me-lab2

October 24, 2019

1 Metaheuristic Algorithms - Lab 2

1.1 Objectives

- Implementation of some trajectory based local and global search: Simulated Annealing, Tabu Search, Nelder Mead and Pattern Search
- Applications in combinatorial optimization (TSP and knapsack problem) and continuous optimization (traditional benchmark functions)

1.2 Combinatorial optimization problems

The search space of combinatorial optimization problems is usually finite but of large size. Thus an exhaustive search space exploration is inapplicable.

Two well-known combinatorial optimization problems, which have several practical applications are: * Travelling salesman problem (TSP) * Knapsack problem

1.2.1 Travelling Salesman Problem

TSP is a well known combinatorial optimization problem asking to find the optimal route for a salesman who has to visit a set of n towns. It is a constrained optimization problem characterized by: * Constraints: the salesman visits each town exactly once * Objective function: the cost of the tour should be minimized

The classical TSP is equivalent with the problem of finding an optimal Hamiltonian tour (a tour which visits exactly once each node and has the smallest cost) in a complete graph (there is an edge between any two nodes). TSP can be solved exactly for small values of n but, since the number of possible tours is $(n-1)!/2$, for large values of n there are no efficient exact methods. TSP belongs to the class of NP-complete problems.

There are several variants of the problem: * Asymmetric TSP: the cost of the connection between two nodes depends on the tour orientation. * Sequential Ordering Problem – SOP: there are additional constraints specifying that a given node should be visited before another one. * Capacitated vehicle routing problem – CVRP: find optimal tours for a set of trucks which have to transport products from a warehouse to different customers. The trucks have all the same capacity. * Generalized TSP: the nodes correspond to clusters of locations and there are several arcs between nodes. TSP is important not only from a theoretical point of view but also from a practical point of view since there are several real-world problems which can be formulated as a TSP: * Vehicle Routing Problem (VRP): find the optimal route for vehicles * Control of drilling machines which are used to construct boards for integrated circuits * Find shortest routes through selections of airports in the world * Reconstruct DNA sequences starting from subsequences (genome assembling)

Other applications are listed at [http://www.tsp.gatech.edu/apps/index.html]

Besides exact methods, there exist a lot of heuristic methods based on incremental improvements of the current tour. One of the most used heuristics for TSP is the Lin-Kernighan heuristic which is based on replacing some arcs of the current tour with other ones such that the total cost becomes smaller. The simplest case is when just two arcs are replaced (2-opt transformation) which is equivalent with reversing the order of visiting the nodes belonging to a subtour.

Example: Let us consider 6 nodes: A, B, C, D, E, F. If the current tour is (A, C, B, E, F, D), by replacing the edge (A, C) with the edge (A, F), and the edge (F, D) with (C, D) and by reversing the order of visiting the nodes B and E one obtains the tour (A, F, E, B, C, D). It is easy to see that this transformation can be obtained directly by reversing the subtour (C, B, E, F).

Another perturbation heuristic for TSP is that based on 4 interchanges (double bridge transform) which transforms a route $[i1..i2]$ $[i3..i4]$ $[i5..i6]$ $[i7..i8]$ into $[i2..i1]$ $[i4..i3]$ $[i6..i5]$ $[i8..i7]$.

1.2.2 Knapsack Problem

The classical variant of the knapsack problem is: "Let us consider a set of n objects, each one being characterized by a given weight and a given value. Select a subset of objects such that the total size of the selected objects is smaller than a knapsack capacity and the total value of the selected objects is as large as possible."

The search space is represented by all possible subsets of the set of n objects, thus the search space size is 2^n.

Real world problems which can be formulated as the knapsack problem are:

- Financial portfolios construction (the aim being the maximization of the profit such that the amount of investment is lower than a given threshold).
- Resource allocation (the selection of some tasks which can use a given resource such that the resource is not overloaded and some gain is maximized).
- The selection of some products to be placed in a container or warehouse.

Variants of the problem: * Multi-criterial case: the aim is not only to maximize a value but optimize several criteria * Multi-dimensional case: the "size"/"weight" of an object is not specified by a single value but by multiple values * Multiple knapsacks: several knapsacks are used (this is related to the bin packing problem)

1.3 Simulated Annealing

1.3.1 General description

Simulated Annealing is a metaheuristic characterized by the fact that lower quality configurations may be accepted. The decision on the acceptance of such configurations is taken probabilistically, and the acceptance probability depends on a parameter called "temperature" (by analogy with the temperature of physical systems which are involved in a thermal process, e.g. annealing of alloys). The probability of accepting a lower quality configuration is higher if the temperature is higher.

General structure of Simulated Annealing:

S=initial configuration

T=initial value of the temperature

Repeat > > S′=perturb(S) > > If accept(S,S′,T) then S=S′ > > T=update(T) > Until &lt stopping condition &gt

The perturbation depends on the problem to be solved and the probability to accept the transition form a configuration S to a configuration S depends on the loss of quality (if the loss is small the acceptance probability is higher). An example of the implementation of an acceptance rule is (in the case of a minimization problem):

Accept(S,S′,T) >If rand(0,1)<exp(-(f(s′)-f(s))/T) then > >> Return True > >Else > >> Return False

1.3.2 Solving TSP by using "Simulated Annealing"

In order to solve a problem by using Simulated Annealing there are several elements to be established:

1. Solution encoding. The natural encoding variant for TSP is the permutation: a tour through n nodes can be described as a permutation of order n. This encoding ensures the satisfaction of the constraint of visiting only once each node.

Example: If the towns are numbered as follows: $1-A, 2-B, 3-C, 4-D, 5-E, 6-F$ then the route (A, C, B, E, F, D) corresponds to the permutation $(1, 3, 2, 5, 6, 4)$

2. Local search mechanism (construction of a new configuration starting from the existing one by perturbation) The simplest mechanism is based on a 2-opt transformation:

- Choose two random indices i and j such that $1 \leq i < j \leq n$
- Reverse the order of elements in the permutation having indices between i and j.

Example: If the current tour is described by the permutation $(1, 3, 2, 5, 6, 4)$ and $i = 2$, $j = 5$ then the new permutation will be: $(1, 6, 5, 2, 3, 4)$

3. Acceptance probability. The probability to accept a configuration S′ obtained from the configuration S can be computed by using the Boltzmann distribution:

$P(S|S) = min\{1, \exp(-(cost(S) - cost(S))/T(k)\}$
where $cost(S)$ is the cost of configuration S and $T(k)$ is a control parameter (temperature).

