Tabu Search: Introduction, Implementation and Improvement

Savannah Wang and **Dingwen Xie**

Columbia University, IEOR 4008 Term Project

4/16/2020

Abstract

Local search algorithms are widely used in many optimization problems, such as graph coloring and traveling salesman problems. However, a main disadvantage of local search is that it has the tendency to become stuck in some search spaces.

To avoid that, Tabu Search has a more flexible search behavior by applying some memory based strategies. In this project, we will introduce the idea of Tabu search with its applications on two famous optimization problems and how we improve the performance of it.

1. Introduction

The idea of tabu Search is first proposed in 1986 by Fred W. Glover with the purpose of crossing boundaries while finding optimality instead of treating them as barriers. Tabu Search is a meta-heuristic that guides a local heuristic search procedure to explore the solution space beyond local optimality. [Fred Glover and Rafael Marti 1986] The word, Tabu, is originally a tongan word, and indicates things that cannot be touched because they are sacred. In tabu search, by limiting the candidates movements with some criterion, the search algorithm can be prevented from being stuck in a local optima, and have a chance to explore more search spaces. In other words, Tabu Search allows moves that are harmful for the current status with the purpose of reaching better values for future iterations.

Terminology:
Tabu list: list of forbidden moves, also referred to as tabu moves.
Aspiration criterion: provides exceptions for Tabu restriction.
Tabu classification: to identify which elements from $\mathcal{N}(\boldsymbol{x})$ will not be put into $\mathcal{N}'(\boldsymbol{x})$.
Stopping Criterion: output the solution when the stopping criterion is reached.

Overview of Simple Tabu Search:
The simple TS starts with a given initial solution, $\boldsymbol{x}curr$, which can commonly be a random generated solution or an output from other local search algorithms. Recording this solution as current best solution $\boldsymbol{x}best$. Based on the current solution we have, we can construct its

neighborhood, $\mathcal{N}'(\boldsymbol{x})$, with adjacent solutions that can be reach from $\boldsymbol{x}$, by following the Tabu classification. Pick the best solution $\boldsymbol{x}'$, and denotes it as $\boldsymbol{x}curr$. If our current solution, $\boldsymbol{x}curr$, gives a better performance for objective function, we update $\boldsymbol{x}curr$ as $\boldsymbol{x}best$ and update the Tabu list accordingly. Then, keep performing the steps describe above until we reach some stopping criterion and output the $\boldsymbol{x}best$ as out solution. Figure 1 is the flowchart of the simple Taba Search.

Flowchart:

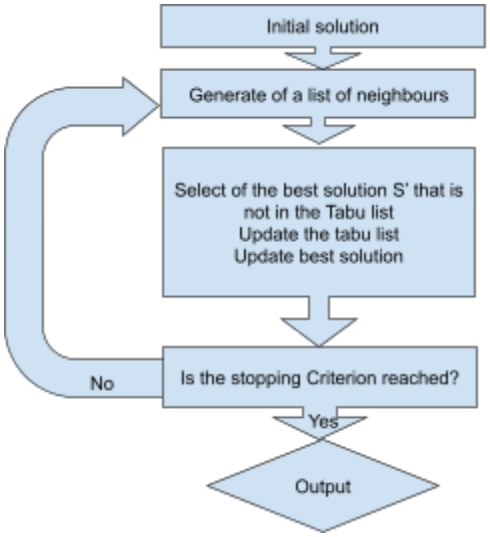
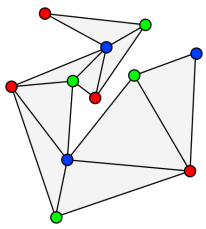


Figure 1

TS explores the neighborhood of current solution $\boldsymbol{x}$ carefully and the solution of the new neighborhood $\mathcal{N}'(\boldsymbol{x})$, which is characteristically a subset of original neighborhood, $\mathcal{N}(\boldsymbol{x})$. For TS, there are some solutions that are highly valuable for some iteration but not contributing much for other iterations, so TS neighborhood is not a static and updates each iteration each interaction. As a result, these neighborhoods are determined through the use of memory structures. Here, the memory structures are roughly categorized into three types: Attributive memory (used for solution candidates), Intermediate-term (used for intensification) and long-term (used for diversification). The attribute memory are widely use to compute the TS neighborhood $\mathcal{N}'(\boldsymbol{x})$, and the most two common used ones are recency based memory and frequency-based memory. Another key factor in TS is to balance between search intensification and diversification. Intensification provides a strategy in cooperation of previous good solutions towards promising areas of the search space; diversification,on the other hand, emphasizes on some infrequently used solutions that drive the search into new regions.

