Tabu Search

Let us first consider the simplest strategy to optimize a function $f(\mathbf{x})$ over a discrete domain $\mathbf{x} \in D$, the (greedy) *local search*. This can be defined as follows.

Let us define a *neighborhood* $N(\mathbf{x})$ for each $\mathbf{x} \in D$. The domain D, together with the neighborhood definition, is often referred to as the *search space*.

For example, if D is the set of n-dimensional binary vectors, then an example of a neighborhood definition is

$$N(\mathbf{x}) = \{\mathbf{y} \mid \mathbf{y} \text{ differs from } \mathbf{x} \text{ in at most 2 bits}\}. \tag{1}$$

Then the local search can be defined as an iteration in which we always move from the current point to its best neighbor, that is, to the one that offers the most improvement in the function value in the neighborhood. If no improvement is possible in the neighborhood, then the algorithm stops.

If we consider minimization, then local search can be visualized as sliding down on the steepest slope until we reach a local minimum.

The critical problem of greedy local search is that it may easily get stuck in a local optimum, which can be very far from the global optimum.

Tabu search offers an approach that diminishes (but does not eliminate!) the danger of getting stuck in a local optimum. The fundamental principle is summarized below.

- The algorithm maintains a *tabu list* that contains disallowed moves. For example, a simple tabu list can contain the last k visited points, for some constant k.

- The algorithm does local search, but it keeps moving even if a local optimum is reached. Thus, it can "climb out" from a local minimum. This would result, however, in cycling, since whenever it moves away from a local minimum, the next step could take it back if only the improvement is considered. This problem is handled by the rule that the points on the tabu list cannot be chosen as next steps. In the tabu list example mentioned above we can decrease the possibility of cycling by disallowing recently visited points.

- The tabu list can also be more complicated than the example above. The algorithm may also maintain more than one tabu lists. For example, one list may contain the last k visited points, while another list can contain those points that are local minimums visited in the search. A third list can contain, for example, those points from which the search directly moved into a local minimum.

- The rule that the points in the actual tabu list cannnot be visited may be too rigid in certain cases. To make it more flexible, it is usually allowed that under certain conditions a point may be removed from the tabu list. These conditions are called *aspiration criteria.* For example, if with respect to the current position a point on the tabu list offers an improvement larger than a certain treshold, then we may remove the point from the tabu list. Or, if all possible moves from the current position would be prohibited by the tabu list(s), then we can remove from the tabu list the point that offers the best move. In this way we can avoid getting "paralyzed" by the tabu list.

- *Stopping criterion.* The search can, in principle, be continued indefinitely. A possible stopping criterion can be obtained in the following way. Let us keep (and refresh after each iteration) the best value that has been achieved so far. If during a certain number of iterations no improvement is found relative to the earlier best value, then we can stop the algorithm.

Advantages of Tabu Search

- Improves greedy local search by avoiding getting trapped in a local optimum too early.
- Offers flexibility by the possibility of adjusting the tabu list structure and rules to the specific problem. If this is done well, good results can be achieved.

Disadvantages of Tabu Search

- Offers only heuristic improvement, but does not *guarantee* anything in a strict sense: there is no guaranteed convergence to a global optimum (as opposed to simulated annealing).
- Usually requires heavy problem-specific adjustment and there are no general rules that would tell how to do it, much is left to intuition. Therefore, it often involves a significant algorithm design challenge.

A critical step: designing a good search space

Recall that the search space consists of an underlying domain, and a neighborhood definition. The domain is usually given, as it is the set of the possible objects in which we search an optimal one. But the neighborhood structure is up to us to choose.

What can we expect from a good neighborhood definition?

- The arising neighborhoods are not too large, so that they can be searched relatively easily.
- At the same time, the neighborhoods are also not too small, so that they do not overly restrict the choices.
- We expect that the search space is *connected*, that is, every point is reachable from every other one, via a sequence of neighbor to neighbor moves. This is important, since otherwise it may be impossible to reach the optimum simply because it may not be reachable from the initial position at all.

In some cases these conditions are easily satisfied, see, e.g., the introductory example of this lecture note, where the neighborhood is defined by (1). But in other cases it may be somewhat harder to design the "right" search space.

Exercises

1. Assume we an n-node complete graph, with non-negative weights on the edges. The weights represent link delays. We want to design a network that has a tree topology, spanning all the nodes, such that the average delay between two uniformly chosen random nodes, when they are connected via a path within the tree, is minimum. Here the delay of a path is the sum of the link delays (edge weights) along the path. That is, we want to select a spanning tree among all spanning trees of the complete graph, such that it is optimal in the above sense. (This problem is known to be NP-hard.)

Task: Design a search space over the set of all spanning trees of the complete graph, such that the neighborhood definition satisfies the above presented three conditions for a good neighborhood structure.

2. Assume we have n nodes, and we want to design a ring network connecting all the nodes. The cost of connecting any two given nodes is known, and we look for the minimum cost ring. (This problem is known to be NP-hard.)

Tasks:

a.) Design a search space over the set of all possible rings, such that the neighborhood definition satisfies the above presented three conditions for a good neighborhood structure.

Hint: Use the result from algebra that every permutation can be represented as the superposition of transpositions (transposition: a special permutation in which only two entries are exchanged).

b.) Show that the search space can be designed in a way that, beyond satisfying the three conditions, it is also *regular*. Regularity means that each element (point) of the space has the same number of neighbors. This is usually a desirable property, because it means that all points are equally well connected, none of them is distinguished by having more neighbors than others.