Drilled Board Production

A TSP approach to optimize the process

Giovanni Misseri

Models and methods for combinatorial optimization

2019, February

1 Introduction

In this paper I'm going to analyze the problem of drilling boards and in particular how to optimize the production time in term of path to follow in order to drill a board. On that problem I will propose two different approaches, an exact optimization method and an approximation algorithm, depending on the situation one could prefer one or the other. For completeness I will underline the advantages and disadvantages of the first and second approch respectively, it will be a company's task to decide in which way solve its problem.

As I said, the problem I will have to deal with is very specific: it is needed to optimize the production time of drilled boards. The machine specific are such that the time spent for drilling each hole is the same, so we can't work on it, our optimization problem is independent on the drilling time. What we can work on is the path the drill takes in order to go through each hole.

The drill indeed takes time to move into the space and if we are able to find the shortest path through the holes position we would see our problem solved.

As it's now clear our optimization problem reduces to a traveling salesman problem, in fact we have to find the shortest path to go through the nodes. In

order to actually implement and test the algorithm so we need a cost matrix where in each position (i,j) there is the cost to go from i to j. One natural choice for the cost matrix is the distance between nodes. Notice also that, given that the drilling machine does not have structural limits, our cost matrix will be simmetric.

From what we said using the pairwise norm two distance between nodes seems meaningfull. It creates in fact a simmetric cost matrix and it measures the shortest path length between two nodes, so the shortest path our machine has to do from node j to node i. Another important fact about using norm two for computing the cost to go from one node to the other is that at the end of our algorithm we will have a score function that in this case will be sum of the distances between consecutive points that compose our shortest path. So the value of the final score function is also proportional to the time the drill takes to walk through the holes.

So given the final score function, the drilling time of each hole and the speed at which the drill moves, we would be able to calculate the exact time we take to drill a board.

2 Exact method

The first possible approach one could use is to find the exact solution to the TSP problem we are dealing with. This would give back, by definition, one of the solutions that globally minimizes the cost function. This sounds great but has also some drowbacks, the TSP problem in fact is an np-hard problem and could take really a long time to be solved (non polynomial indeed).

Model definition

One of the possible TSP formulation is the network flow formulation, so I will use this formulation and I will implement this problem on IBM OPL in order to solve it.

Let's see then the TSP problem intended as network flow implemented on

OPL.

```
int Nnodes = ...;
setof(int) I = asSet(1..Nnodes);
setof(int) J = asSet(1..Nnodes);

int start= ...;
float  C[I][J] = ...;

dvar int+ x[I][J];
dvar boolean y[I][J];
```

In these first lines I start declaring the size of the problem and the set on which the variables will range. Than I call "C" the cost matrix of the problem and just after that, the variables are initialized.

x will be a matrix (n x n) where n is the number of nodes. In each entry of $x_{i,j}$ we will find the amount of the flow shipped from i to j; y will be a matrix (n x n) where each entry $y_{i,j}$ will be 0 or 1 whether if the arc (i, j) is present or not.

```
minimize sum (i in I, j in J) (C[i][j] * y[i][j]);

subject to {

startFlow: sum ( j in J)  x[start][j] == Nnodes;

forall ( j in J :j!=start ) {
    nodeFlow: sum ( i in I ) x[i][j] - sum( i in I) x[j][i] ==1;
}

forall ( i in I ) {
    exitEdge: sum ( j in J ) y[i][j] ==1;
}

forall ( j in J ) {
    enteringEdge: sum ( i in I ) y[i][j] ==1;
}
forall ( j in J, i in I ) {
    tec: x[i][j] <= Nnodes*y[i][j];
}


}
```

The one reported above is the main part of the minimization problem definition; actually it's exactly the transposition of the TSP problem with flow formulation in OPL.

I started defining the cost function to minimize, which is simply the sum of the moves cost for all the existing arcs. This is what we want to minimize, but we put on that some conditions in order to obtain the Hamiltonian path of shortest length.

I impose that the initial flow is equal to the number of nodes I have to visit, then I impose that the flow that comes in each node minus the flow that goes outside is equal to one. I impose that for each node there is only one outgoing arc and only one incoming arc. Finally I say that if one node sends some flow to some other node, the link connecting them needs to be active.