2. Applications



I. Graph Coloring

Given a graph G = (V, E), find a coloring of its nodes V with a minimum number of colors. In other words, find a mapping c: V -> N such that $c(v_1) \neq c(v_2)\ \forall\ v_1v_2 \in E$ *such that* $|\{c(v) : v \in V\}|$ *is minimized.*

Goal: try to find a coloring with a fixed number of color, k, of the given graph, G.
Objective function:
$f(s) = \sum (|E(V_i)| : i = 1, ..., k)$,
$E(V_i)$ *is the collection of edges of G with both endpoints in V.*
(REMARK: all s here can be a not feasible coloring under this setting.)
Basic tabu search with graph coloring:

1) Given the initial solution, *s,* generated by a random assignment of k colors and set *scurr = s, sbest = s.*
2) Generating neighborhood of the current solution, $\mathcal{N}'(s)$, which is other partitions, *s',* of nodes with k subsets of nodes. This neighborhood is generated with the following steps: randomly choose a node, *v,* such that $v \in \{\cup E(V_i): i = 1, ..., k\}$. Suppose $v \in V_i$ (i.e. *v* is colored with ith color), and we change *v* from color i to color j.
3) From $\mathcal{N}'(s)$, pick the best one, *v*, which is changed from color i to j, let *scurr = v:*
   If *scurr* is not in the tabu list: add the pair *(v,i)* into the tabu list. Also update *sbest,* if *scurr* has a better performance than *sbest.*
   If *scurr* is in the tabu list: next iteration
4) Until reach the stopping criterion: the maximum number of iterations or objective function, *f(s)*, equals to zero.

II. Traveling Salesman Problem

Given a weighted graph G = (V, E) with weights w, find a tour T of minimum total weight.

Goal: find a tour that minimizes the the total weights
Basic tabu search with TSP:

1) Initial solution, *s,* is obtained by other algorithms or purely randomly generated, and set *scurr = s, sbest = s.*
2) From the neighborhood, $\mathcal{N}'(s)$, a list that includes all candidate solutions and the total distance of each solution that are produced with 1-1 exchange from the initial solution, *s*.
3) Find the solution, *s',* with shortest distance from the list above, and *scurr = s*:
   If *scurr* is not in the tabu list: add the 1-1 exchanged pair to the tabu list, update *sbest,* if *scurr* has a better performance than *sbest.*
   If yes: next iteration
4) Until reach the maximum number of iteration

3. Methodology

Compare how Tabu search performs on practical problems, we take Traveling Salesman Problem as our experiment objective and implement it with three algorithms: local search (greedy algorithm), basic Tabu search and improved Tabu search.

For our experiment, we use the random graph generator from homework, which input the number of nodes, *n*, and output a dense graph with all weights randomly generated. Take the output from the random graph generator as the input for our experiment, and run the greedy algorithm with it using the greedy algorithm code given from homework.

For the basic tabu search algorithm, we take the solution from greedy described above as our

initial solution input, then find neighbourhoods so that we could swap elements. Next create a tabu list of size 10 to store recently visited nodes in order to prevent cycling, and set a stop condition. In our experiment, we define that the algorithm must stop when achieving max iterations. Let max iterations = # nodes/2 for all graph inputs.

One drawback of basic Tabu search is that we need to manually define a stop criterion: too many iterations may lead to long running time, yet too few iterations may not find the best solution. Our way to improve that is to introduce the idea of Strategic Oscillation.

Strategic oscillation operates by orienting moves in relation to a critical level, as identified by a stage of construction or a chosen interval of functional values. Such a critical level or oscillation boundary often represents a point where the method would normally stop. The process of repeatedly approaching and crossing the critical level from different directions creates an oscillatory behavior, which gives the method its name.

