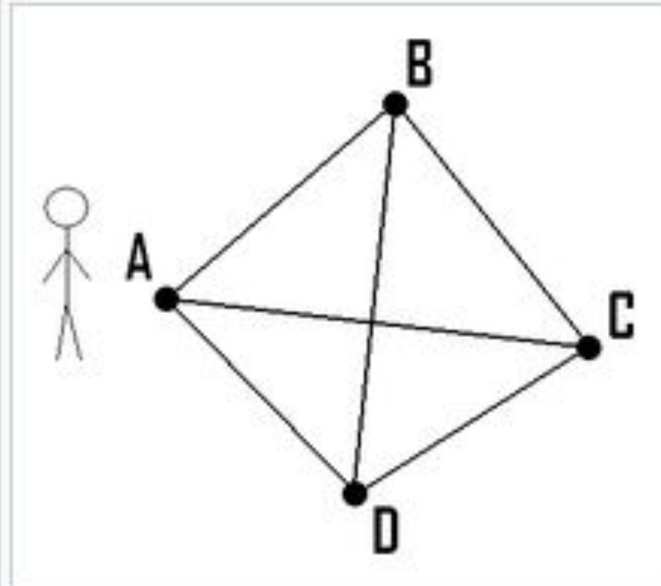
Source: Festa (2002)

Part II:

GRASP (TSP) w. Python

The Traveling Salesman Problem (TSP)

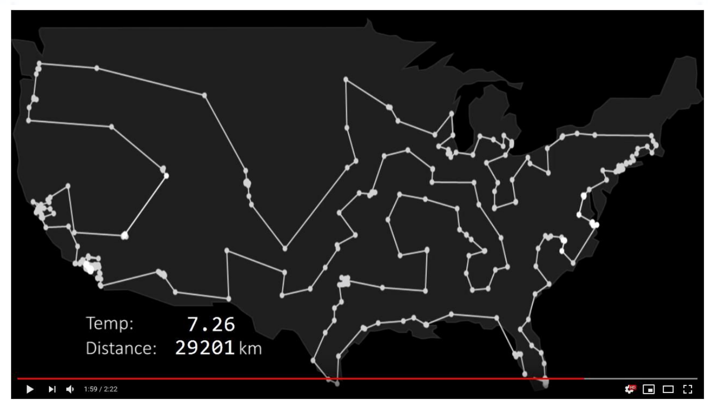
- The Traveling Salesman Problem (TSP) is a classic algorithmic problem in the field of Computer Science and Operations Research.
- The goal is to find, for a finite set of points whose pairwise distances are known, the shortest route connecting all of them.
- A solution for the TSP is a permutation of the nodes (order in which they are visited). For n cities you have (n-1)! possibilities. The TSP is an NP-hard problem.
- The insertion rule (first go from the starting point to the closest point, then to the point closest to this, etc.), does not usually yield the shortest route.



A salesman wants to visit all cities,A, B, C and D. What is the best way to do this (cheapest airline tickets, and minimal travel time)?

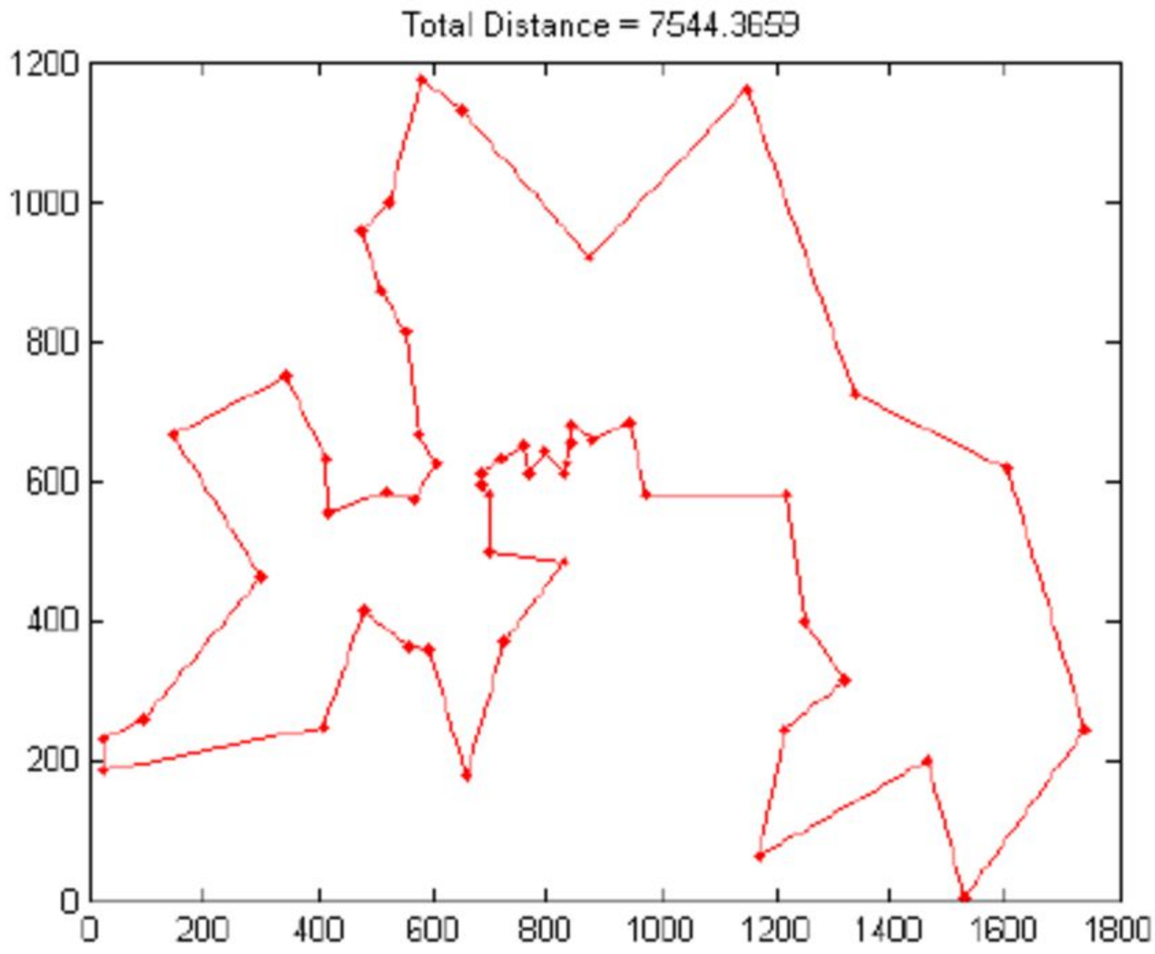
YouTube

https://www.youtube.com/watch?v=SC5CX8drAtU



The Berlin52 instance for the TSP

Optimal value for Belin52 is 7544.37.