4. Cooling schedule. If $T(k)$ denotes the temperature corresponding to the iteration k then the value corresponding to the next iteration can be computed as follows:

- $T(k+1) = T(0)/log(k+c)$, c being a constant
- $T(k+1) = T(0)/k$
- $T(k+1) = aT(k)$, with a denoting a value smaller but close to 1 (e.g. $a = 0.99$)

Remark. The initial value of the temperature ($T(0)$) should be large enough to allow the transition between any two configurations.

- Stopping condition. The stopping criterion can be related to the value of the temperature (stop when the temperature is low enough), to the number of iterations (stop after a given number of iterations) or to the value of the objective function (cost of the tour).

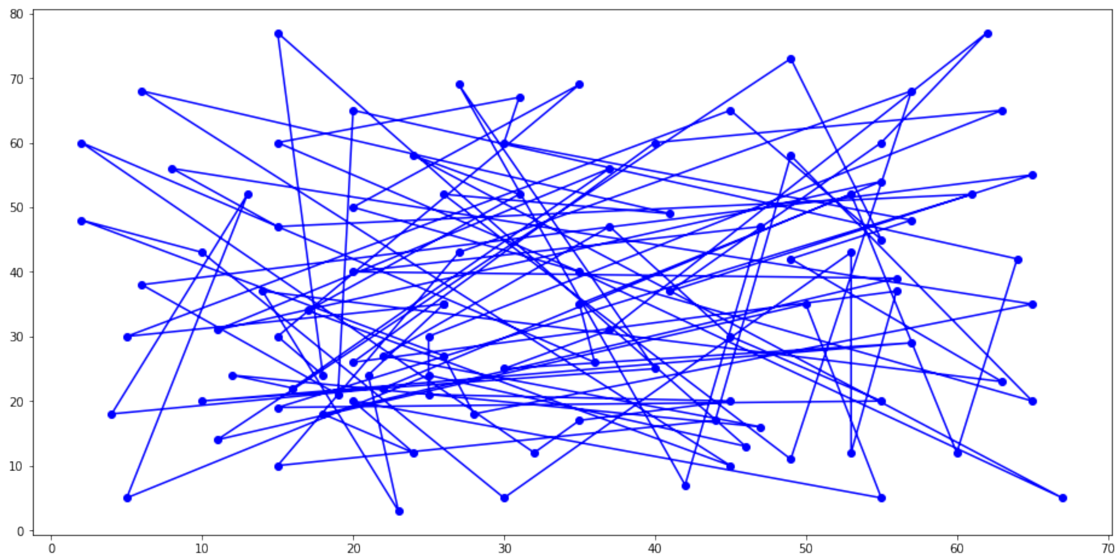
Application 1. Implement the Simulated Annealing for TSP. Use test instances from http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Exercises: 1. Analyze the behavior of the algorithm for the problems: eil51.tsp, eil76.tsp, eil101.tsp 1. Test the algorithm for each of the cooling schedules mentioned above 1. Modify the function which compute the cost of a tour such that the distance between two nodes is computed only once (hint: store the distances in a matrix)

[6]:
```
%matplotlib inline
import random, numpy, math, copy, matplotlib.pyplot as plt
import numpy as np
class City:
    """class for the coordinates of a location """
    def __init__(self, coords):
        self.x = int(coords[0])
        self.y = int(coords[1])

    def distance(self, city):
        """
        Euclidean distance between two locations
        """
        xDis = self.x - city.x
        yDis = self.y - city.y
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance

    def __repr__(self):
        return "(" + str(self.x) + "," + str(self.y) + ")"

    #def getCoord(self):
    #    return [self.x, self.y]

class TSP:
    def __init__(self, filename = None, cities_no = 10):
        """
         random generation of location coordinates / reading data from a .tsp␣
 ↪file
        """
        if filename is None:
            self.N = cities_no
            self.cities = [City(random.sample(range(100), 2)) for i in␣
 ↪range(self.N)];
        else:
            self.N, self.cities = self.___read_TSP_file(filename)
```

```python
    def ___read_TSP_file(self, filename):
        nodelist = []

        # Open input file
        with open(filename, 'r') as infile:

            # Read instance header
            Name = infile.readline().strip().split()[1] # NAME
            FileType = infile.readline().strip().split()[1] # TYPE
            Comment = infile.readline().strip().split()[1] # COMMENT
            Dimension = infile.readline().strip().split(":")[1] # DIMENSION
            EdgeWeightType = infile.readline().strip().split()[1] # EDGE_WEIGHT_TYPE
            infile.readline()

            # Read node list
            N = int(Dimension)
            for i in range(N):
                coords = infile.readline().strip().split()[1:]
                nodelist.append(City(coords))

        return N, nodelist

    def eval(self, tour):
        """computation of a tour cost"""
        val = 0
        for i in range(self.N-1):
            val += self.cities[tour[i]].distance(self.cities[tour[i+1]])
        val += self.cities[tour[0]].distance(self.cities[tour[self.N-1]])
        return val

    def displayTour(self, tour):
        """plot the tour"""

        plt.figure(figsize = (16,8))

        plt.axes()
        plt.plot([self.cities[tour[i % self.N]].x for i in range(self.N+1)], [self.cities[tour[i % self.N]].y for i in range(self.N+1)], 'bo-');
        plt.show()

    def init_solution(self):
        """initial solution - random perturbation"""
        return random.sample(range(self.N), self.N);

    def perturb_solution(self, S):
        """2-opt perturbation"""
```

```
        i, j = random.sample(range(self.N),2)
        if i > j: i,j=j,i
        new_S = S.copy()
        for k in range((j-i)//2):
            new_S[i+k],new_S[j-k]=new_S[j-k],new_S[i+k]
        return new_S


prob = TSP("eil101.tsp")
#prob = TSP()
tour = random.sample(range(prob.N),prob.N); # random initial tour
prob.displayTour(tour)
print(prob.eval(tour))
```



