Metaheuristics TP2: The Tabu-Search on a Quadratic Assignment Problem

Ning 11 October 2019

In [1]:
```python
import random
import numpy as np
import math
import sys
import jdc

sys.setrecursionlimit(20000) #WARNING: python recursion issue

class QuadraticAssignment:
    pass
```

Please note that all code in the report are snippets, and the working code in its entirety is attached in qap.py

Introduction

The goal of the assignment was to implement a Tabu-Search metaheuristic on a Quadratic Assignment Problem. The Quadratic Assignment Problem is an optimization problem that is concerned with the optimal placement of facilities on locations in order to minimize the overall cost of the system (placement, configuration), which is determined by the equation:

$$argmin(I) = \sum w * d$$

This equation is a simplified in order to give a better overview. The products of the flow between facilities (denoted by w) and the distance of that flow (denoted by d) is summed together in order to find the cost of that configuration.

Theory and Methodology

One of the ways in which the Quadratic Assignment Problem (QAP) can be optimized is to use the Tabu-Search Algorithm. The Tabu-Search algorithm in context of solving our QAP has these basic principles:

1. An initial starting configuration is generated at random.
2. The best non-taboo solution is selected at each iteration. (Short-Term Memory)
3. If a move has not been made in the last u iterations, then it will be made. (Long-Term Memory)
4. Once the algorithm has reached a designated maximum number of iterations, it will terminate.

We will look at a brief overview of the implementation process (with code), but before looking at the implementation of the metaheuristic, we will discuss the process for calculating fitness (cost) and how it can be best thought about.

Calculating the Fitness Function (Cost Function)

In [2]:
```
%%add_to QuadraticAssignment

def calculate_fitness(self, N, D, W, configuration):
        sum = 0
        #triangle is 2x for loop i,j with j = i+1
        for i in range(0, N):
            for j in range(i+1, N):
                sum += W[configuration[i][1]][configuration[j][1]]*D[configura
tion[i][0]][configuration[j][0]]
        return sum*2
```

The equation for the cost function is given in the introduction, but an intuitive way to think about the calculation is to think about the two matrices (w and d) to be plug-boards, and the multiplication to be the wires that plug into both boards. The optimization is thus a question of what configuration to plug the wires into the board such that the total cost is minimal.

However, even in our analogy, this calculation is expensive to do at every iteration. Therefore, there needs to be some way to calculate whether or not a swap of our configuration improves or degrades the optimality. This implementation is given below.

In [3]:
```
%%add_to QuadraticAssignment

def partial_sum(self, N, D, W, swap):
        partial = 0
        for k in range(0,N):
            if(k!=swap[0][0] and k!=swap[1][0]):
                partial += (W[swap[1][1]][k]-W[swap[0][1]][k])*(D[swap[0][0]]
[k]-D[swap[1][0]][k])

        return partial  * 2
```

The intuition here is that when you make a swap (of two facilities), you affect the calculation of every other connection in the board related to the facilities that we swapped (going by our plug-board analogy). Thus, we can calculate the difference of the fitness (cost) of the configuration by seeing how every other value has changed. Here, k is representative of every other facility connected to the facilities in question (i and j). But since swapping two facilities don't affect the calculation associated with those two facilities (distance between those two facilities will be the same, as well as the flow between those two facilities) and instead affects everything else connected to i and j, we introduce the condition that k cannot be equal to i and j (denoted swap in our code). This allows us to make the calculation to evaluate improvement of our cost-function without calculating the entire configuration.

Initial Starting Configuration is Generated at Random

In [4]:
```
%%add_to QuadraticAssignment

#generate initial solution
self.placement = [None]*self.N
for i in range(0, self.N):
    self.placement[i] = i
random.shuffle(self.placement) #random initial state
for i in range(0, self.N):
    self.placement[i] = (i, self.placement[i]) #put it in (location, facility)
forme.
```

For keeping track of which facility is placed on which location, we chose to create a list of tuples, organized as (location, facility). This list of tuples then has its facilities values shuffled randomly at the start of the initialization of each run.