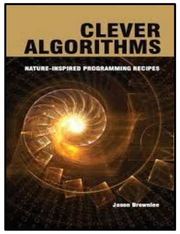
Many more TSP instances available at: https://www.coin-or.org/

GRASP_TSP.py   Algorithm Framework

```python
''' ALGORITHM FRAMEWORK '''
algorithmName = "GRASP"
print("Best Sol by " + algorithmName + "...")
# Problem configuration
inputsTSP = berlin52
maxIterations = 100
maxNoImprove = 50
greedinessFactor = 0.3 # In the range [0,1]. 0 is more greedy and 1 less
start = time.clock()
# Main loop
bestCost = float("inf") # infinity
while maxIterations > 0:
    maxIterations -= 1
    # Construct a Greedy solution
    newSol, newCost = constructGreedySolution(inputsTSP, greedinessFactor)
    # refine it using a local search heuristic
    newSol, newCost = localSearch(newSol, newCost, maxNoImprove)
    if newCost < bestCost:
        bestSol = newSol
        bestCost = newCost
        print("Cost = %.2f ; Iter = %d" % (bestCost, maxIterations))
# Stop clock and return outputs
stop = time.clock()
print("BestCost = %.2f ; Elapsed = %.2fs " % (bestCost, stop - start))
print("BestSol = %s " % bestSol)
```

GRASP_TSP.py   Local Search

```
from Shared import berlin52, stochasticTwoOpt, tourCost, euclideanDistance
import random, time


''' Aux Funct to Apply a Local Search '''
def localSearch(aSol, aCost, maxIter):
    count = 0
    while count < maxIter:
        newSol = stochasticTwoOpt(aSol)
        newCost = tourCost(newSol)
        if newCost < aCost: # Restart the search when we find an imporvement
            aSol = newSol
            aCost = newCost
            count = 0
        else:
            count += 1
    # return solution and cost
    return aSol, aCost
```



Python code on GRASP is based on the ones by Jason Brownlee (http://www.cleveralgorithms.com/) and Sain Panyam (https://www.saipanyam.net/2011/06/clever-algorithms-python.html)

GRASP_TSP.py Greedy Solution

```python
''' Aux Funct to Construct a Greedy Solution '''
def constructGreedySolution(perm, alpha):
    # Select one node randomly and incorporate it to the emerging sol
    emergingSol = [] # permutation (list) of nodes
    problemSize = len(perm)
    emergingSol = [perm[random.randrange(0, problemSize)]]
    # While sol size is not equal to the original permutation size
    while len(emergingSol) < problemSize:
        # Get all nodes not already in the emerging sol
        notInSolNodes = [node for node in perm if node not in emergingSol]
        # For each node not in emergingSol, compute distance w.r.t. last element
        costs = []
        emergingSolSize = len(emergingSol)
        for node in notInSolNodes:
            costs.append(euclideanDistance(emergingSol[emergingSolSize-1], node))
        # Determining the max cost and min cost from the feature set
        maxCost, minCost = max(costs), min(costs)
        # Build the RCL by adding the nodes satisfying the condition
        rcl = []
        for index, cost in enumerate(costs): # get both the index and the item
            if cost <= minCost + alpha * (maxCost-minCost):
                # Add it to the RCL
                rcl.append(notInSolNodes[index])
        # Select random feature from RCL and add it to the solution
        emergingSol.append(rcl[random.randrange(0, len(rcl))])
    # calculate the final tour cost before returning the new solution
    newCost = tourCost(emergingSol)
    # return solution and cost
    return emergingSol, newCost
```