Below an example of the file containing the data for the model.

```
Nnodes = 7;
start = 1;
C = [[0.        , 5.94481082, 6.08425041, 2.91565877, 4.36822691,
        4.61693553, 5.74479478],
       [5.94481082, 0.        , 0.14145163, 3.16252605, 1.69391934,
        7.50996878, 8.45049615],
       [6.08425041, 0.14145163, 0.        , 3.30391456, 1.83395492,
        7.60584327, 8.53721779],
       [2.91565877, 3.16252605, 3.30391456, 0.        , 1.49327624,
        5.96115689, 7.08755597],
       [4.36822691, 1.69391934, 1.83395492, 1.49327624, 0.        ,
        6.71595625, 7.77206968],
       [4.61693553, 7.50996878, 7.60584327, 5.96115689, 6.71595625,
        0.        , 1.15643066],
       [5.74479478, 8.45049615, 8.53721779, 7.08755597, 7.77206968,
        1.15643066, 0.        ]];
```

So to make the code work one would need to input the number of nodes the actual problem has, from which node one would like to start the path (actually given that we are looking for the exact minimum Hamiltonian cycle, changing the starting point doesn't change the solution) and the cost matrix.

3 Metaheuristic method

In this chapter I'm going to describe a different approach to the same problem we analysed in the previous section.

In chapter two we proposed a method to find the exact solution to the TSP

problem derived by the main problem. The exact solution is by definition the best we can achieve, but actually given that TSP problem is np-hard can happen that the problem is too big to be solved in a reasonable amount of time.

Here the heuristic methods come into play, in fact with a metaheuristic method we are able to give a, hopefully, good solution in a reasonable amount of time.

The metaheuristic method I'm going to use to solve the relevant problem is Tabu Search. I decided to implement it because it's effective and I liked local search methods.

Model Definition

As said above the metaheuristic I'm going to present is the Tabu Search. Given that Tabu Search is a local search method, we need to define what we mean with "neighbour". From now on I will consider x as the current solution and the solution representation considered will be the path representation. So $[1, 2, \ldots, n-1, n, 1]$ will be the solution in which one goes through the holes from the one called "1" to the one called "n" in order.

The Tabu Search I implemented follows an alternation of intensification and diversification phases, in order to explore the space in the most convenient way.

Below I will show comment and justify what I did.

```
def TabuSearch_tsp(dis_mat,x,length_memory):
    start_time = time.time()
    f=0
    for i in range(0,len(x)-1):
        f=f+(dis_mat[x[i]-1,x[i+1]-1])
    memory=[]
    x_temp=x
    f_temp=f
```

First of all I'm defining a function named *TabuSearch_tsp*, this will be the function to solve the TSP problem using heuristic. Its inputs are the distance matrix of the holes, a starting feasible solution and the length of the tabu list one wants to adopt.

I make the cronometer run, in order to measure later the running time. I initialize the cost function equal to zero and through the first loop I calculate the initial value of the cost function as the sum of the cost to go through the initial solution.

The tabu list is initialized and x and f are renamed.

```
for u in range(0,length_memory):
    f_trash=9999
    for i in range(1, len(x)-2):
        for j in range(i+1, len(x)-1):
            x_new = x_temp[:i] + list(reversed(x_temp[i:j+1])) + x_temp[j+1:]
            f_new= f_temp - dis_mat[x_temp[i-1]-1,x_temp[i]-1] - dis_mat[x_temp
            if f_new<f_trash and x_new not in memory:
                f_trash=f_new
                f_temp1=f_new
                x_temp1=x_new

    f_temp=f_temp1
    x_temp=x_temp1
    memory.append(x_temp)
    if f_temp<f:
        x=x_temp
        f=f_temp
```

Just after the initialization I start creating the tabu list of the specified length. So with a for loop long *length_memory* I start from the initial solution to check which is the neighbor with lower score function.

I do this considering as neighbours all the nodes I can reach from x with a 2-opt move. So I start looping through neighbours and I upde the best solution found. Once I checked all the neighbours that were not already in the memory, I move to the best neighbour and I add it to the memory. If I find a node that has lower cost function with respect to the current best solution, it becomes the new best solution.

At the end of this loop the list *memory* that contains all the tabu solution. So we can start to iterate to find the minimum.

```
itera=0
tot=0
diversification_phase=0

while itera<100 and tot<1000 and diversification_phase<5:
```

I initialize: the *itera* counter to zero, it will represent the number of iteration without changing the optimal solution; the *tot* counter, it will be needed to count the total iteration done by the algorithm; the $diversificatio\_phase$ counter, it will count the number of diversification phases done by the algorithm.