3397.984945534632

[7]:
```
import random
import math

def accept(current_cost, new_cost, T):
    if new_cost <= current_cost:
        return True
    if random.random() < np.exp(-(new_cost-current_cost)/T) :
        return True
    else:
        return False

def updateTemperature(T, k):
```

```
    return  T*0.9995
    #return T/k

def SA(prob, T_Max, T_Min):
    S = prob.init_solution()
    S_cost = prob.eval(S)

    #prob.displayTour(S)

    S_best = S
    S_best_cost = S_cost

    T = T_Max
    k=0
    while T > T_Min:
        k=k+1
        S_prim = prob.perturb_solution(S)
        S_prim_cost = prob.eval(S_prim)

        if accept(S_cost, S_prim_cost, T):
            S = S_prim.copy()
            S_cost = S_prim_cost
        if S_cost < S_best_cost:
            S_best = S.copy()
            S_best_cost = S_cost
            print(S_best_cost, T)

        T = updateTemperature(T,k)

    return S_best_cost, S

best, S = SA(prob, 1000., 0.000001)
prob.displayTour(S)
print("best", best)
```

```
3509.8763404010406 994.5137293956111
3503.8233199648257 994.0164725309133
3501.3047449700653 993.519464294648
3494.1330920860028 993.0227045625006
3486.9841141589122 992.0299301136143
3462.845183045833 989.5523341234284
3440.510249240353 979.2138219691252
3426.733954944235 976.7682342250463
3413.885674992578 975.7917101828799
3369.0458949297345 975.3038143277885
3350.4658384511504 974.8161624206247
3344.3188152608654 974.3287543394144
```

3323.667451424176 973.8415899622448
3321.8758827862293 973.3546691672638
3275.395345613939 966.5632860750145
3273.513784397381 958.8597091353823
3264.4277130866853 958.3802792808146
3262.7106812747056 957.9010891411743
3230.763832999593 905.2674235521021
3222.1771213458587 704.2740984433084
3198.262436846813 703.5700004133898
3157.7620327615336 703.2182154131832
3156.4158922718602 701.812833458115
3141.9102169670705 700.7606404798265
3130.149597592858 669.248118145823
3118.2807727720465 643.9655573522036
3057.1966789334947 643.3217527862408
3050.558741239188 643.0000919098477
3043.41140797118 634.0585916441267
3038.5532052288736 615.0070544966559
3012.9377530191005 614.6995509694076
3012.679457181675 608.2773754464006
2969.6966425397527 607.9732367586774
2931.514704437692 607.6692501402981
2908.364314218007 59.892507831051674
2880.654150023758 59.86256157713615
2872.822391016347 59.65335662829667
2830.769384545448 59.56392132591504
2816.3557267461615 49.40261106209782
2797.8047551795357 48.56965857985555
2795.72510673166 48.424131618957055
2735.0506483341514 48.399919553147576
2693.1927465199815 47.03965757967659
2689.0466000942447 47.016137750886756
2686.99787417905 46.758194496412166
2675.7624633971445 46.73481539916396
2675.2028635207225 34.636723731116966
2654.1929261829873 34.60209566656679
2632.1801561751126 33.545626459671226
2630.8348357260193 33.52885364644139
2630.8181068694694 32.68446844198679
2613.8052670468282 23.898643924869635
2583.2333876512484 22.149229524668616
2554.0974999607524 22.127085832451332
2531.319630062627 22.10496427839034
2519.4119038532735 22.08286484035302
2517.7737582144155 22.060787496228883
2504.668339429556 22.03873222392953
2501.5229270076597 21.950731411984645
2493.7237265537046 21.928786168255513

2490.5991487050387 21.44077716123251
2464.6769132587497 21.41934174426557
2463.6486848256845 21.387228793478066
2452.0262887456684 21.33381416283901
2443.6147507044357 21.291178524569006
2412.0981017366103 21.280532935306724
2404.285812781348 21.2486280936434
2392.6458633464126 21.18496183861618
2386.3865461637074 21.163782173018024
2382.1358682240066 20.89036570357265
2373.068296622001 20.869480560460506
2360.2732388472928 20.682451368346502
2357.5405064941583 20.67211014266233
2357.020145351872 20.641117478946935
2348.658444543481 20.630796920207462
2342.950144103117 20.599866195345992
2280.7811101822977 20.527874722880448
2249.9167140013174 20.517610785519008
2232.6131880928356 19.034722737878724
2231.3846093908246 18.9302930980592
2227.231404671357 18.676388373782856
2204.1703220312315 18.564637690924993
2193.966129117487 16.130686231302274
2185.2650647409987 15.874585491821112
2175.290041202416 12.618475374510833
2159.187442802709 12.605860053755167
2147.3417057239208 12.568089714066272
2121.125767422372 12.555524766374635
2115.850491876974 11.91320751368251
2108.1891677766434 11.89534663582847
2090.5151925050764 11.883454263029302
2089.232805763904 11.847848433496207
2056.3158894247053 11.836003547024822
2053.8784647476637 10.790033688997417
2041.4215874772206 10.773856729640436
2041.0674756635128 10.746949008994207
2035.205221075889 10.730836644349106
2032.648453612701 10.704036366419652
2027.6961464975222 10.429250419297457
2002.4757873225299 10.372032729296254
1982.136159999756 10.273940568963527
1981.6728543875902 10.263669196879707
1981.4938051258994 10.258537362281269
1973.6071359206683 9.985195240403304
1972.4136434742734 9.930413757276796
1959.201255947338 9.925448550398158
1943.9751745349256 9.890761545674525
1916.799109422888 9.866059356353844