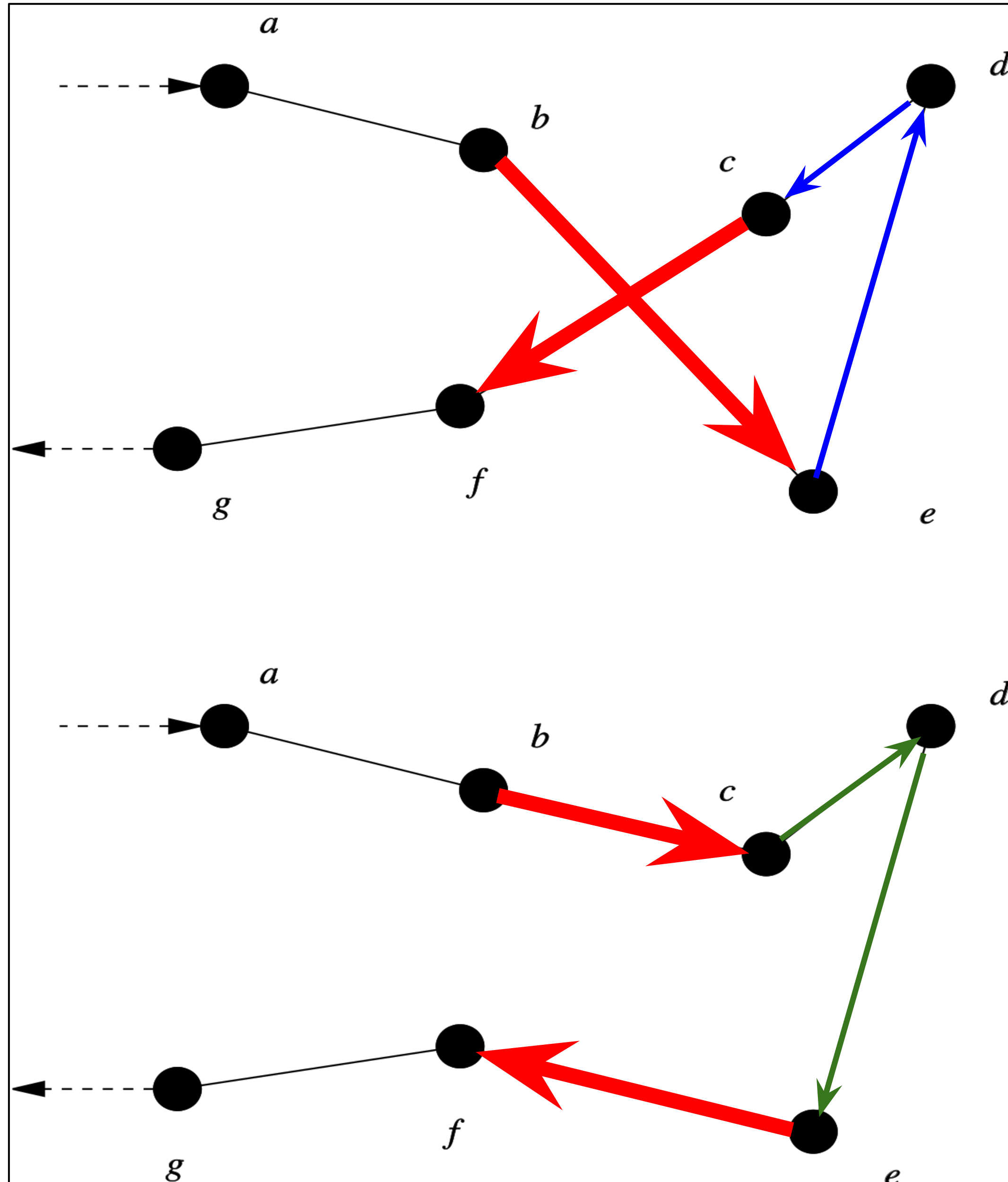
Shared.py Berlin52 Data

```python
"""
* Common variables and functions that are used by different algorithms
* Code based on 'Clever Algorithms' by Jason Brownlee.
"""

import math, random
```

```python
# Input data (nodes) for the TSP
# The optimal solution (using real numbers for distances) is 7544.3659 according to:
# https://www.researchgate.net/figure/The-optimal-solution-of-Berlin52_fig2_221901574
berlin52 = [[565,575],[25,185],[345,750],[945,685],[845,655],
            [880,660],[25,230],[525,1000],[580,1175],[650,1130],[1605,620],
            [1220,580],[1465,200],[1530,5],[845,680],[725,370],[145,665],
            [415,635],[510,875],[560,365],[300,465],[520,585],[480,415],
            [835,625],[975,580],[1215,245],[1320,315],[1250,400],[660,180],
            [410,250],[420,555],[575,665],[1150,1160],[700,580],[685,595],
            [685,610],[770,610],[795,645],[720,635],[760,650],[475,960],
            [95,260],[875,920],[700,500],[555,815],[830,485],[1170,65],
            [830,610],[605,625],[595,360],[1340,725],[1740,245]]
```

2-Opt Operator for Local Search

How a 2-opt operator works:
1. Select two non-consecutive edges (b,e) and (c,f).
2. Swap (interchange) them to obtain edges (b,c) and (e,f).
3. Reverse the edges between them to complete the tour.



Shared.py Stochastic 2-Opt

```python
# Deletes two edges and reverses the sequence in between the deleted edges
def stochasticTwoOpt(perm):
    result = perm[:] # to avoid chaning the original sol (perm), make a copy
    size = len(result)
    # select indices of two random points in the tour
    p1, p2 = random.randrange(0,size), random.randrange(0,size)
    # do this so as not to overshoot tour boundaries
    exclude = set([p1])
    if p1 == 0:
        exclude.add(size-1)
    else:
        exclude.add(p1-1)

    if p1 == size-1:
        exclude.add(0)
    else:
        exclude.add(p1+1)

    while p2 in exclude:
        p2 = random.randrange(0,size)

    # to ensure we always have p1<p2
    if p2 < p1:
        p1, p2 = p2, p1

    # now reverse the tour segment between p1 and p2
    result[p1:p2] = reversed(result[p1:p2])

    return result
```

This code guarantees that p2 is different from p1 and from any of its two adjacent nodes.

Shared.py   Tour Cost & Euclidean Distance

```python
# Evaluates the total length of a TSP solution (permutation of nodes)
def tourCost(perm):
    # Is the sum of the euclidean distance between consecutive points in the path
    totalDistance = 0.0
    size = len(perm)
    for index in range(size):
        startNode = perm[index]
        # select the end point point for calculating the segment length
        if index == size-1:
            # In order to complete the 'tour' we need to reach the starting point
            endNode = perm[0]
        else: # select the next point
            endNode = perm[index+1]

        totalDistance +=  euclideanDistance(startNode, endNode)
    return totalDistance

# Calculates the euclidean distance between two points
def euclideanDistance(xNode, yNode):
    sum = 0.0
    # use Zip to iterate over the two vectors (nodes)
    for xi, yi in zip(xNode, yNode):
        sum += pow((xi-yi), 2)
    return math.sqrt(sum)
```

GRASP_TSP.py   Results after 1,000 iter (1 run)

IPython console

Console 1/A

```
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Restarting kernel...

 /usr/lib/python3/dist-packages/traitlets/config/configurable.py:84: UserWarning: Config option `use_jedi` not
recognized by `IPCompleter`.
  self.config = config

In [1]: runfile('/home/aajp/Documents/CleverAlgorithms/GRASP_TSP.py', wdir='/home/aajp/Documents/
CleverAlgorithms')
Best Sol by GRASP...
Cost = 12188.60 ; Iter = 999
Cost = 11757.08 ; Iter = 997
Cost = 10119.38 ; Iter = 996
Cost = 9966.61 ; Iter = 960
Cost = 9380.00 ; Iter = 670
Cost = 9238.10 ; Iter = 408
BestCost = 9238.10 ; Elapsed = 35.13s
BestSol = [[475, 960], [525, 1000], [580, 1175], [650, 1130], [875, 920], [1150, 1160], [1340, 725], [1605,
620], [1740, 245], [1530, 5], [1215, 245], [1170, 65], [1465, 200], [1320, 315], [1250, 400], [1220, 580],
[945, 685], [975, 580], [880, 660], [830, 610], [845, 680], [845, 655], [835, 625], [795, 645], [700, 500],
[700, 580], [770, 610], [830, 485], [760, 650], [720, 635], [685, 610], [685, 595], [560, 365], [595, 360],
[725, 370], [660, 180], [410, 250], [480, 415], [565, 575], [520, 585], [555, 815], [575, 665], [605, 625],
[510, 875], [415, 635], [420, 555], [300, 465], [95, 260], [25, 230], [25, 185], [145, 665], [345, 750]]
```

Optimal value for Belin52 is 7544.37.