After that I open a while loop, so I will run the Tabu Search until: the best solution does not change for 100 iteration consecutively, the total number of intensification iteration, *tot*, is lower than 1000 and we computed less than six diversification phases.

```
f_trash=9999
for i in range(1, len(x)-2):
    for j in range(i+1, len(x)-1):
        x_new = x_temp[:i] + list(reversed(x_temp[i:j+1])) + x_temp[j+1:]
        f_new= f_temp - dis_mat[x_temp[i-1]-1,x_temp[i]-1] - dis_mat[x_temp[j]-
        if f_new<f_trash and x_new not in memory:
            f_trash=f_new
            f_temp1=f_new
            x_temp1=x_new
itera=itera+1
f_temp=f_temp1
x_temp=x_temp1
memory.append(x_temp)
memory.pop(0)

if f_temp<f:
    x=x_temp
    f=f_temp
    itera=0

tot=tot+1
```

The one reported above is the intensification phase, in this case done using 2-opt principle to find neighbours. Notice also that the score function is updated every time removing the cost of the removed arcs and adding the cost of the added arcs. Here is not reported the whole formula because the line was long and here there are space limits.

The code is the same as before, we check each node reversing part of the solution list. The only difference is on the tabu list update. Here I pop out the first element of it and add the new solution to the end of the list.

```
if itera ==50:
    totn=0
    while totn<50:
        f_trash=9999
        for i in range(1, len(x)-4):
            for j in range(i+1, len(x)-3):
                for k in range(j+1,len(x)-2):
                    x_new = x_temp[:i] + list(reversed(x_temp[i:j+1])) +list(re
                    f_new= f_temp - dis_mat[x_temp[i-1]-1,x_temp[i]-1] - dis_ma
                    if f_new<f_trash and x_new not in memory:
                        f_trash=f_new
                        f_temp1=f_new
                        x_temp1=x_new
        f_temp=f_temp1
        x_temp=x_temp1
        memory.append(x_temp)
        memory.pop(0)
        if f_temp<f:
            x=x_temp
            f=f_temp
            itera=0

        totn=totn+1
    itera=itera+1
    diversification_phase= diversification_phase + 1
```

After each intensification iteration I check whether the *itera* counter is exactly equal to 50, if yes I start a diversification phase, in order to move to different solution space. I run 50 steps of diversification phase, done with 3-opt neighbour search, and if I find a new best solution I save it.

After the diversification phase, an intensification phase starts again. If also in the new space solution no better solution is found the algorithm stops (it will end due to 100 iteration without changing the incumbent solution), otherwise it continue to loop.

```
print(" Final solution: "+ str(x))
print("\n Score function takes value: "+str(f))
print("\n Number of iteratio without improvement: "+str(itera))
print("\n Total number of iteration: "+str(tot+diversification_phase*50 ))
print("\n Total number of diversification phases: "+str(diversification_phase))
print("\n ---Tabu Search took %s seconds ---" % (time.time() - start_time))

return {"Solution":x,"ScoreFunc":f,"TotIter":tot+diversification_phase*50,"Dive
```

As last thing the function print the results and return a dictionary containing all the relevant information of the Tabu Search run.

4 Test

In this chapter I'm going to run some test and report some statistics about the two approaches, here I will also highlight the differences of the two and their advantages or disadvantages.

First of all I want to clarify that I will do a comparison of the two approaches but actually given that one is implemented in a commercial software and surely well optimize and the other is implemented by me on Python the comparison could be biased, so on execution time there could be some more uncertainty to solve before considering the comparison exaustive.

Said that we can start analysing the two proposed methods.

Generating relevant instances

In order to run analysis I needed some instances to solve, so I created a function that generate some points on a two dimension plane. The function *dist_holes* will be used to generate all the test instances. *dist_holes*, given the shape of the plane, the number of holes to practice, the number of centers to create and the dispersion value, it extracts *num_centers* "center" points uniformly on the plane and than it extracts $floor((num\_holes)/(num\_centers))$ point for each center.

The points are extracted on the plane defined by $center \pm disp$ both for x and y axis.

So actualy *dist_holes* extract some points inside the square of dimension ($2dist$ x $2dist$) with some random center. This is a meaningfull way to create instances able to represent the real problem of drilling boards, infact here I suppose that there are some areas in which we need to drill and other where we don't need or also areas where it's more probable we need to drill. The holes I create are also correlated one with the other, they are on a specific area, they are not simply uniformly distributed on the board.