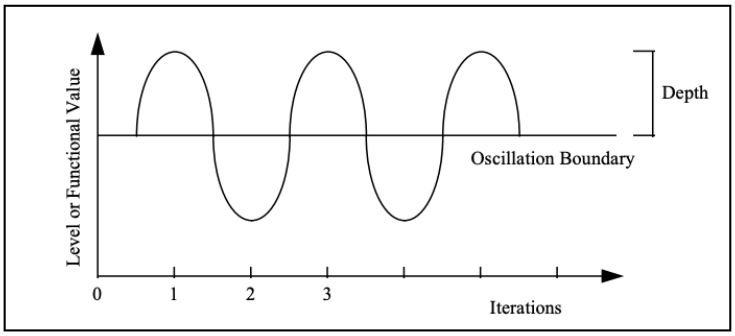


Figure 4-1. Strategic Oscillation

In basic Tabu search, the algorithm will stop when maximum iterations have reached. To adapt this advance design in our case, let the oscillation boundary/critical level be the max iterations. If our current solution is improved, we increase the max iterations by depth of 5; otherwise decrease by 5 since further iterations may not improve significantly. In this way we do not need to predefine a stop condition. Instead, the stop condition changes depending on the improvement we have made so far. While approaching the local optimality (i.e. no further improvement), max iterations will gradually decrease and finally leave the loop. This also prevents endless loop, because the algorithm always computes equivalent or smaller cost in each iteration.

Pseudocode for oscillation approach:

while 1< iters:
    If solution improved: {iters +5}
    Else: {iters - 5}

For detailed implementation, a piece of python code is attached at the end of this report. The result table is attached below.

Results:

10 Nodes

Alg Name	Tabu Iterrations	Best Cost	Time /s
Greedy	/	2.43	0.0001
Tabu	5	2.1	0.0062
SO Tabu	6	2.1	0.0081

25 Nodes

Alg Name	Tabu Iterrations	Best Cost	Time /s
Greedy	/	5.01	0.0008
Tabu	13	4.24	0.4363
SO Tabu	6	4.24	0.1447

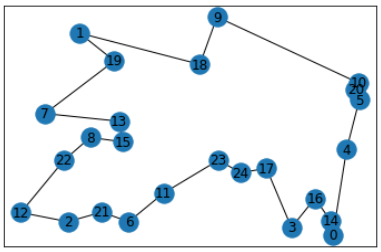
50 Nodes

Alg Name	Tabu Iterrations	Best Cost	Time /s
Greedy	/	6.64	0.0027
Tabu	25	5.98	11.5961
SO Tabu	12	5.98	5.0487

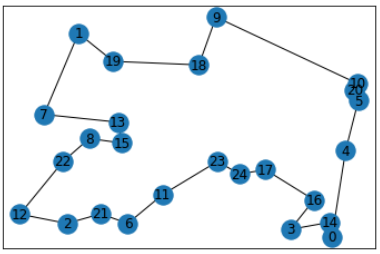
100 Nodes

Alg Name	Tabu Iterrations	Best Cost	Time /s
Greedy	/	9.59	0.0168
Tabu	50	8.64	389.9804
SO Tabu	8	8.64	55.1098

Example plot: (25 Nodes)



Greedy output



Tabu output

4. Conclusions

In general, tabu search does not guarantee a global optimum, but is able to escape from the local optimum by accepting non-improving solutions. Another drawback is that it always takes much longer running time than greedy, and the stopping criterion has to be determined manually most of the time. To deal with its run time, we can use advanced design of Strategic Oscillation. Based on our experiment above, with a variety number of nodes, Tabu Search algorithms perform

better than Greedy Algorithm in searching for the best result, and SO Tabu Search finds the same result much faster than the basic Tabu Search.

In the previous experiment, we used a fixed oscillation depth of 5 for all graphs. To further improve the efficiency of Tabu search, a potential improvement for Strategic Oscillation is to dynamically optimize the depth, so that the algorithm will converge to the final output more quickly.

5.References and Appendix

References:

Tabu Search, by Fred Glover and Manuel Laguna, Kluwer Academic Publishers, 2002.

Hertz, A., and D. De Werra. "Using Tabu Search Techniques for Graph Coloring." *Computing*, vol. 39, no. 4, 1987, pp. 345–351., doi:10.1007/bf02239976.

Python code for Tabu search and SO Tabu search:

```
def get_neighbor(G):
    neighbor_dict={}
    for n in list(G.nodes()):
        temp=[]
        n_neighbor=list(G.neighbors(n))
        for i in n_neighbor:
            w=G[n][i]['weight']
            temp.append([i,w])
        neighbor_dict[n]=temp
     return neighbor_dict


def find_neighborhood(solution, dict_of_neighbours):
    neighborhood_of_solution = []
    for n in solution[1:-1]:
        idx1 = solution.index(n)
        for kn in solution[1:-1]:
            idx2 = solution.index(kn)
            if n == kn:
                continue
            _tmp = copy.deepcopy(solution)
            _tmp[idx1] = kn
            _tmp[idx2] = n
```

```
            distance = 0
            for k in _tmp[:-1]:
                next_node = _tmp[_tmp.index(k) + 1]
                for i in dict_of_neighbours[k]:
                    if i[0] == next_node:
                        distance = distance + i[1]
            _tmp.append(distance)
            if _tmp not in neighborhood_of_solution:
                neighborhood_of_solution.append(_tmp)
    indexOfLastItemInTheList = len(neighborhood_of_solution[0]) - 1
    neighborhood_of_solution.sort(key=lambda x: x[indexOfLastItemInTheList])
    return neighborhood_of_solution

def tabu_search(first_solution, dict_of_neighbours, iters, size):
    start_time = time.time()
    count=1
    solution=first_solution[1]
    tabu_list=[]
    best_cost=first_solution[0]
    ultimate_best_cost = solution
    while count <= iters:
        neighborhood = find_neighborhood(solution, dict_of_neighbours)
        index_of_best_solution = 0
        best_solution = neighborhood[index_of_best_solution]
        best_cost_index = len(best_solution) - 1
        found = False
        while not found:
            i = 0
            while i < len(best_solution):

                if best_solution[i] != solution[i]:
                    first_exchange_node = best_solution[i]
                    second_exchange_node = solution[i]
                    break
                i = i + 1
            if [first_exchange_node, second_exchange_node] not in tabu_list and [
                second_exchange_node,
                first_exchange_node,
            ] not in tabu_list:
                tabu_list.append([first_exchange_node, second_exchange_node])
                found = True
                solution = best_solution[:-1]
                cost = neighborhood[index_of_best_solution][best_cost_index]
                if cost < best_cost:
                    best_cost = cost
                    ultimate_best_solution = solution
```

```
            else:
                index_of_best_solution = index_of_best_solution + 1
                best_solution = neighborhood[index_of_best_solution]
        if len(tabu_list) >= size:
            tabu_list.pop(0)
        count = count + 1
    print("Running time of Tabu Search: %s seconds" % (time.time() - start_time))
    return best_cost, ultimate_best_solution


def SO_tabu_search(first_solution, dict_of_neighbours,  size):
    start_time = time.time()
    count=1
    iters=15
    solution=first_solution[1]
    tabu_list=[]
    best_cost=first_solution[0]
    ultimate_best_cost = solution
    prev_cost=best_cost
    while 1 <= iters:
        neighborhood = find_neighborhood(solution, dict_of_neighbours)
        index_of_best_solution = 0
        best_solution = neighborhood[index_of_best_solution]
        best_cost_index = len(best_solution) - 1
        found = False
        while not found:
            i = 0
            while i < len(best_solution):
                if best_solution[i] != solution[i]:
                    first_exchange_node = best_solution[i]
                    second_exchange_node = solution[i]
                    break
                i = i + 1
            if [first_exchange_node, second_exchange_node] not in tabu_list and [
                second_exchange_node,
                first_exchange_node,
            ] not in tabu_list:
                tabu_list.append([first_exchange_node, second_exchange_node])
                found = True
                solution = best_solution[:-1]
                cost = neighborhood[index_of_best_solution][best_cost_index]
                if cost < best_cost:
                    best_cost = cost
                    ultimate_best_solution = solution
            else:
                index_of_best_solution = index_of_best_solution + 1
```

```
                best_solution = neighborhood[index_of_best_solution]
        if len(tabu_list) >= size:
            tabu_list.pop(0)
##Strategic oscillation part
        if best_cost<prev_cost:
            iters = iters+5
        else:
            iters = iters-5
        count=count+1
        prev_cost=best_cost
    print("Running time of SO Tabu Search: %s seconds" % (time.time() - start_time))
    print("SO Final iter count = ", count)
    return best_cost, ultimate_best_solution
```