This is a probabilistic algorithm --> using more iterations (or more runs in parallel) will help to get better solutions.

References

Ferone, D., Gruler, A., Festa, P., & Juan, A. A. (2019). Enhancing and extending the classical GRASP framework with biased randomisation and simulation. Journal of the Operational Research Society, 70(8), 1362-1375.

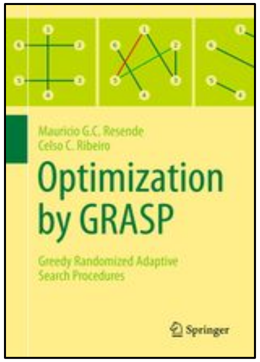
Festa, P. (2002): Greedy Randomized Adaptive Search Procedures. AIRO News, 7-11.

Festa, P., & Resende, M. G. (2009). An annotated bibliography of GRASP–Part I: Algorithms. International Transactions in Operational Research, 16(1), 1-24.

Festa, P., & Resende, M. G. (2009). An annotated bibliography of GRASP–Part II: Applications. International Transactions in Operational Research, 16(2), 131-172.

Juan, A. A., Faulin, J., Ferrer, A., Lourenço, H. R., & Barrios, B. (2013). MIRHA: multi-start biased randomization of heuristics with adaptive local search for solving non-smooth routing problems. Top, 21(1), 109-132.

Resende, M. G., & Ribeiro, C. C. (2016). Optimization by GRASP. Springer Science + Business Media New York.



https://www.dropbox.com/sh/qfk6i2858v8dz2w/AADCeHaLPIEpFOQJPdfQNbwha?dl=0

Homework Activities

1. Read one **GRASP-related article** and write a brief summary on it. Assign a score between 0 and 10.

2. Construct your own **Python program** to implement a GRASP algorithm for solving the TSP.

3. (Optional) Explain the ‘hidden’ logic behind the `stochasticTwoOpt()` function.

4. (Optional) Try to improve your GRASP program by **adjusting the parameters** or introducing some enhancements. Write a short report on it.

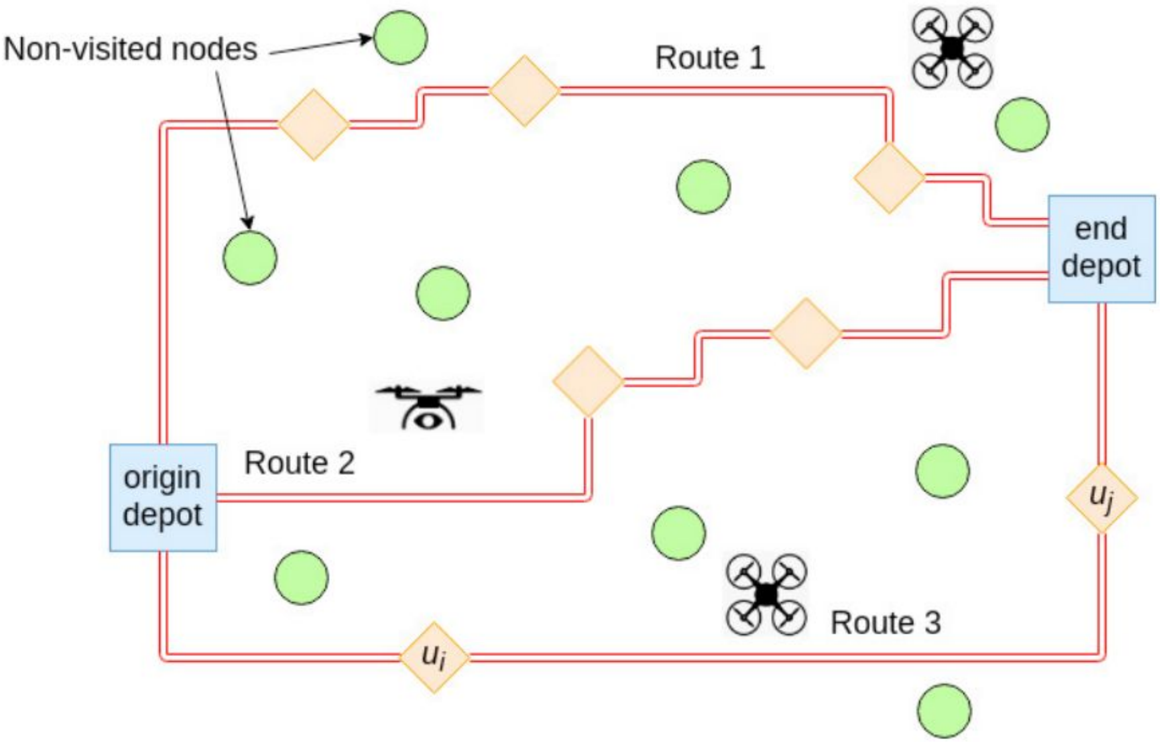
5. (Optional) Complete computational experiments on other **TSP instances** in http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html. Write a short report.