Below I report the function to create the point position and the relative distance matrix.

```
def dist_holes(shape,num_holes,num_centers,disp):
    x_c= np.random.uniform(low=disp,high=shape[0]-disp,size=num_centers)
    y_c= np.random.uniform(low=disp,high=shape[1]-disp,size=num_centers)

    x=np.empty([int(num_holes - (num_centers-1)*np.floor(num_holes/num_centers)),2

    x_p= np.random.uniform(low=x_c[num_centers-1]-disp,high=x_c[num_centers-1]+dis
    y_p= np.random.uniform(low=y_c[num_centers-1]-disp,high=y_c[num_centers-1]+dis


    x[:,0]=x_p
    x[:,1]=y_p

    for i in range(0,num_centers-1):
        x_p=np.random.uniform(low=x_c[i]-disp,high=x_c[i]+disp,size=int(np.floor(r
        y_p=np.random.uniform(low=y_c[i]-disp,high=y_c[i]+disp,size=int(np.floor(r

        xtemp=np.concatenate(([x_p],[y_p]), axis=0)
        xtemp=np.transpose(xtemp)

        x=np.concatenate((x,xtemp),axis=0)



    dat=pd.DataFrame(x)
    temp=scipy.spatial.distance.pdist(dat)
    dis_mat=scipy.spatial.distance.squareform(temp)
    return dis_mat,dat
```

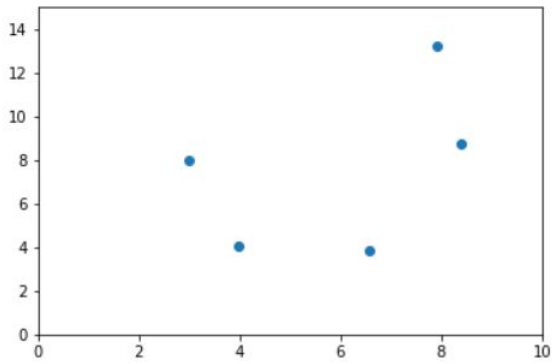
As it's possible to see, the function above returns the distance matrix and the points position of the generated drilling board problem.

In order to generate random starting point for the Tabu Search algorithm I wrote the function *random_start* that generates a random starting Hemiltonian cycle. This will be the input starting solution for the Tabu Search. Below the correspondent code.

```
def random_start(num_holes):
    pro=np.random.permutation(np.array(range(0,num_holes))+1)
    x=np.append(pro,pro[0])
    x_temp=list(x)
    return x_temp
```

Comparison on k-holes

5 Holes I will start the actual test of the two methods giving them a TSP problem with five nodes to visit and using a tabulist long 5. The nodes cost matrix is computed as described before and the same instance is given to both methods. Below an image of the board I'm considering with the holes position indicated by points.



Here the shape of the board is $(10, 15)$ and the analytics I'm going to present are collected over a sample of 10 random initialization of my Tabu Search algorithm and the OPL time execution is estimated as the mean of five trials over the instance.

So I report below the table with the results concerning the 5 holes drilling board problem

Metric	Mean	Variance	Min	Max	OPL	Tabu/OPL
Score Function	23.556696	5.048710e-30	23.556696	23.556696	23.557	0.999987
Tot Iterations	158.900000	8.820900e+02	149.000000	248.000000	82.000	1.937805
Time	0.006293	2.610587e-06	0.004978	0.009994	0.370	0.017009

The first important news is that Tabu Search seems to work, in fact it reaches a really good result, actually given that the ratio $\frac{tabusearch\_scorefunction}{OPL\_scorefunction}$ is lower than one and OPL finds one of the global minima it means that also Tabu Search reached the minimum (the ratio results < 1 only due to the fact that OPL only gives 3 decimals for the score function).

Another important fact is that Tabu Search seems to have a really low

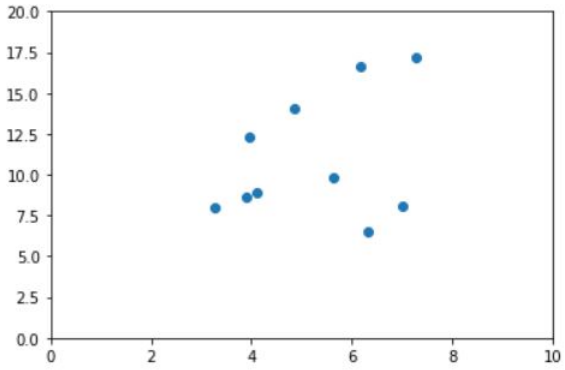
variance, actually it's equal to zero, so in this situation it has reached the minimum in each one of the ten run.