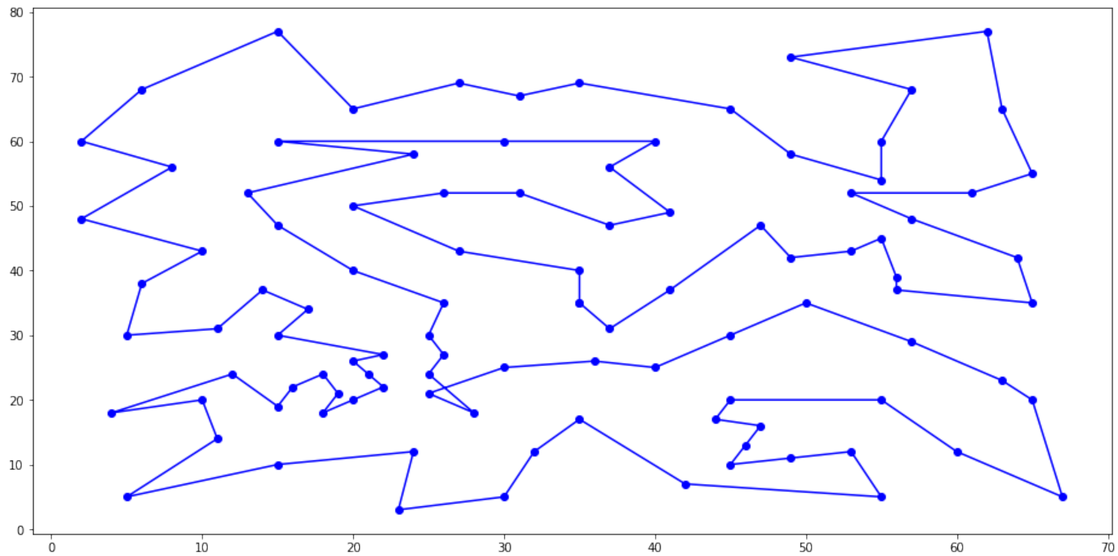
1897.0890887808184 9.821750780268497
1892.894088057955 9.772752373866007
1888.2919718186497 9.709419703167603
1874.3377144892768 9.70456499331602
1873.4794699312486 9.685170415325203
1864.0219718859169 9.680327830117541
1844.109183609142 9.670649922369382
1836.0421096524924 9.102784787712453
1826.220916276827 9.089137436481629
1796.5348009005509 9.025719839315263
1790.722281569135 9.016696375905909
1748.0774966037702 9.012188027717956
1746.8417704547687 6.793778303258232
1745.3106873413094 6.739631415388654
1739.1559882114266 6.702655973669991
1727.672200671015 6.669217998229429
1724.6410711412964 6.622684893918695
1719.1607443189184 6.530590445591521
1717.57828910396 6.507767636074742
1716.9060785576544 6.465593815547025
1711.4230314262409 6.4591298381299325
1704.7556747968706 6.452672323074263
1692.809694193567 6.433338487500623
1664.4074126233675 6.312228961101036
1643.158333787423 6.2996139683665575
1617.8778007140038 6.296464161382374
1613.7285586971345 6.26505258143291
1613.6450626059247 6.218228788911465
1610.264354851865 6.046490996747148
1599.6357016470342 6.010311651776527
1598.6119056054058 5.992303240469747
1572.6467210680867 5.980327619447893
1571.2186739557292 5.947517923824701
1569.4347771415546 5.926732809553032
1555.9046277029593 5.917847154647469
1551.2008580357387 5.838473351910927
1535.8539533727496 5.465484893751321
1534.0688671111113 5.440939343748913
1527.8195588605515 5.370647445369291
1522.8849574757526 5.3492024125488875
1505.2890857062039 5.3251790883896115
1503.258369855178 5.301263653231191
1498.6808962880364 5.22231852952023
1492.1678298580587 5.204067809002969
1483.749691782557 5.191070643146527
1475.235335022079 5.149697447335485
1461.1387228080223 5.141976762793856
1460.57105113991 5.134267653489491

1458.984220665249 5.116324649156267
1458.5175770563417 5.030060882427112
1455.561996777459 4.903384646149699
1446.8061335176817 4.808671421170249
1444.5201709393712 4.801462019940976
1440.10171024418 4.787075627364574
1435.8426805880442 4.770345973882856
1413.4056689106321 4.7346933297536475
1409.157544546385 4.718146739535563
1405.4307980557855 4.715787666165796
1404.0159871912845 4.713429772332713
1401.1534592417715 4.627011549075112
1401.045847708708 4.603928475865186
1396.848682663942 4.578670078403338
1394.0067409566989 4.560387416749608
1386.0755535560986 4.490229003403635
1369.933575875971 4.4566698993903255
1360.3675570080275 4.43665494811221
1359.7050006064055 4.22233512984565
1358.9361646749567 4.201270892186992
1353.9977122051034 4.199170256740898
1339.304694716916 4.1615379151791565
1338.2134143869002 4.153221079575163
1334.9372750785806 4.144920865171281
1324.8835589489806 4.050652360656232
1322.44489482785 4.0264151709885265
1322.3352220597712 3.9843485306016784
1314.2255272428981 3.926978074332386
1310.9010492774696 3.837666272642439
1309.4419521073703 3.797571617680084
1292.6051100636207 3.780516683700473
1285.2298975141755 3.776737112145944
1285.0347809899756 3.7616565744365036
1283.3671525493155 3.752261832440954
1281.8199834345658 3.7260811930771602
1276.5520183016572 3.7130594545341986
1268.9664539796106 3.646805890937346
1264.667484665254 3.6249851277144955
1258.8163045539648 3.528396199722795
1257.442942045139 3.5090384633716183
1244.8346986035979 3.500274635424147
1236.3725175830357 3.486298014318286
1231.2479057855378 3.441258272049317
1225.7990555483632 3.4292319196550953
1222.305454900759 3.4104182253786592
1222.1077322804101 3.3545941826573458
1214.716870258054 3.3295225938780435
1204.338290959821 3.1957400069975805