Part III:

PJ Savings Heuristic (TOP) w. Python

The Team Orienteering Problem (TOP)

- **The Team Orienteering Problem (TOP) is a well-known NP-hard problem:**
  - **Start and finish depots, customers' rewards *u(i)*, limited fleet of vehicles, etc.**
  - **Travel costs *c(i, j)***
  - **Constraints: max. cost / time per rute, etc.**



- **Goal: select customers to visit and define routes that maxime the collected utility.**
- **Different applications include: self-driving vehicles, unmanned aerial vehicles, ride sharing, etc.**

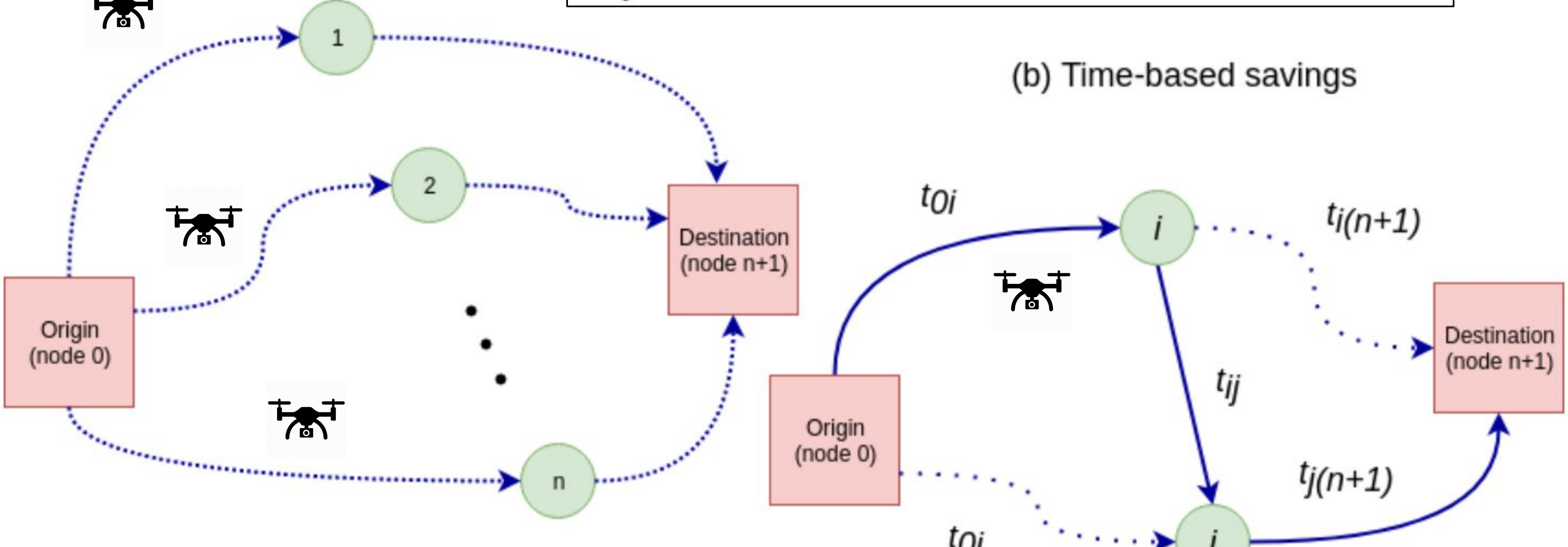
The PJS Heuristic for the TOP

Juan, A. A., Freixes, A., Panadero, J., Serrat, C., & Estrada, A. (2020). Routing Drones in Smart Cities: a Biased-Randomized Algorithm for Solving the Team Orienteering Problem in Real Time. Transportation Research Procedia, 47, 243-250.

Reyes, L. , Ospina, C., Faulin, J., Mozos, J., Panadero, J., & Juan, A. A. (2018). The team orienteering problem with stochastic service times and driving-range limitations. In 2018 Winter Simulation Conference (pp. 3025-3035). IEEE.

Efficiency (or enriched savings) value: consider a linear combination of classical savings and collected utilities associated with an edge. In order to this linear combination to make sense, both quantities should be in the same order of magnitude.

$$s'_{ij} = \alpha \cdot s_{ij} + (1-\alpha) \cdot (u_i + u_j)$$



(b) Time-based savings

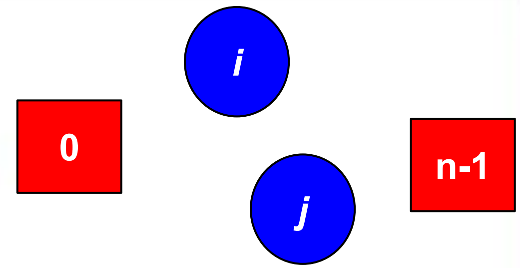
(a) Dummy solution

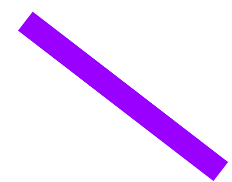
routing_objects.py Classes Node and Edge

routing_objects.py | vrp_cws_heuristic.py | top_pjs_heuristic.py

```python
class Node:

    def __init__(self, ID, x, y, demand):
        self.ID = ID # node identifier (depot ID = 0)
        self.x = x # Euclidean x-coordinate
        self.y = y # Euclidean y-coordinate
        self.demand = demand # demand (is 0 for depot and positive for others)
        self.inRoute = None # route to which node belongs
        self.isInterior = False # an interior node is not connected to depot
        self.dnEdge = None # edge (arc) from depot to this node
        self.ndEdge = None # edge (arc) from this node to depot
        self.isLinkedToStart = False # linked to start depot?
        self.isLindedToFinish = False # linked to finish depot?


class Edge:

    def __init__(self, origin, end):
        self.origin = origin # origin node of the edge (arc)
        self.end = end # end node of the edge (arc)
        self.cost = 0.0 # edge cost
        self.savings = 0.0 # edge savings (Clarke & Wright)
        self.invEdge = None # inverse edge (arc)
        self.efficiency = 0.0 # edge efficiency (enriched savings)
```

We can use the demand field in a node to save the reward.