The execution time seems to be good in both cases but the ratio between the result of the two methods shows that Tabu Search takes less then the 2% of the time that OPL takes to reach the minimum.

From the point of view of the iteration seems to be vantagious OPL, Tabu Search infact on average doubles the number of iterations OPL does.

10 Holes Let's try now to double the number of holes to practice and modify a bit the board. So now we are dealing with a TSP problem with 10 nodes, a good choise could be the one of increasing the tabu list length. Indeed here we have more neighbours and we can than increase the length of the tabu list without increasing the risk of being stucked.

Below the image of the board and the holes position.



Here the board has shape $(10, 20)$ and the analytics below are computed as for the 5 holes case.

Metric	Mean	Variance	Min	Max	OPL	Tabu/OPL
Score Function	25.527737	6.813865e-26	25.527737	25.527737	25.528	0.999990
Tot Iterations	781.200000	1.382564e+05	149.000000	1100.000000	411.000	1.900730
Time	0.214677	1.304089e-02	0.048972	0.464729	0.750	0.286236

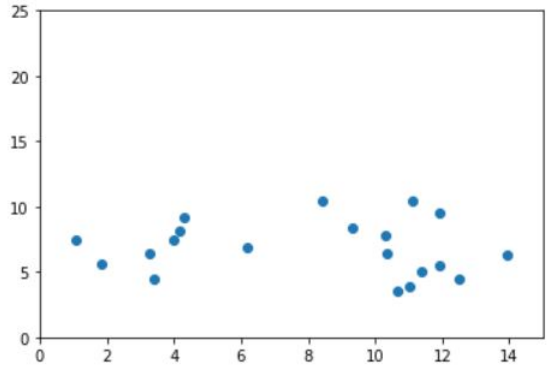
Also with 10 nodes we observe a great result: Tabu Search on average reaches

the global minimum. The variance of the score function is zero and this is a good news.

As for the 5 nodes case, also here we see that Tabu Search takes much less time to converge with respect to OPL, it takes on average 0.2 seconds agains the 0.75 of OPL.

The number of iteration of Tabu Search remains almost twice the one of OPL but as one can notice there is a quite big variance on that metric. This suggest that we have speed convergence for particularly good starting solution while instead we need different diversification phases for particularly bad startng point.

20 Holes Let's double again the number of holes to practice and change again the board.



Now we consider a board with shape $(15, 25)$ and 20 holes needs to be done on it. Below is reported the results relative to this board using the two proposed methods where for the heuristic method I increares the tabu list, setting its length to 10.

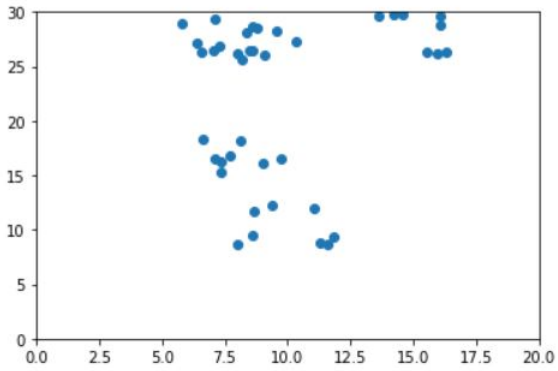
Metric	Mean	Variance	Min	Max	OPL	Tabu/OPL
Score Function	39.911056	0.538075	39.558191	42.019043	39.558	1.008925
Tot Iterations	415.600000	80548.040000	152.000000	1000.000000	1034.000	0.401934
Time	1.180122	0.559303	0.484739	2.312671	1.690	0.698297

Starting from the score function obtained by the heuristic method we can say that it has low variance and its mean is really similar to the optimum obtained by OPL. The distance fom the optimum, so the error committed taking the heuristic solution instead of the exact one, is on the order of 1% so really small.

As in all the other cases the iteration maesure is quite variable due to the random start and this infuences the execution time.

This time, differently from the other cases we don't have impressive difference in OPL and Tabu Search execution time, and also considering that the average execution time here is 1.18 seconds we actually can't consider the variance small. As said before this is probably due to the shape of the score function that impose for some starting points long path before reaching good solutions.

40 Holes I doubled again the number of holes to practice, working on a different board shaped $(20, 30)$. Below a picture to understand how the board would be.