1198.2291752267158 3.1481498272398314
1193.0305989703913 3.1105884572557216
1190.1689387219515 2.9945753034999063
1189.4717296608192 2.976657179913167
1184.9734086344529 2.9618073367960642
1173.8256669566247 2.897343164578742
1170.9294923129737 2.8527695394084236
1165.3631442025983 2.8371184737432706
1161.3631439796268 2.793471363881224
1149.3165024566574 2.7670524691416465
1149.2316987964593 2.7449988675832917
1148.0694127186514 2.6852535635799386
1144.464691045986 2.6571989315537823
1135.8940958674116 2.653215125723501
1132.0843888096492 2.6479126739683014
1130.0789751248562 2.629437404949326
1127.4655904643055 2.5696357489120976
1126.3012380163425 2.5326356787052093
1126.17736303866 2.5111941701515015
1118.7659467866563 2.4725611807843295
1117.0859680384517 2.392278540391177
1114.1615800699303 1.8987361995094285
1113.8573402202285 1.849068279449133
1094.963329881228 1.8315809549421849
1089.8082631215686 1.8160746351653585
1089.2307625922979 1.769453737068071
1068.4751845625244 1.7396198709865578
1057.1968177452081 1.6531025740123093
1049.3555650386465 1.6325623005674628
1037.0127061330481 1.6098600390468272
1035.8897903953048 1.581134680407935
1034.2527414142198 1.5176035639228507
1026.1175916411935 1.382798342316611
1018.1485446808367 1.3332240441892946
1013.2672036261754 1.1777067223681466
1004.3375197455541 1.1677361573264642
999.6182994419162 1.072557205761041
996.6437495156531 0.9802214157478366
995.2447542447721 0.9782624427584152
992.8011839294068 0.9641764856035353
986.8543474715079 0.8949391462940404
985.6533171057256 0.8585503646917169
981.6334439838145 0.8478826302545619
980.9067580210415 0.8444970289804854
978.1476591497992 0.7846382919704439
974.1479571665352 0.7304822672414839
971.4844386188079 0.7297519675948093
968.6560114940617 0.7293870916110119

966.9361213833422 0.7250227839127601
965.1739119215623 0.6780264675494707
965.1580047388098 0.6439837195408675
964.6170651743015 0.6334425564647502
964.2769670441786 0.6328092722689247
959.0248772877754 0.6274518552751163
956.0428701344315 0.6190360854037751
950.4140825940165 0.6159478598552961
949.6501505715164 0.613795273378642
947.0138144293351 0.6064719270885808
933.3793363214525 0.585900402009606
931.0020505659948 0.5697187456432872
928.2056866597025 0.5548158090885544
927.36359881202 0.5520479634060047
924.0188970599631 0.5080674304171706
923.6985353528285 0.5002511636105584
910.4191490428693 0.48449266740028674
903.8590502828956 0.46154989724616147
903.6598165721849 0.4585588066572562
902.5622180191762 0.4535410534918116
901.0256829813496 0.3907424037479362
900.9076230454376 0.37560522211623043
900.5396786348118 0.37001180087441454
900.1042580560613 0.3696418815764904
898.2643456527643 0.35496689120870484
896.2843874587896 0.3530194467879353
895.3398000844543 0.34845876240755214
884.0864175692589 0.3420699566800195
877.4464704557818 0.334123067567781
875.9093836421628 0.32947678285487025
871.0917633263247 0.3105954106336085
870.5699869886151 0.30520593892226283
868.2292132458393 0.2991609601096141
863.6133818856575 0.29089860043532867
852.6964880819279 0.2685278808006337
852.6777562909871 0.22563332663532787
851.4039670434539 0.21678394877881196
849.4092842326874 0.20964022601297264
846.3793843723278 0.19693568863747468
844.4758851374526 0.19487817036341784
834.669047394851 0.19063661832874246
831.8159853331548 0.18704782681636206
829.2042701431016 0.17034758370948627
828.4169577270651 0.12063321741360349
822.917045383792 0.1158440109091091
819.9111474990374 0.10855188887931772
818.3254618174574 0.10325645964292487
816.2122294406786 0.10305010155670673

814.138636858714 0.10000376050630505
812.1093507833758 0.08628630570455094
800.4535862620573 0.0842395815996064
798.3982573121177 0.07463687148674233
798.1862671437175 0.07206913654213262
796.5662692925403 0.0674312970118006
788.6951330934774 0.06716204337110832
788.1627082908061 0.061012834834393015
785.0224602880926 0.0566031984262188
780.2172858747704 0.05476532510697784
779.6031213444523 0.05413897568163852
778.2134886719609 0.054057807815580464
774.9260710114543 0.05005061195981782
773.1647536081575 0.04610927974904427
772.2040195141421 0.038821383326449674
772.023679626643 0.03373163343824286
771.1727764562776 0.028901613872224237
770.0115140534703 0.027001176286444953
755.440732065027 0.025543082889305558
752.2369569782899 0.020734819927952477
747.0820704941251 0.020121862423156157
746.8437384194309 0.01979249410721658
741.0373325035918 0.01744880683043663
738.4343917929795 0.0108770286862757
738.4343917929793 0.00928698160689373
733.10849649295 0.009171588197748002
731.0661800405846 0.00624645706999132
731.0661800405845 0.0038821748291445078
729.3442088893268 0.0034190535020650037
726.7091073545242 0.0005476635482404924
724.8626937324223 3.358109140168668e-05
724.3791562660292 2.9992126380755145e-05
723.6987676360811 2.6361447806135323e-05
721.1732718560188 2.569756538622844e-05
714.1702978641586 1.7926968969761755e-05
711.5649208020384 1.2906409816536709e-05
710.0860144020526 1.0736911294084815e-06




```
best 710.0860144020526
```

1.3.3 Solving the knapsack problem using Simulated Annealing

1. Solution encoding: binary vector

- $S_i = 1$ if object i is selected
- $S_i = 1$ if object i is not selected

2. Local perturbation: change the value of a randomly selected component: $S_i = 1 - S_i$
3. Evaluation of a configuration: a common variant is to include in the objection function the degree of constraints satisfaction (penalty function technique) – the value of a configuration which does not satisfy the constraint is penalized by a term which is proportional with the amount by which the constraint is violated (e.g. the weight which overpasses the knapsack capacity).

$V(S) = \sum_{i=1}^{n} v_i S_i$ if $\sum_{i=1}^{n} w_i S_i \leq C$
$V(S) = \alpha \sum_{i=1}^{n} v_i S_i + (1-\alpha)(C - \sum_{i=1}^{n} w_i S_i)$ if $\sum_{i=1}^{n} w_i S_i > C$

Notations: v_i denotes the value of object i, w_i denotes the weight of object i and C it the knapsack capacity, $\alpha \in (0,1)$ quantifies the relative importance of the total value and the constraint satisfaction.