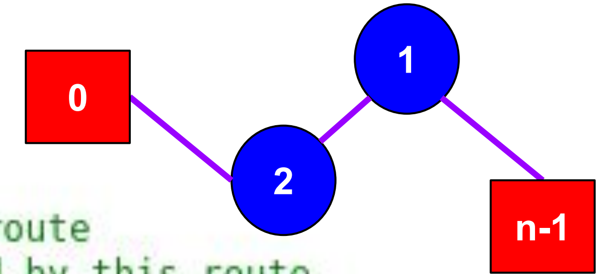




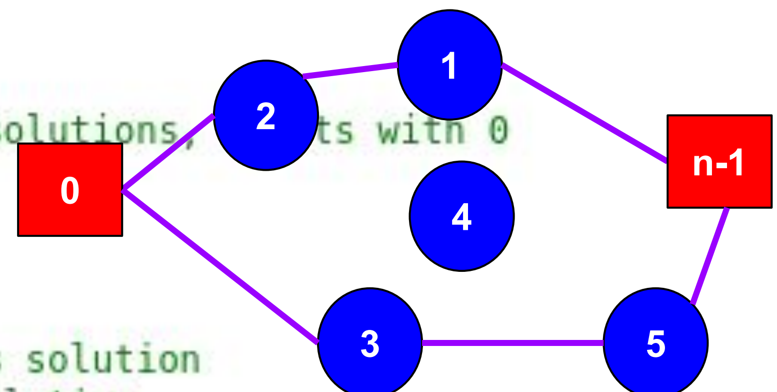
spyder

routing_objects.py Classes Route and Solution

```python
class Route:

    def __init__(self):
        self.cost = 0.0 # cost of this route
        self.edges = [] # sorted edges in this route
        self.demand = 0.0 # total demand covered by this route

    def reverse(self): # e.g. 0 -> 2 -> 6 -> 0 becomes 0 -> 6 -> 2 -> 0
        size = len(self.edges)
        for i in range(size):
            edge = self.edges[i]
            invEdge = edge.invEdge
            self.edges.remove(edge)
            self.edges.insert(0, invEdge)


class Solution:

    last_ID = -1 # counts the number of solutions, ts with 0

    def __init__(self):
        Solution.last_ID += 1
        self.ID = Solution.last_ID
        self.routes = [] # routes in this solution
        self.cost = 0.0 # cost of this solution
        self.demand = 0.0 # total demand covered by this solution
```



Since in the TOP we are using arcs (not edges), we'll not need the reverse method.



We can use the demand field in a route to save the reward.

pjs_heuristic.py   Reading an Instance Data

```python
""" PANADERO & JUAN SAVINGS HEURISTIC FOR THE TEAM ORIENTEERING PROBLEM (TOP) """

import networkx as nx
from routing_objects import Node, Edge, Route, Solution
import math
import operator

""" Set algorithm parameters """

# alpha is used to compute the edge efficiency (enriched savings), its best value
# might depend on the specific instance as explained in Panadero et al. (2020)
alpha = 0.7


""" Read instance data from txt file """

instanceName = 'p5.3.q' # name of the instance
# txt file with the TOP instance data
fileName = 'data/' + instanceName + '.txt'


with open(fileName) as instance:
    i = -3 # we start at -3 so that the first node is node 0
    nodes = []
    for line in instance:
        if i == -3: pass # line 0 contains the number of nodes, not needed
        elif i == -2: fleetSize = int( line.split(';')[1] )
        elif i == -1: routeMaxCost = float( line.split(';')[1] )
        else:
            # array data with node data: x, y, demand (reward in TOP)
            data = [float(x) for x in line.split(';')]
            aNode = Node(i, data[0], data[1], data[2])
            nodes.append(aNode)
        i += 1
```

Test different values of alpha: 0.1, 0.2, …, 0.9

Data on instances is available at: https://www.dropbox.com/sh/uwbixk6iuvxdozg/AADngjZHg765Qd0IDj1kRGSaa?dl=0

pjs_heuristic.py   Creating the Edges and List

```python
startTime = time.time()
""" Construct edges with costs and efficiency list from nodes """

start = nodes[0] # first node is the start depot
finish = nodes[-1] # last node is the finish depot

for node in nodes[1:-1]: # excludes both depots
    snEdge = Edge(start, node) # creates the (start, node) edge (arc)
    nfEdge = Edge(node, finish)
    # compute the Euclidean distance as cost
    snEdge.cost = math.sqrt((node.x - start.x)**2 + (node.y - start.y)**2)
    nfEdge.cost = math.sqrt((node.x - finish.x)**2 + (node.y - finish.y)**2)
    # save in node a reference to the (depot, node) edge (arc)
    node.dnEdge = snEdge
    node.ndEdge = nfEdge

efficiencyList = []
for i in range(1, len(nodes) - 2): # excludes the start and finish depots
    iNode = nodes[i]
    for j in range(i + 1, len(nodes) - 1):
        jNode = nodes[j]
        ijEdge = Edge(iNode, jNode) # creates the (i, j) edge
        jiEdge = Edge(jNode, iNode)
        ijEdge.invEdge = jiEdge # sets the inverse edge (arc)
        jiEdge.invEdge = ijEdge
        # compute the Euclidean distance as cost
        ijEdge.cost = math.sqrt((jNode.x - iNode.x)**2 + (jNode.y - iNode.y)**2)
        jiEdge.cost = ijEdge.cost # assume symmetric costs
        # compute efficiency as proposed by Panadero et al.(2020)
        ijSavings = iNode.ndEdge.cost + jNode.dnEdge.cost - ijEdge.cost
        edgeReward = iNode.demand + jNode.demand
        ijEdge.savings = ijSavings
        ijEdge.efficiency = alpha * ijSavings + (1 - alpha) * edgeReward
        jiSavings = jNode.ndEdge.cost + iNode.dnEdge.cost - jiEdge.cost
        jiEdge.savings = jiSavings
        jiEdge.efficiency = alpha * jiSavings + (1 - alpha) * edgeReward
        # save both edges in the efficiency list
        efficiencyList.append(ijEdge)
        efficiencyList.append(jiEdge)

# sort the list of edges from higher to lower efficiency
efficiencyList.sort(key = operator.attrgetter("efficiency"), reverse = True)
```