The Best Non-Taboo Solution is Selected (Short Term Memory)

In [5]:
```
%%add_to QuadraticAssignment

#choose the best non-tabu neighbor, first run is okay
if(iter == 0 or (taboo[pot_swap[0][0]][pot_swap[1][1]] != taboo[pot_swap[1]
[0]][pot_swap[0][1]])): #if a valid placement
    if(iter == 0 or (iter > taboo[pot_swap[0][0]][pot_swap[1][1]] and iter > t
aboo[pot_swap[1][0]][pot_swap[0][1]])): #from loc 0-> fac 1, and loc 1-> fac 0
        #swap is accepted
        taboo = self.add_to_tabu(N, iter, tenure, taboo, pot_swap) #adjust tab
u list

        #swaps fac from roll1 with fac from roll2 while maintaining respective
loc in config
        (tloc_a, tfac_a) = configuration[pot_swap[0][0]]
        (tloc_b, tfac_b) = configuration[pot_swap[1][0]]
        configuration[pot_swap[0][0]] = (tloc_a, tfac_b)
        configuration[pot_swap[1][0]] = (tloc_b, tfac_a)
```

The Tabu-list is a NxN matrix, with the rows represented by the location, and the columns represented by the facilities. This setup allows us to keep track of whether or not a facility i has been placed on a given location r since the past t iterations. There are two things to note here:

1. We care about whether or not facility i has been placed onto location r, but we need to be able to allow swaps that involve facility i (as long as it is not placed on location r). Therefore we must check that the value at (i, r) is different from (j, s) before we consider the value of that entry in the Tabu-list.
2. The Tabu-list values represent the iteration count until we are allowed to make the given swap again. This Tabu-list "tenure" (denoted &l&) is defined to be $l = t + k$ where t is varied over $\{1, 0.5n, 0.9n\}$ according to our assignment requirements, and where k is some random value.

This keeping of the Tabu-list based on whether or not the iteration value is past the tabu-value is referred to as the "Short Term Memory" segment of the Tabu-Search algorithm. The implementation of this type of list is to allow for exploration of the search space by taking moves that can sub-optimal, but prevents backtracking onto moves that were recently made. The tabu-tenure can be thought of as how short (or long) the Short-Term-Memory is, since it determines how many iterations it takes before a recent move is considered legal again.

The implementation of adding of values to the Tabu-list (described in point 2) is given below.

In [6]:
```
%%add_to QuadraticAssignment

def add_to_tabu(self, N, iter, tenure, taboo, swap):
        t = tenure*N #taboo tenure l, defined by the notes
        if (tenure==1):
            t = 1
        #edit the taboo value
        tab = t + random.randint(0,N)
        taboo[swap[0][0]][swap[0][1]] = tab  #loc 0 -> fac 0
        taboo[swap[1][0]][swap[1][1]] = tab #loc 1 -> fac 1
        taboo[swap[0][0]][swap[1][1]] = tab #loc 0 -> fac 1
        taboo[swap[1][0]][swap[0][1]] = tab #loc 1 -> fac 0

        return taboo
```

If a Move Has Not Been Made in the Last u Iterations It Will Be Made (Long-Term Memory)

In [7]:
```
%%add_to QuadraticAssignment

def long_term_memory(self, N, iter, taboo):
    '''
    simulate the long-term memory of a tabu-list
    '''
    smallest = math.inf #so that anything will be smaller on the first run
    second_smallest = math.inf
    old_timer = [(0,0),(0,0)] #placeholder for the result

    for loc in range(0, len(taboo)): #search the entire tabu list for smallest 
value
        for fac in range(0, len(taboo)):
            if( iter-taboo[loc][fac] >= (N*N) ): #if move has not been made la
st u iterations
                if taboo[loc][fac] < smallest:
                    #set new smallest
                    smallest = taboo[loc][fac]
                    old_timer[0] = (loc,fac)
                if (taboo[loc][fac] < second_smallest) and (taboo[loc][fac] >
 smallest):
                    #set new second_smallest
                    second_smallest = taboo[loc][fac]
                    old_timer[1] = (loc,fac)

    return old_timer #the swap that we need to make according to long-term mem
ory
```