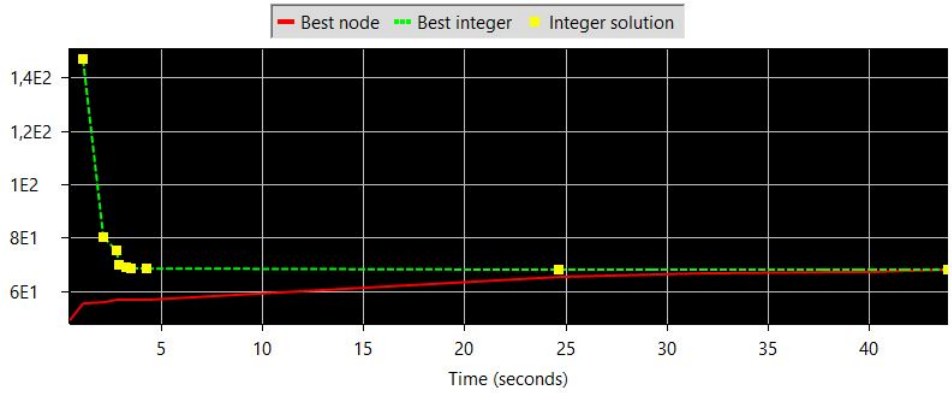
Metric	Mean	Variance	Min	Max	OPL	Tabu/OPL
Score Function	68.525362	0.790623	68.010082	70.640491	68.01	1.007578
Tot Iterations	345.800000	52930.760000	175.000000	1000.000000	448439.00	0.000771
Time	7.680115	9.583521	4.367494	13.188429	47.31	0.162336

As it's clear from the table above that the heuristic methods becomes extreamly useful when the number of holes to practice increases. Here infact we

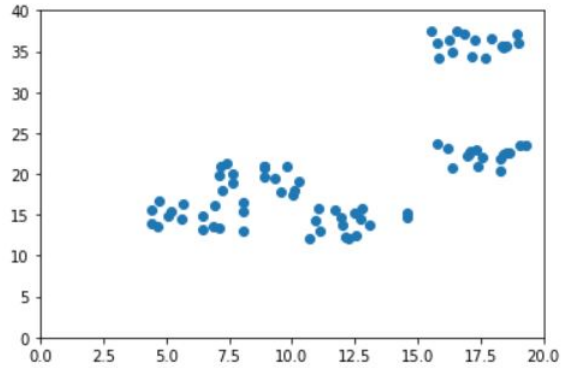
can see that tabu search stops after 7.68 seconds while OPL computation take almost 40 seconds more. This is important, but what is even more surprising is the fact that the distance of the average Tabu Search score function from the optimal one is less than 1%.

At this point also another things becomes evident, average total number of iteration made by the heuristic method is much less then the total number of iteration made by OPL, less then the 0.08% of that.

One important thing to notice is that even if OPL takes on average 47.31 seconds almost half of the running time is spent prooving that the current incumbent solution is actually the optimal one. The OPL convergence plot clearly shows this behaviour.



80 Holes The last test I will run is on a 80 holes drilling board problem. Below a plot to visually inspect the holes position.



Metric	Mean	Variance	Min	Max	OPL	Tabu/OPL
Score Function	103.824973	1.201043	102.156752	105.341333	100.56	1.032468
Tot Iterations	357.700000	8003.410000	228.000000	585.000000	3121627.00	0.000115
Time	91.315150	1582.227582	40.935524	199.051294	450.15	0.202855

The situation in this case is similar to the one before, heuristic metod is much faster without losing too much in terms of score function. Also the number of iteration show a big difference in the two approaches.

5 Conclusions

We analysed two approaches to solve the same problem, one gives exact solution while the other gives no guarantee of goodness.

Tabu Search with alternated two-obs and three-obs local search, as litterature suggests, seems to give good results and with the adopted parameters it seems to explore the space effectively. Actually I put strong stopping criterion in order to favour good results in terms of score function. If one need to have results fast and is ok with lower accuracy should modify the stopping criterion in order to favour the speed of the algorithm.

Even if we are not guaranteed with optimality of Tabu Search results, evidences show that good solution are obtained.

Exact method instead guarantee optimality of the proposed solution, but they have the disadvantage of the long running time. Anyway the test I run show that for small instances, even if TSP problem is an np-hard problem, exact methods return solutions in a reasonable amount of time.

So in general the good practice to follow when dealing with this problem, but the same could be generalised to all problems, one should go first to exact method and making it run for the available time, if that's not enougth for convergence then heuristic can help.