spyder

pjs_heuristic.py   Dummy Sol and Aux. Funct.

```
    """ Construct the dummy solution """

    sol = Solution()
    for node in nodes[1:-1]: # excludes the start and finish depots
        snEdge = node.dnEdge # get the (start, node) edge
        nfEdge = node.ndEdge # get the (node, finish) edge
        snfRoute = Route() # construct the route (start, node, finish)
        snfRoute.edges.append(snEdge)
        snfRoute.demand += node.demand
        snfRoute.cost += snEdge.cost
        snfRoute.edges.append(nfEdge)
        snfRoute.cost += nfEdge.cost
        node.inRoute = snfRoute # save in node a reference to its current route
        node.isLinkedToStart = True # this node is currently linked to start depot
        node.isLinkedToFinish = True # this node is currently linked to finish depot
        sol.routes.append(snfRoute) # add this route to the solution
        sol.cost += snfRoute.cost
        sol.demand += snfRoute.demand # total reward in route




    """ Perform the edge-selection & routing-merging iterative process """

    def checkMergingConditions(iNode, jNode, iRoute, jRoute, ijEdge):
        # condition 1: iRoute and jRoure are not the same route object
        if iRoute == jRoute: return False
        # condition 2: jNode has to be linked to start and i node to finish
        if iNode.isLinkedToFinish == False or jNode.isLinkedToStart == False: return False
        # condition 3: cost after merging does not exceed maxTime (or maxCost)
        if routeMaxCost < iRoute.cost + jRoute.cost - ijEdge.savings: return False
        # else, merging is feasible
        return True
```

pjs_heuristic.py   Iterative Merging Process

```python
while len(efficiencyList) > 0: # list is not empty
    index = 0 # greedy behavior
    ijEdge = efficiencyList.pop(index) # select the next edge from the list
    # determine the nodes i < j that define the edge
    iNode = ijEdge.origin
    jNode = ijEdge.end
    # determine the routes associated with each node
    iRoute = iNode.inRoute
    jRoute = jNode.inRoute
    # check if merge is possible
    isMergeFeasible = checkMergingConditions(iNode, jNode, iRoute, jRoute, ijEdge)
    # if all necessary conditions are satisfied, merge and delete edge (j, i)
    if isMergeFeasible == True:
        # if still in list, delete edge (j, i) since it will not be used
        jiEdge = ijEdge.invEdge
        if jiEdge in efficiencyList: efficiencyList.remove(jiEdge)
        # iRoute will contain edge (i, finish)
        iEdge = iRoute.edges[-1] # iEdge is (i, finish)
        # remove iEdge from iRoute and update iRoute cost
        iRoute.edges.remove(iEdge)
        iRoute.cost -= iEdge.cost
        # node i will not be linked to finish depot anymore
        iNode.isLinkedToFinish = False
        # jRoute will contain edge (start, j)
        jEdge = jRoute.edges[0]
        # remove jEdge from jRoute and update jRoute cost
        jRoute.edges.remove(jEdge)
        jRoute.cost -= jEdge.cost
        # node j will not be linked to start depot anymore
        jNode.isLinkedToStart = False
        # add ijEdge to iRoute
        iRoute.edges.append(ijEdge)
        iRoute.cost += ijEdge.cost
        iRoute.demand += jNode.demand
        jNode.inRoute = iRoute
        # add jRoute to new iRoute
        for edge in jRoute.edges:
            iRoute.edges.append(edge)
            iRoute.cost += edge.cost
            iRoute.demand += edge.end.demand
            edge.end.inRoute = iRoute
        # delete jRoute from emerging solution
        sol.cost -= ijEdge.savings
        sol.routes.remove(jRoute)
```

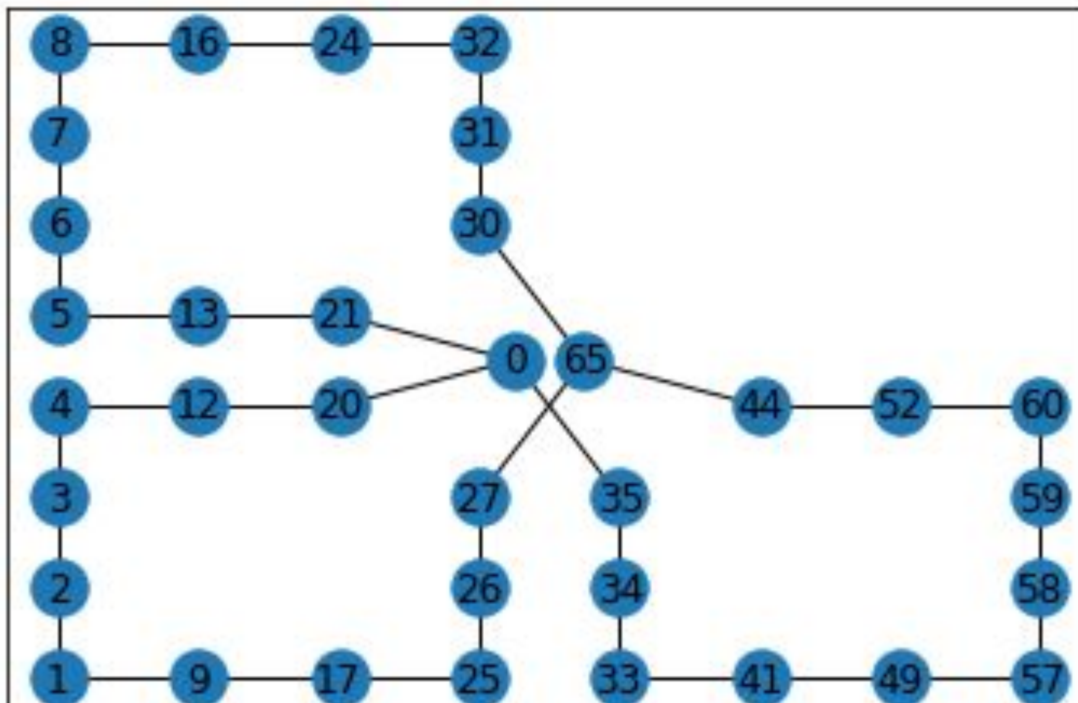
pjs_heuristic.py   Selection of Routes and Plot

```python
# sort the list of routes in sol by demand (reward) and delete extra routes
sol.routes.sort(key = operator.attrgetter("demand"), reverse = True)
for route in sol.routes[fleetSize:]:
    sol.demand -= route.demand # update reward
    sol.cost -= route.cost # update cost
    sol.routes.remove(route) # delete extra route

endTime = time.time()

''' Print the PJS Solution '''

print('Instance: ', instanceName)
print('Reward obtained with PJS heuristic sol =', "{:.{}f}".format(sol.demand, 2))
print('Computational time:', "{:.{}f}".format(endTime - startTime, 2), 'sec.')
for route in sol.routes:
    s = str(0)
    for edge in route.edges:
        s = s + '-' + str(edge.end.ID)
    print('Route: ' + s + ' || Reward = ' + "{:.{}f}".format(route.demand, 2)
          + ' || Cost / Time = ' + "{:.{}f}".format(route.cost, 2))


# Plot the solution

G = nx.Graph()
G.add_node(start.ID, coord=(start.x, start.y))
for route in sol.routes:
    for edge in route.edges:
        G.add_edge(edge.origin.ID, edge.end.ID)
        G.add_node(edge.end.ID, coord = (edge.end.x, edge.end.y))
coord = nx.get_node_attributes(G, 'coord')
nx.draw_networkx(G, coord, node_color = 'pink')
```

