Introduction to Simulated Annealing and Tabu Search

Illinois Institute of Technology
Department of Applied Mathematics

Adam Rumpf
arumpf@hawk.iit.edu

March 14, 2019

1 Introduction

Throughout the following sections, we assume that we are attempting to solve a difficult combinatorial minimization problem. Specifically, we are attempting to minimize a function $f : X \to \mathbb{R}$. The objective value of any given feasible solution $x \in X$ is $f(x)$.

Both of the following algorithms are based on the idea of local search, in which case we define a neighborhood $N(x) \subset X$ around each feasible solution $x \in X$. If X is a set of binary vectors, then a common choice for $N(x)$ is to simply be the set of binary vectors that differ from x in exactly one coordinate (i.e. Hamming distance 1), but depending on the problem structure, a different notion of neighbors may be appropriate.

A standard local search (like the hill climb algorithm) simply consists of beginning with an initial solution $x \in X$. During each iteration of the algorithm, we find the current solution's best neighbor and move to it. Once no neighbors offer an improvement, we are at a local optimum, which is the best that a pure local search can hope to achieve. There are a few basic tricks, like random restarts, that may help to expand the search area. The following is a straightforward implementation of local search.

Generic Local Search Algorithm

1. Choose an initial solution $x \in X$.
2. Search the neighborhood for a candidate $y \in N(x)$ such that $f(y) < f(x)$.
   (a) If we find such a candidate, then set $x \leftarrow y$ and repeat Step 2.
   (b) If we find no such candidate, then x is locally optimal and we can stop.

2 Simulated Annealing

The basic idea of simulated annealing (SA) is similar to a local search, but we introduce randomness to attempt to escape from local optima. Instead of always selecting the best neighbor to move to, we select a random neighbor. If it is better, we always move to it, and if it is worse, we may still move to it anyways with a small *acceptance probability* P. The probability of accepting a suboptimal solution depends on the difference in objective values, and on a parameter T, called *temperature*, which makes suboptimal moves more likely when it is high and less likely when it is low. The temperature decreases as the algorithm advances according to a *temperature schedule*, making suboptimal moves less likely as the algorithm proceeds. If the temperature schedule and the neighborhoods meet some simple criteria, it is possible to show that the algorithm defines a Markov chain whose steady state is concentrated at the globally optimal solutions.

The following provides a basic description of a SA algorithm.

Generic Simulated Annealing Algorithm

1. Set the initial step counter to 0, set the initial temperature T, and choose an initial solution $x \in X$.
2. Choose a neighbor $y \in N(x)$.
   (a) If $f(y) \leq f(x)$, then make y our candidate solution and proceed to Step 3.
   (b) Otherwise, calculate and apply the acceptance probability $P(x, y, T)$. If we choose to accept, then y is our candidate and we proceed to Step 3. If not, we choose a different neighbor of x and return to Step 2.
3. Set $x \leftarrow y$, making the candidate solution our new current solution. Increment the step counter. Apply the temperature schedule to T.

4. Determine if we have reached the stopping criterion.

   (a) If yes, then stop and return the best solution found so far.

   (b) If no, then go back to Step 2.

The specifics of how to choose a neighbor, how to calculate the acceptance probability, and how to choose when to stop, must all be specified. The only requirement for choosing a neighbor is that we define a proper probability distribution on $N(x)$ for all possible solutions $x \in X$. A common choice of acceptance probability is

$$P(x, y, T) = \begin{cases} \exp\left(-\dfrac{f(y) - f(x)}{T}\right) & \text{if } f(y) > f(x) \\ 1 & \text{otherwise} \end{cases}$$

[1] To explain, if $f(y) \leq f(x)$, meaning that y is at least as good as x, then we always accept y. Otherwise, we may still choose to accept it with probability $\exp(-(f(y) - f(x))/T)$. This value is largest when $f(y)$ and $f(x)$ are similar or when the temperature T is high.

A common choice of temperature schedule is

$$T(t) = \frac{C}{\log t}$$

[1] where t is the current step and C is a constant which must be sufficiently large. Other common methods of updating temperature include defining $T = \frac{1}{t}$ and defining T as a particular fraction of the previous temperature.

As with any metaheuristic, common choices of stopping criterion include simply cutting off after a certain number of iterations, or stopping after the best known solution has not improved by a wide enough margin in a certain amount of time.

3 Tabu Search

Like SA, tabu search (TS) is a modification of local search that attempts to avoid getting trapped at local optima. Rather than allowing a random chance to move to a suboptimal solution, we maintain a list of recently-explored solutions in memory, and force ourselves to choose only solutions not appearing on this list.

There are many variants of TS that use different memory structures in different ways to encode information about the explored solusions' quality, influence, recency, and frequency [2]. Recency and frequency refer to how recently a particular solution was explored, and how frequently it comes up. Quality refers to its objective value. Influence considers the impact of particular pieces of a solution's structure.