Application 2. Implement a Simulated Annealing algorithm for a knapsack problem. Test data can be found at http://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

[8]:
```
class Knapsack:
    def __init__(self, capacity, objects_values, objects_weights):
        self.capacity = capacity
        self.objects_values = np.array(objects_values)
        self.objects_weights = np.array(objects_weights)
        self.N = len(objects_values)
```

```
    def eval(self, S, alpha=0.5):
        weight = np.sum(S * self.objects_weights)
        value = np.sum(S * self.objects_values)
        #print (weight, value, S)
        if weight <= self.capacity: # the selected objects fit into the␣
↪knapsack
            return -value
        else:
            return  -(alpha*value + (1-alpha) * (self.capacity - weight))

    def init_solution(self):
        """ Solution initialization - each candidate solution is encoded as a␣
↪binary array S
            S[i]=0 if object i is not selected
            S[i]=1 if object i is selected
        """
        return np.random.choice([0, 1], size=(self.N,))

    def perturb_solution(self, S):
        """
        A new candidate solution is constructed by complementing a randomly␣
↪selected element
        (a random object is either inserted or removed from the knapsack)
        """
        i = random.sample(range(self.N),1)
        new_S = S.copy()
        new_S[i] = 1 - S[i]
        return new_S
```

[9]:
```
#prob = Knapsack(165, [92,57,59,68,60,43,67,84,87,72],␣
↪[23,31,29,44,53,38,63,85,89,82])
#optimal solution [1,1,1,1,0,1,0,0,0,0]
prob = Knapsack(20, [3,4,1,5,2], [10,8,3,6,5])
best, S = SA(prob, 1000., 0.000001)
#S=[1,1,1,1,0,1,0,0,0,0]
print("best", np.sum(S * prob.objects_values), S)
```

```
-10 998.5007498750001
-11 993.519464294648
best 11 [0 1 0 1 1]
```

1.4 Tabu Search

1.4.1 Description of the method

Tabu Search is a metaheuristic based on an iterated local search which relies on the usage of a list of already visited configurations which become "forbidden" (tabu) at least for a given number of iterations.

General structure of Tabu Search:

```
S=initial configuration
Sbest=S
TabuList=[] // the tabu list is initially empty
Iter=1
Repeat
    S=perturb(S,TabuList)
    If better(S,Sbest) then Sbest=S endif
    iter=iter+1
    Until iter<=iterMax
```

The perturbation of the current configuration is based on the identification (in its neighborhood) of a better configuration which is not in the tabu list. Once a configuration is chosen it is inserted in the tabu list. The tabu list is implemented as a circular queue (when the maximal size of the list is reached the first element in the list is removed).

```
Perturb(S,TabuList)
Sbest=S
For each element S' from the neighborhood N(S) > > If better(S',Sbest) and (S' is not in
TabuList) then >> >> Sbest=S' >> > Endif
Endfor
S=Sbest
update the TabuList by adding S
Return S, TabuList
```

The function better(S',Sbest) checks if configuration S is better than configuration Sbest. Unlike Simulated Annealing which uses directly the value of the objective function to compute the acceptance probability, in Tabu Search it is enough to decide which of the configurations is better. This means that the constraints can be analyzed directly, without using the penalty method.

1.4.2 Example: Solving the knapsack problem using Tabu Search

1. Solution encoding: binary vector
2. Local perturbation: change the value (0->1,1->0) of a randomly selected component
3. Tabu list structure: it contains candidate solutions (binary vectors)
4. Comparison between two candidate solutions:

- If both S and S' are feasible then the configuration having a higher value is better.
- If only one of the solutions is feasible then it is better than the other one (a feasible solution is always better than an unfeasible one).
- If none of the solutions is feasible then that which violates less the constraint is better

[10]:
```
# Remark: other implementations: https://www.techconductor.com/algorithms/
↪python/Search/Tabu_Search.php


def ElementInTabuList(el, TabuList):
    '''
    :param el current element (to be searched in the TabuList)
    :param TabuList - list with "forbidden" elements

    :return True if el is in TabuList and False otherwise
    '''
    dim = list(range(len(el)))
    for a in TabuList:
        found = True
        for i in dim:
            if el[i] != a[i]:
                found = False
                break
        if found:
            return True;
    return False

def PerturbTabuSearch(prob, S, cost_S, TabuList, dimNeighborhood,␣
↪TabuListMaxSize):
    '''
    :param prob - problem to be solved
    :param S - current candidate solution
    :param cost_S - cost of S
    :param TabuList - list  with "forbidden" elements
    :param dimNeighborhood - size of the neighborhood
    :param TabuListMaxSize - maximal size of the TabuList

    :return new candidate solution and update TabuList
    '''
    C = prob.perturb_solution(S) # element din vecintatea lui S
    cost_C = prob.eval(C)
    # genereaza candidai i îl selecteaz pe cel mai bun care nu e in tabuList
    for i in range(dimNeighborhood-1):
        el = prob.perturb_solution(S)
        cost_el = prob.eval(el) # genereaz un nou candidat
        if (not ElementInTabuList(el,TabuList)) and cost_el < cost_C:
            C = el.copy()
            cost_C = cost_el

    if cost_C < cost_S:
        S = C.copy()
        cost_S = cost_C
```

```
        if len(TabuList) > TabuListMaxSize:
            TabuList.pop(0)
        TabuList.append(S)

    return S, cost_S, TabuList

def TabuSearch(prob, max_iter=1000, dimNeighborhood = 20, TabuListMaxSize=10):
    S = prob.init_solution()
    cost_S = prob.eval(S)
    Best=S.copy()
    cost_Best = cost_S
    TabuList=[]
    TabuList.append(S.copy())
    it=1
    while it < max_iter:
        S, cost_S, TabuList = PerturbTabuSearch(prob, S, cost_S, TabuList,␣
 ↪dimNeighborhood, TabuListMaxSize)
        if cost_S < cost_Best:
            Best = S.copy();
            cost_Best = cost_S
        it=it+1
    print (TabuList)
    return Best, cost_Best

prob = Knapsack(20, [3,4,1,5,2], [10,8,3,6,5])
S, best = TabuSearch(prob,dimNeighborhood=prob.N)
print(best,S)
print(prob.objects_values)
#prob.displayTour(S)
print("best", np.sum(S * prob.objects_values))
```

```
[array([0, 1, 0, 0, 0]), array([0, 1, 0, 1, 0]), array([0, 1, 0, 1, 1])]
-11 [0 1 0 1 1]
[3 4 1 5 2]
best 11
```

1.5 Continuous optimization problems: zero order (without derivatives) methods

1.5.1 Pattern Search

Main idea: search by constructing 2n candidate solutions from the current one (by sequentially adding a positive and a negative adaptive step on each of the n coordinates). Details: lecture 2