pjs_heuristic.py   Results

```
Console 1/A

Python 3.7.6 (default, Jan  8 2020, 19:59:22)
Type "copyright", "credits" or "license" for more information.

IPython 7.12.0 -- An enhanced Interactive Python.

In [1]: runfile('/home/aajp/2020_VRP_TOP_savings_heuristic_Python/top_pjs_heuristic.py',
aajp/2020_VRP_TOP_savings_heuristic_Python')
Instance:  p5.3.q
Reward obtained with PJS heuristic sol = 975.00
Route: 0-20-12-4-3-2-1-9-17-25-26-27-65 || Reward = 325.00 || Cost / Time = 26.05
Route: 0-21-13-5-6-7-8-16-24-32-31-30-65 || Reward = 325.00 || Cost / Time = 26.05
Route: 0-35-34-33-41-49-57-58-59-60-52-44-65 || Reward = 325.00 || Cost / Time = 26.05
```

This is the result for alpha = 0.7, you should test other values as well. Also, encapsulating the heuristic into a multi-start biased-randomized algorithm will noticeably improve the best-found solution.



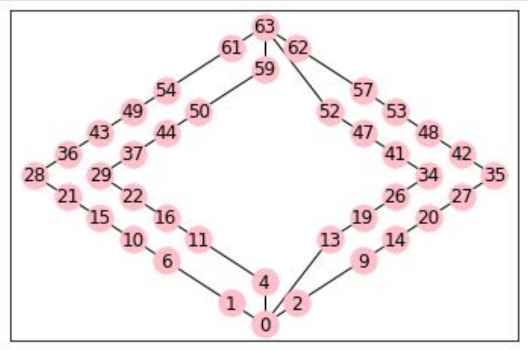
Homework Activities

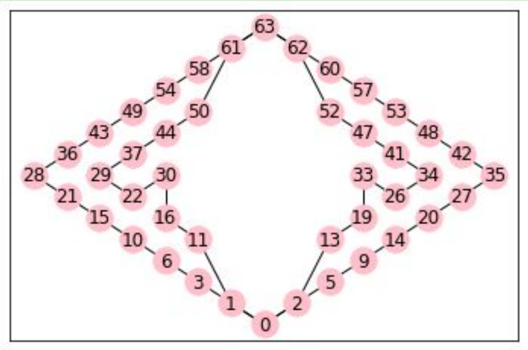
1. Construct your own Python program to implement the PJS heuristic for solving the TOP and test it in different instances.

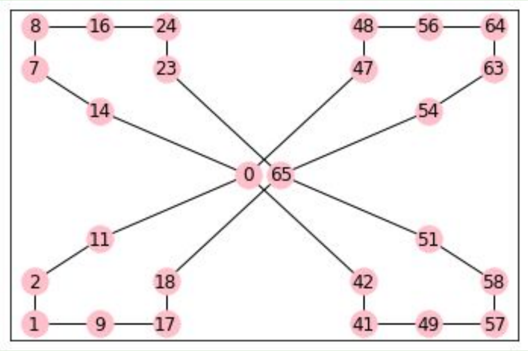
2. Complete a data analysis on the results for the different instances tested.

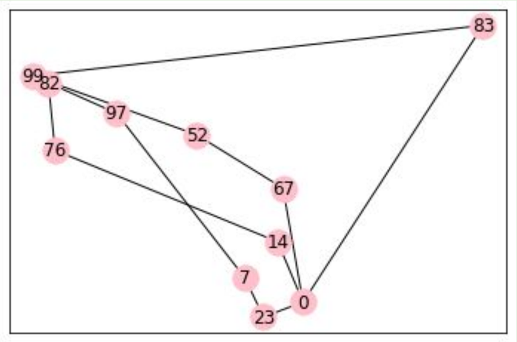
3. (Optional) Combine GRASP concepts with the PJS heuristic and analyze the results.

4. (Optional) Try to enhance the GRASP-PJS algorithm by using biased randomization concepts.









References

Barry, P. (2016). Head First Python: A Brain-Friendly Guide. O'Reilly Media, Inc.

Brownlee, J. (2011). Clever algorithms: nature-inspired programming recipes. Jason Brownlee.

Downey, A. B. (2015): Think Python: How to Think Like a Computer Scientist. O'Reilly Media

Johnson, M. J. (2018). A concise introduction to programming in Python. CRC Press.

Luke, S. (2009). Essentials of Metaheuristics. Raleigh: Lulu.

Panyam, S. (2011). Clever Algorithms in Python.

Panadero, J., Currie, C., Juan, A. A., Bayliss, C. (2020): Maximizing Reward from a Team of Surveillance Drones under Uncertainty Conditions: a simheuristic approach. European Journal of Industrial Engineering.

Talbi, E. (2009): Metaheuristics: From Design to Implementation. John Wiley & Sons.