Particular TS implementations are often described in terms of their intensification and diversification strategies. Intensification strategies use information obtained during the search to try to search more intelligently for good solutions, and to focus on areas of the search region more likely to hold good solutions. During an intensification stage, we focus our search on neighbors of solutions historically found to be good.

Many TS implementations are based on a steepest descent local search, in which case we attempt to find not just an improved neighbor of our current solution, but actually the best neighbor of our current solution. This may be expensive if calculating f is expensive, or if the neighborhoods are too large. As a result, depending on the problem, we may wish to conduct this neighborhood search based on some criterion other than direct calculation of f.

Different TS implementations also make use of different types of memory, which generally fall into the category of short-term memory and long-term memory. Both serve to somehow affect the neighborhood $N(x)$ used in the local search. We will denote the modified neighborhood as $N^*(x)$. The simplest versions of TS use only short-term memory, in which case we generally have $N^*(x) \subseteq N(x)$. If we also include longer-term memory, then $N^*(x)$ may

include some elements not appearing in $N(x)$ to allow us to return to more distant solutions historically known to be good. How we define $N^*(x)$ essentially defines our specific TS implementation.

A common usage of recency-based memory is to keep track of attributes appearing in recently-visited solutions. We may add whole solutions, or attributes, or combinations of attributes, to the tabu list. The modified neighborhood then excludes solutions having such tabu attributes. Moves that would take us to such a solution are called *tabu moves*.

For example, in the minimum k-tree problem, we are given a weighted graph G, and our goal is to find a tree consisting of k edges of G whose weight is minimized. Typical moves consist of static swaps (maintaining the current vertex set but trading one edge for another) and dynamic swaps (edge swaps that add one vertex and drop another). We might define our tabu structure to include the added/dropped edges involved in the most recent swap. For example, if we have just added an edge, we might make it tabu to drop it in the near future. If we have just dropped one, we might make it tabu to re-add it in the near future. We might include both of those rules, and make each last for a different amount of time.

It may be possible for poorly-planned tabu rules to cause us to miss good solutions. To attempt to avoid this, we define *aspiration criteria*. A simple version of this is to allow ourselves to make a tabu move if it takes us to a solution that is better than any previously known solution. This is called the *improved-best* aspiration criterion. More generally, an aspiration criterion is simply a rule that allows us to occasionally make an otherwise tabu move.

We do not have to use the same set of rules throughout the entire execution of TS. It is common to begin with a simple first-level implementation to simply explore the problem space, and then use that information as the basis of a more sophisticated implementation.

To aid in diversification, it is important to be able to restart the algorithm to search for significantly different solutions than those previously explored. One way to help in this process is through use of *critical event memory*. For example, we might wish to remember every previous solution visited, or maybe just the solutions that were at the time the best known solution, or the solutions used to initialize all of the restarts. One common usage is to back up the search to just before a particular critical event occurred, while making that event and the immediately preceding events tabu, in order to ensure that our search proceeds in a different direction than it did during the first pass.

It should also be noted that, rather than strictly prohibiting tabu moves, we may simply penalize them by adding a penalty term to the objective value resulting from their use. This penalty is usually made to decay over time. This is especially common when using frequency-based memory, since we may allow penalties to stack up if a particular attribute is included in many critical events.

The following provides one particular implementation of a TS algorithm [2].

Simple Tabu Search Algorithm

1. Choose the initial solution $x \in X$. Set the best known solution as x and the best known value as $f(x)$.
2. Calculate the acceptable neighborhood $N^*(x)$.
3. Choose a neighbor $y \in N^*(x)$.
4. Decide if the move from x to y is tabu.
   (a) If no, or if y is the best candiate found so far, then it becomes our new candidate.
   (b) If yes, and y is not the best candidate found so far, then update the tabu list and return to Step 2.
5. Update $x \leftarrow y$, and update the tabu list based on this move.
6. Determine if we have reached the stopping criterion.

(a) If yes, then stop and return the best solution found so far.

(b) If no, then go back to Step 2.

Here are some specific considerations for designing and refining a TS algorithm [2].

- Neighborhoods
  - Decide on how to define neighborhoods, or equivalently how to define allowed moves. Note that larger neighborhoods take longer to search but allow for more diversification.
  - For small enough neighborhoods we can search the entire thing and choose the best (steepest descent). Otherwise we should come up with some systematic way to choose a "good" neighbor. Picking the first random neighbor that gives an improvement is fine, but it is better if we can somehow use the problem structure to preferentially choose candidates likely to be good.
  - It is possible on occasion that the tabu rules will cause all neighbors to be tabu, in which case we usually just choose the "least tabu" neighbor. This is not an issue if we implement tabus as penalties rather than hard rules.
  - One common candidate generations trategy is called Aspiration Plus. We define a threshold of minimum desired move quality and search the neighborhood until we find a move that achieves it. Then we find a set number of additional moves and take the best of the group. We can also specify minimum and maximum numbers of moves to consider, and always ensure that we never end the search too early or keep it going for too long. The aspiration threshold can be dynamically updated during the search, and should typically be lower if we are in a diversification phase.
  - In order to ensure that the neighborhood search constantly produces new solutions, we might keep every possible move in a circular list that we cycle through to ensure we cannot repeat a move too often.
  - In the Elite Candidate List, we maintain a list (usually of fixed length) of the best moves from the current location. Then we cycle through the list until either it is exhausted or we fail to improve by a large enough margin, after which the list is reconstructed. The assumption is that good moves at a certain location will remain good in nearby locations, and it allows us to avoid searching entire neighborhoods too often.
  - If we allow moves that are combinations of elementary moves (for example, ADD plus DROP equals SWAP), then we might consider just considering the best sets of the component moves and then only considering the combinations of the best few.
  - If parallel processing is available, we may keep separate streams of solutions going at the same time. All streams are updated simultaneously, as are the tabu rules. We can carry this on for a limited number of iterations and then take the best stream as the starting point for another branch point.
- Critical Events
  - Have the algorithm backtrack if it has gone for too long without finding any new best solutions. This could involve randomized restart, but it is better to go back to a critical event.
  - A common choice for critical event includes solutions that are better than the preceding and following solution in the TS search history (i.e. valleys in the current objective curve).
  - We don't just have to return to a single critical event. We might be able to find a way to combine them, for example by aggregating or averaging them.
  - When returning to a critical event, we may also add tabu rules related to the current solution to make sure we go with something very different after the restart.

- Tabu Tenure
  - The best decay time for a tabu list depends on the problem and must be determined through experimentation.
  - Generally, more restrictive rules should have shorter tenures.
  - Try starting with small tenures, then lengthen them if needed. You can tell that they are too short if the objective appears to clearly by periodically cycling (this is because shorter tenures tend to keep us closer to local optima). Telling whether they are too long is harder, but could be indicated if we go for too long without seeing any objective improvement (this is because longer tenures force us to keep moving, possibly away from local optima).
  - It is also possible to have the tabu search adjust its own tabu tenures reactively. We can include measures to test for self-cycling and increase the tabu tenures if we suspect self-cycling. We can then decrease the tenures over time. We can also activate diversification measures by temporarily making random moves for a number of iterations proportional to the cycle length.
  - Many implementations converge fastest if their tenure lengths are varied within small ranges during the course of the runtime. This allows the algorithm to alternate between periods of intensification and diversification.
  - We can define random tenures either by giving each rule a random tenure or by giving all rules a particular tenure that is occasionally randomized.
  - Systematic tenure can be used by simply cycling the tenure through an ascending list of values. This provides a simple and natural means of intensification and diversification.
- Aspiration Criteria
  - Aspiration criteria allow us to break the usual tabu rules if doing so is likely to yield a good solution. The most common aspiration criteria, and one used in almost every implementation, is to always pick a move if it is the best known solution. There is also a local version of this.
  - We can also override tabu rules as long as our objective keeps moving in the same direction (increasing or decreasing).
- Long-Term Memory
  - Frequency-based memory keeps track of the attributes that are either (a) added the most often or (b) most often present in the selected solutions. Frequent attributes for good solutions are probable desirable, while frequent attributes for bad solutions are probably not. Frequent solutions in both cases may indicate that we are not exploring enough of the solution space.
  - LTS generally consists of deciding on a definition for "elite solutions", which are the solutions returned to periodically after stagnation. One definition attempts to maximize diversification, and encourages solutions that are distant from each other (in terms of number of moves). Another simply maintains a list of solutions which were the best known at the time. Yet another keeps a list of neighbors which were found to be good but ultimately not chosen during the search.
  - After some time we may choose to lock in components of the solution by holding them constant.
  - In order to better explore the search space, we may wish to temporarily ignore certain constraints. We run the search to continuously improve the solution, even going beyond the usual constraint bounds to achieve a solution better than anything feasible, then we turn back by attempting to minimize either a penalized solution or a constraint satisfaction objective. The overall search process consists of these oscillatory

phases moving into and out of the feasible region. A common way to achieve this is to alternate between considering only ADD moves and then considering only DROP moves, deciding on when to switch based on the desired crossing depth.

– The usual implementation of the elite solution method stores previous attractive solutions (possibly critical events) in memory along with the other TS structures at the time. If our search stagnates for too long, we consider the elite solutions in some order (usually from best to worst, but possibly with multiple passes in the opposite direction with some filtration) and pick one to return to. Since we also saved the TS structures from the first pass, we can make sure to move in a different direction in the second pass to avoid repeating the same search. Alternatively, we can slightly randomize our choices.

References

[1] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, MA, 1997.

[2] P. Pardalos and M. G. Resende. *Handbook of Applied Optimization*. Oxford University Press, Oxford, 2002.