1.5.2 Nelder Mead

Main idea: use a set of (n+1) points (simplex) to search the function landscape and update the vertices of the simplex by using some geometric transformations (extension, contraction, shrinking) in such a way that the quality of the vertices is improved. Details: lecture 2

Exercises:

1. Compare the behavior of Pattern Search and Nelder Mead for the 2D Sphere, 2D Rosenbrock and 2D Ackley functions (using the same computational budget, i.e. number of function evaluations)
2. Analyze the influence of the control parameters: (i) initial value of r in the case of Pattern Search and r, e, c and s in the case of Nelder Mead
3. Modify the test functions and the Pattern_Search function such that it can be applied for functions with an arbitrary number of components (hint: exclude the operations used for the graphical illustration)

[11]:
```
# Test functions: Sphere, Rosenbrock, Ackley
# 2-dimensional case

def Sphere(v):
    x = v[0]
    y = v[1]
    return x**2 + y**2

def Rosenbrock(v):
    x = v[0]
    y = v[1]
    return (1.0 - x)**2 + 100.0*(y - x**2)**2

def Ackley(v):
    x=v[0]
    y=v[1]
    term1 = -20 * np.exp(-0.2 * ((1/2.) * (x**2 + y**2)**(0.5)))
    term2 = np.exp((1/2.)*(np.cos(2*np.pi*x) + np.cos(2*np.pi*y)))
    return term1 - term2 + 20 + np.exp(1)
```

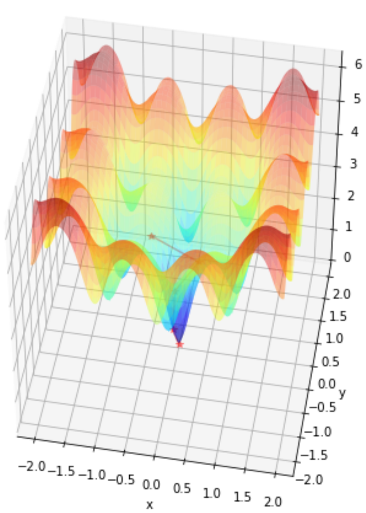
[12]:
```
def Graphical_illustration(func, iter_x, iter_y, x_start=-2, x_stop=2,␣
 ↪y_start=-2, y_stop=2):
    x = np.linspace(x_start,x_stop,250)
    y = np.linspace(y_start,y_stop,250)
    X, Y = np.meshgrid(x, y)
    Z = func([X, Y])

    #Angles needed for quiver plot
    anglesx = iter_x[1:] - iter_x[:-1]
    anglesy = iter_y[1:] - iter_y[:-1]

    %matplotlib inline
    import matplotlib.pyplot as plt
    #plt.style.use('seaborn-white')
    from mpl_toolkits import mplot3d

    fig = plt.figure(figsize = (16,8))
```

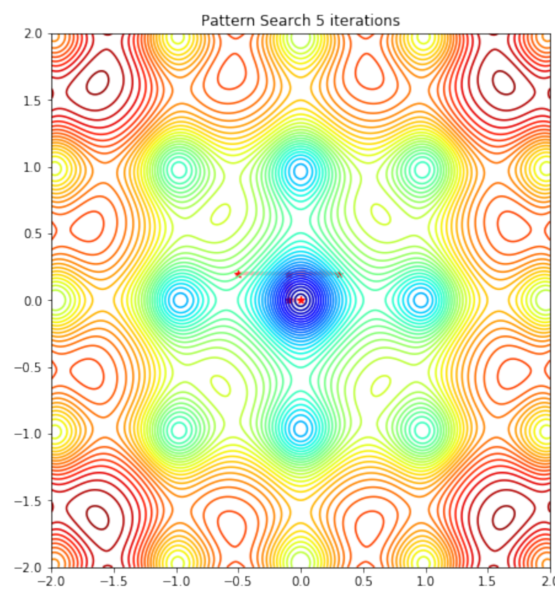
```
    #Surface plot
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    ax.plot_surface(X,Y,Z,rstride = 5, cstride = 5, cmap = 'jet', alpha = .4,␣
 ↪edgecolor = 'none' )
    ax.plot(iter_x,iter_y, func([iter_x,iter_y]),color = 'r', marker = '*',␣
 ↪alpha = .4)

    ax.view_init(45, 280)
    ax.set_xlabel('x')
    ax.set_ylabel('y')


    #Contour plot
    ax = fig.add_subplot(1, 2, 2)
    ax.contour(X,Y,Z, 50, cmap = 'jet')
    #Plotting the iterations and intermediate values
    ax.scatter(iter_x,iter_y,color = 'r', marker = '*')
    ax.quiver(iter_x[:-1], iter_y[:-1], anglesx, anglesy, scale_units = 'xy',␣
 ↪angles = 'xy', scale = 1, color = 'r', alpha = .3)
    ax.set_title('Pattern Search {} iterations'.format(len(iter_count)))

    plt.show()
```

[33]:
```
# Pattern Search implementation

%matplotlib inline
import random, numpy, math, copy, matplotlib.pyplot as plt
import numpy as np
def Pattern_Search(f ,x, y, r=0.8, nMax = 50):
    #Initialization
    i = 0
    iter_x, iter_y, iter_count = np.empty(0),np.empty(0), np.empty(0)
    S = np.array([x,y])
    E = np.array([0,0])
    best = S
    val_best = f(S)
    points =[]
    # Iterating as long as the number of iterations is smaller than a maximal␣
 ↪value
    while i < nMax:
        i +=1
        S_prim = S
        for j in range(len(E)):
            E[j-1] = 0
            E[j] = 1
            val_S_prim = f(S_prim)
```

```
            S_plus = S + r*E
            if f(S_plus)< val_S_prim:
               S_prim = S_plus
            S_minus = S - r*E
            if f(S_minus)< val_S_prim:
               S_prim = S_minus
        if np.array_equal(S, S_prim ):
            r = r/2
        else:
            S = S_prim

        if f(S) < val_best:
            best = S
            val_best = f(S)
            x,y = S[0], S[1]

            iter_x = np.append(iter_x,x)
            iter_y = np.append(iter_y,y)
            iter_count = np.append(iter_count ,i)


    return best, iter_x,iter_y, iter_count


best,iter_x,iter_y, iter_count = Pattern_Search(Ackley, -0.5,1)
print("best", best)
Graphical_illustration(Ackley, iter_x, iter_y,  x_start=-2, x_stop=2,␣
 ↪y_start=-2, y_stop=2)
```

```
best [ 2.77555756e-17 -5.55111512e-17]
```





