Tabu Search

Prasangsha Ganguly

January 18, 2021

We continue with our objective of escaping local optima. Recall our algorithm for *Hill climbing*. If c is the current node (solution) and n is the next node or solution. Then, $n = \text{Best}(\text{Neighbor}(c))$. That is, we consider all the neighbors of c, then chose the best among the neighbors. Say, that is n. Then we check if the objective of the node n is better than c. If that is true, then we move to node n.

Algorithm 1: Hill Climbing

```
begin
1   while True do
2       n ← Best(Neighbor(c));
3       if Objective(n) is better than Objective(c) then
4           n ← c;
```

The main problem is that we may get stuck in a local optima. What hill climbing does is *Exploitation* of the gradient. Basically, it follows the gradient. If there is a better neighbor, it moves to that. If there i no such neighbor, then it gets stuck to the local optima. To escape the local optima, what is needed is *Exploration*. That is, the algorithm needs to go the direction against the heuristic function. That is even if the $Objective(n)$ is worse than $Objective(c)$, we may move to n.

To get rid of the local optima, we will design a deterministic algorithm. In this case, we will generate the neighbors of c upon a *Allowed()* constraint which we will discuss later. Let n be such a neighbor. Then we move to n. That is, in this case we don't check if $Objective(n)$ is better than $Objective(n)$ or not. We move to n without such check. Of course, if there is no such check, then we have to construct a termination criteria.

The termination criteria can be simply time based, like number of iterations or it can be if there is no significant improvements over a number of iterations. We have to keep track of the best solution obtained till now. Even if we move to a worse solution to explore the space, upon termination we will return the best solution till found.

Now we discuss about *Allowed*() constraint and the use of it. Consider the one dimensional maximization problem shown in Fig 1. Say we are currently at a local maxima at point c. Now say, c has two neighbors n and n', out of which n is better. So, we move to the node n. Now, say n has two neighbors c from which it came and m. Now as we are

Algorithm 2: Escaping Local Optima

```
begin
1   while not termination do
2       n ← Best(Allowed(Neighbor(c)));
3       n ← c;
```

maximizing, clearly c is the best neighbor of n. So, we go back to c again. Hence we end up in an infinite loop $c \to n \to c....$ To get rid of this, we don't allow to move back to c from n. Hence, we have only a set of allowed neighbors. This can be implemented using a circular queue. Which we call *Tabu List*. We keep recently explored nodes in the tabu list. The length of the tabu list is called *Tabu tenure*. The tabu tenure is the amount of time (iterations) a particular node (solution) is not allowed to explored again.

The other way to implement this is to keep track of the moves that has been made in near past. That is, not tracking the visited nodes, but the moves made to reach a node. When a move e is performed, then an entry corresponding to the reverse move $\bar{e}$ is entered into the tabu list to restrict a reverse move and going back to the previous node again.

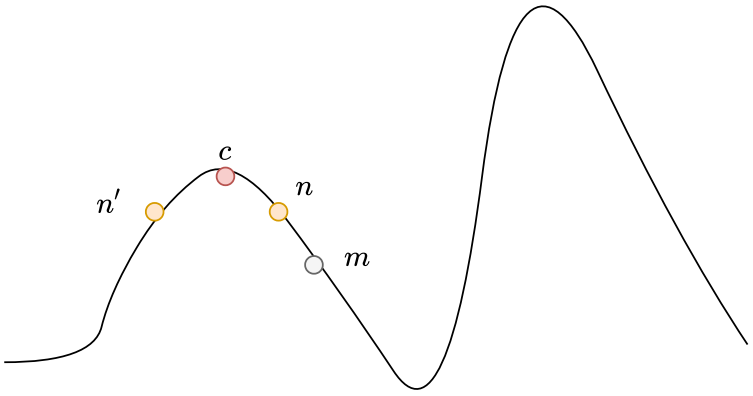

Figure 1: Neighborhood exploration in Tabu search

In the detailed version of the Tabu search algorithm, sometimes an unallowed (taboo) move is considered to make allowed. This is called *Aspiration criteria*. If allowed moves are bad and a taboo (disallowed) move leads to a node n whose objective is better than the current best till found, then allow the taboo move. Note that, a tabu move is only allowed if the objective of a tabu move is better than the best solution till obtained; not better than the current node c.

Another feature which has sometimes been used is called the *Frequency based method* to bias the search direction. So, we have a frequency table which depicts how many times a particular move is made. We can bias our search to explore new space by directing the search towards the moves that are less made. If we investigate, a move is less made because the resulting node is not the best node out of the neighbors. So, that move is not made. We can bias the algorithm to make such a move by incorporating a penalty in the objective function.

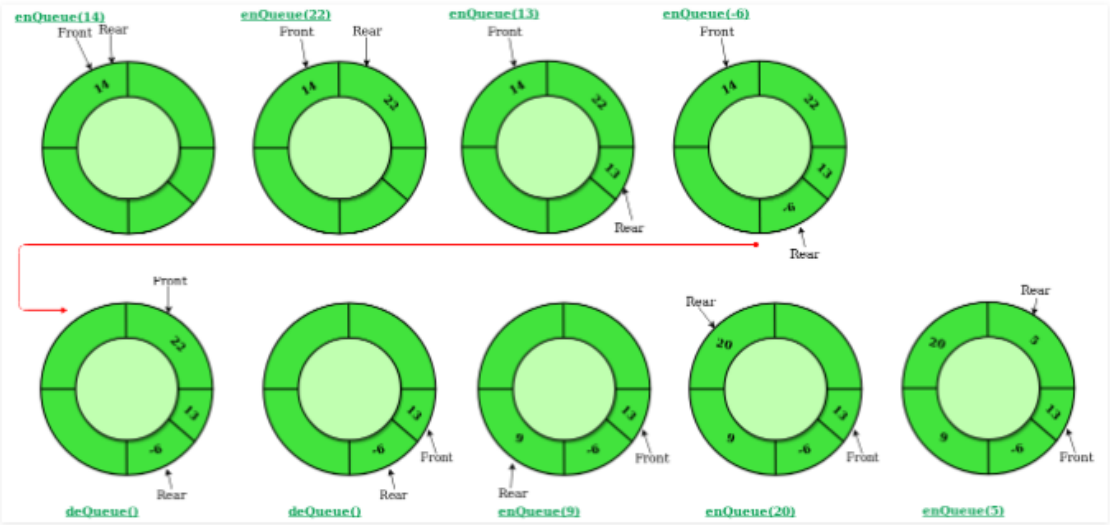

Figure 2: A Circular Queue