As seen previously, the Tabu-list allows for exploration of the search space by allowing for moves that can might be sub-optimal (yet does not allow backtracking onto recent moves). However, this type of prevention of backtracking will add a "bias" to the movement, since it will favor exploration away from the direction where we originated from. Thus, a Long-Term Memory system is created in order to force the algorithm to make a move if the move has not been made within the last u iterations, where u is defined by our assignment to be N^2. The implementation of this is based on first determining what value is "the last u iterations" by subtracting the Tabu-list entry from the iterations, and seeing if it is greater than u. Then, finding the value in the tabu-list (location and facility) that pairs with another value (another location and facility) where these two entries have the lowest values in the Tabu-list will give us the value that has not been changed (and thus the move that has not been made) for the longest time. The Long-Term memory will allow our algorithm to "correct" the bias in a way that favors a more-fair exploration of the search space.

Results

Results are in the format (Starting Configuration Fitness, Ending Configuration Fitness)

Trial 1 Run (Tenure == 1)

```
 (838, 794), (758, 784), (734, 714), (782, 778), (826, 930), (718, 750), (796, 816), (830, 
822), (836, 846), (704, 842)
```

The Mean is: 25.4

The Standard Deviation is: 35.607

Trial 2 Run (Tenure == 0.5n)

```
 (874, 828), (766, 802), (844, 844), (766, 858), (786, 820), (832, 848), (852, 810), (726, 
732), (804, 772), (864, 764)
```

The Mean is: -3.6

The Standard Deviation is: 30.484

Trial 3 Run (Tenure == 0.9n)

```
 (800, 798), (840, 828), (796, 726), (778, 714), (766, 800), (832, 846), (828, 852), (906, 
774), (756, 782), (838, 852)
```

The Mean is: -18.2

The Standard Deviation is: 10.182

Discussion

From our results, we can see an improvement in our optimality when we increase our tabu-tenure $\{1, 0.5n, 0.9n\}$ with the best results from tabu-tenure 0.9n. The results are also more consistent (lower standard deviation) as we increase our tabu-tenure.

We expect to see the results improve with a higher tabu-tenure since the tabu-tenure is the factor preventing us from backtracking on our results, which allows us to explore more of the search space. This is why a tabu-tenure of 1 provides poor results because the metaheuristic becomes not much more different than a local-search algorithm that allows for sub-optimal moves. As we increase the tabu-tenure, we see an improvement due to our ability to explore the search space more by preventing backtracking. However, we can expect a degredation of our optimality as the tabu-tenure increases too high, since this causes the problem of having a short-term memory that is too long, leading to dead-ends or inability to return to optimal areas of the search-space (too much exploration, not enough exploitation). The long-term memory component of our metaheuristic alleviates a little bit of the problem since it allows for us to take moves that we otherwise would not be allowed to take, but does not solve entirely the problem of a short-term memory that's too long since we still can lead ourselves to dead-ends, which effectively terminates the optimization since our implementation does not include a way to get out of dead-ends besides waiting (which essentially causes us to wait out the entire allowed time if our short-term memory is too long).

Additionally, our optimizations are limited by our t_{max} value being low due to how Python handles recursion, as our implementation could not allow 20,000+ iterations due to how Python handles storage of recursive function calls. However, the results do show that the implementation generally improves the cost of the configuration towards optimality, which supports the conclusion that our metaheuristic optimizes our problem.

Conclusion

Our implementation of Tabu-Search for the Quadratic Assignment Problem succeeds in improving the cost of the configuration towards a more optimal solution utilizing a short-term and long-term memory structure.