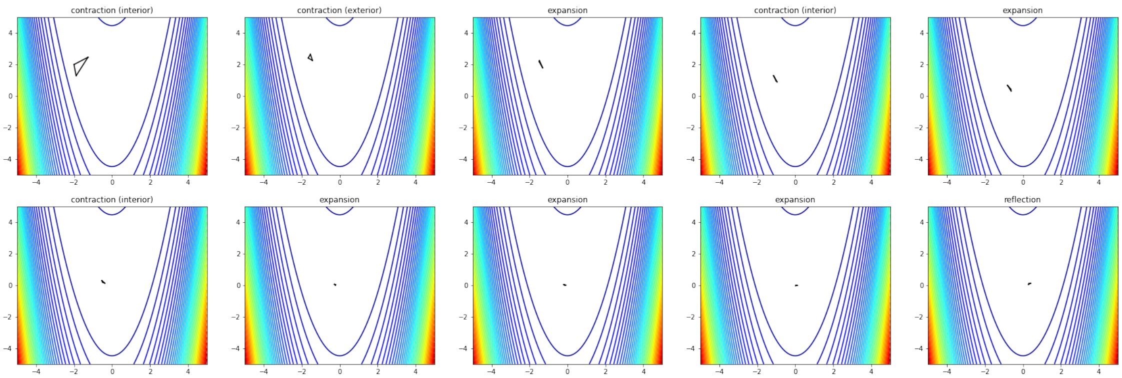
[13]:
```
# Nelder Mead implementation

class Element:
    def __init__(self, el, cost):
        self.el = el
        self.cost = cost
    def __repr__(self):
        return "[{self.el}, {self.cost}]".format(self=self)

def NelderMead(func, x_start,  start = -5, stop = 5, max_iter = 50, r=1, e=2, c␣
↪=0.5, s=0.5):
    """
    :param fun - function to be optimized
    :param x_start - initial configuration
    :param start - lower bound of the function domain
    :param stop - upper bound of the function domain
    """
    n = len(x_start)
    best = Element(np.array(x_start), func(x_start))

    points =[]
    modification="initial simplex"

    # selection of the initial vertices of the simplex - (n+1) random points
    l = [best]
    for i in range(n):
        el = np.random.uniform(low=start, high=stop, size=(n,))
        l.append(Element(el, func(el)))
    print(l)
    it = 0
    while it < max_iter:
        it += 1
        # increasing sorting by cost (first vertex is the best one, the last␣
↪vertex is the worst one)
        l = sorted(l, key=lambda x: x.cost)
        if it%5==0:
            points.append([[x.el for x in l], modification])

        # average of the first n vertices
        M = np.zeros(n)
        for i in range(n):
            M += l[i].el
        M /= n
        # Sequence of transformations
        # Reflection
```

```python
        xr = M + r * (M - l[n].el)
        cost_xr = func(xr)
        if l[0].cost <= cost_xr < l[n-1].cost:
                l.pop()
                l.append(Element(xr, cost_xr))
                modification = "reflection"
                continue

        # Expansion
        if cost_xr < l[0].cost:
                xe = M + e * (xr - M)
                cost_xe = func(xe)
                if cost_xe < cost_xr:
                    l.pop()
                    l.append(Element(xe, cost_xe))
                    modification = "expansion"
                    continue
                else:
                    l.pop()
                    l.append(Element(xr, cost_xr))
                    modification = "expansion"
                    continue

        #  Contraction (Exterior/ Interior)

        if l[n-1].cost<=cost_xr  < l[n].cost:
                xc = M + c * (xr - M)
                cost_xc = func(xc)
                if cost_xc < cost_xr:
                    l.pop()
                    l.append(Element(xc, cost_xc))
                    modification = "contraction (exterior)"
                    continue
        elif cost_xr>= l[n].cost:
                xcc = M + c * (l[n].el - M)
                cost_xcc = func(xcc)
                if cost_xcc < l[n].cost:
                    l.pop()
                    l.append(Element(xcc, cost_xcc))
                    modification = "contraction (interior)"
                    continue

        # Shrinking
        new_l = [l[0]]
        for i in range(1,n+1):
            v = l[0].el + s * (l[i].el - l[0].el)
            new_l.append(Element(v, func(v)))
```

```
        l = new_l
        modification = "shrinking"
    return l[0], points

sol, points = NelderMead(Rosenbrock, [-2,2], -5,5)
print("Sol=", sol)
```

```
[[[-2  2], 409.0], [[-1.90597918  2.29983027], 186.11397953451342], [[-1.2966844
4.67581803], 901.9344225376697]]
Sol= [[-0.52047568  0.24802941], 2.364129510195738]
```

[26]:
```
def FunctionPlot(func, points, start=-2, stop=2):
    x = np.linspace(start,stop,2500)
    y = np.linspace(start,stop,2500)
    X, Y = np.meshgrid(x, y)
    Z = func([X, Y])

    %matplotlib inline
    import matplotlib.pyplot as plt
    #plt.style.use('seaborn-white')
    from mpl_toolkits import mplot3d

    fig = plt.figure(figsize = (30,15))

    grid_y = 5
    grid_x = len(points) // grid_y + 1

    k=0
    for t in points:
            #Contour plot
            ax = fig.add_subplot(grid_x, grid_y, k+1)
            ax.contour(X,Y,Z, 50, cmap = 'jet')
            #Plotting the iterations and intermediate values
            plt.triplot([el[0] for el in t[0]],[el[1] for el in t[0]],␣
 ↪color="black")
            ax.set_title(t[1])
            k += 1



    #ax.set_title('Function (surface and contour plot)'.
 ↪format(len(iter_count)))

    plt.show()

FunctionPlot(Rosenbrock, points, start=-5, stop=5)
```


contraction (interior)
contraction (exterior)
expansion
contraction (interior)
expansion
contraction (interior)
expansion
expansion
expansion
reflection

[ ]: