14 Heuristic Algorithms

For all we know, a complete exploration of the search tree, either explicitly or implicitly, might be required to guarantee that an optimal solution is found. When this is impractical, heuristic methods can be used to find a satisfactory, but possibly suboptimal, solution. Heuristics sacrifice solution quality in order to gain computational performance or conceptual simplicity.

14.1 Basic Heuristics for the TSP

A straightforward way of constructing a TSP tour is by starting from an arbitrary vertex, traversing a minimum cost edge to an unvisited vertex until all vertices have been visited, and returning to the initial vertex to complete the tour. This greedy algorithm is known as the *nearest neighbor heuristic* and has an asymptotic complexity of $O(n^2)$, where n is the number of vertices. Intuitively it will work well most of the time, but will sometimes have to add an edge with very high cost because all vertices connected to the current one by an edge with low cost have already been visited.

Instead of adding a minimum cost edge among those adjacent to the current vertex, we could add an edge that has minimum cost overall, while ensuring that the resulting set of edges will form a tour. This corresponds to ordering the edges by increasing cost and adding them to the tour in that order, skipping edges that would lead to a vertex with degree more than two or a cycle of length less than n. This so-called *savings heuristic* has complexity $O(n^2 \log n)$, which is the complexity of sorting a set with n^2 elements.

Another intuitive approach is to start with a subtour, i.e., a tour on a subset of the set of vertices, and extending it with additional vertices. Heuristics following this general approach are known as *insertion heuristics*. A particular heuristics of this type has to specify a way of choosing (i) the initial subtour, (ii) the vertex to be inserted, and (iii) the way the new vertex is inserted into the subtour. Obvious choices for the initial subtour are cycles of length two or three. The *cheapest insertion heuristic* then chooses a vertex, and a place to insert the vertex into the subtour, in order to minimize the overall length of the resulting subtour. The *farthest insertion heuristic*, on the other hand, inserts a vertex whose minimum distance to any vertex in the current subtour is maximal. The idea behind the latter strategy is to fix the overall layout of the tour as soon as possible.

Of course, optimization does not need to end once we have constructed a tour. Rather, we could try to make a small modification to the tour in order to reduce its cost, and repeat this procedure as long as we can find such an improving modification. An algorithm that follows this general procedure is known as *local search algorithm*,

because it makes local modifications to a solution to obtain a new solution with a better objective value. A tour created using the nearest neighbor heuristic, for example, will usually contain a few edges with very high cost, so we would be interested in local modifications that eliminate these edges from the tour.

14.2 Local Search

Assume that we want to

$$\begin{aligned} &\text{minimize} && c(x) \\ &\text{subject to} && x \in X, \end{aligned}$$

and that for any feasible solution $x \in X$, the cost $c(x)$ and a *neighborhood* $N(x) \subseteq X$ can be computed efficiently. Local search then proceeds as follows:

1. Find an initial feasible solution $x \in X$.
2. Find a solution $y \in N(x)$ such that $c(y) < c(x)$.
3. If there is no such solution, then stop and return x; otherwise set the current solution x to y and return to Step 2.

The solution returned by this procedure is a *local optimum*, in the sense that its cost is no larger than that of any solution in its neighborhood. It need not be globally optimal, as there might be a solution outside the neighborhood with strictly smaller cost.

Any of the basic tour construction heuristics can be used to find an initial feasible solution in Step 1, and the whole procedure can also be run several times with different initial solutions. Step 2 requires a choice if more than one neighboring solution provides a decrease in cost. Natural options include the first such solution to be found, or the solution providing the largest decrease.

Most importantly, however, any implementation of a local search method must specify the neighborhood function N. A natural neighborhood for the TSP is the k-OPT neighborhood. Here, the neighbors of a given tour are obtained by removing any set of k edges, for some $k \geqslant 2$, and reconnecting the k paths thus obtained to a tour by adding k edges. Viewing tours as permutations, k-OPT cuts a permutation into k segments and reverses and swaps these segments in a arbitrary way. An illustration for $k = 2$ and $k = 3$ is shown in Figure 14.1.

The choice of k provides a tradeoff between solution quality and speed: the k-OPT neighborhood of a solution contains its ℓ-OPT neighborhood if $k \geqslant \ell$, so the quality of the solution increases with k; the same is also true for the complexity of the method, because the k-OPT neighborhood of a tour of length n has size $O(n^k)$ and computing the change in cost between two neighboring tours requires $O(k)$ operations. Empirical evidence suggests that 3-OPT often performs better than 2-OPT, while there is little gain in taking $k > 3$.

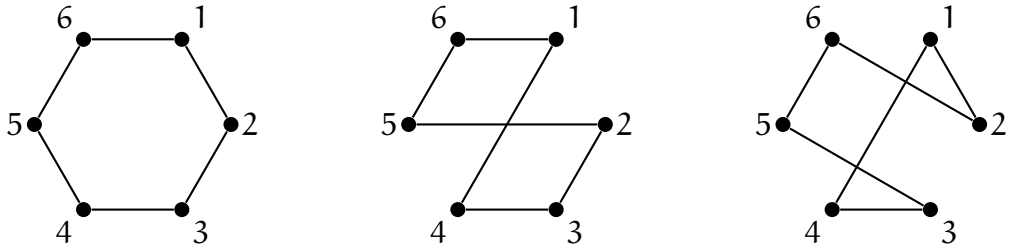

Figure 14.1: A TSP tour (left) and neighboring tours under the 2-OPT (middle) and 3-OPT neighborhoods (right). The tours respectively correspond to the permutations 123456, 143256, and 126534.

Note that the simplex method for linear programming can also be viewed as a local search algorithm, where two basic feasible solutions are neighbors if their bases differ by exactly one element. We have seen that in this case every local optimum is also a global optimum, so that the simplex method yields a globally optimal solution.

In general, however, local search might get stuck in a local optimum and fail to find a global one. Consider for example the TSP instance given by the cost matrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 4 & 4 \\ 4 & 0 & 1 & 0 & 4 \\ 4 & 4 & 0 & 1 & 0 \\ 0 & 4 & 4 & 0 & 1 \\ 1 & 0 & 4 & 4 & 0 \end{pmatrix}.$$

There are $4! = 24$ TSP tours, and

$$\begin{array}{llll} c(12345) = 5, & c(13245) = 6, & c(14235) = 10, & c(15234) = 6, \\ c(12354) = 6, & c(13254) = 12, & c(14253) = 20, & c(15243) = 12, \\ c(12435) = 6, & c(13425) = 10, & c(14325) = 17, & c(15324) = 12, \\ c(12453) = 10, & c(13452) = 6, & c(14352) = 9, & c(15342) = 17, \\ c(12534) = 10, & c(13524) = 0, & c(14523) = 10, & c(15423) = 17, \\ c(12543) = 17, & c(13542) = 12, & c(14532) = 17, & c(15432) = 20. \end{array}$$

It is easily verified that the tour 12345 is a local optimum under the 2-OPT neighborhood, while the global optimum is the tour 13524.

14.3 Simulated Annealing

To prevent local search methods from getting stuck in a local optimum, one could allow transitions to a neighbor even if it has higher cost, with the hope that solutions with lower cost will be reachable from there. *Simulated annealing* implements this idea

using an analogy to the process of annealing in metallurgy, in which a metal is heated and then cooled gradually in order to bring it to a low-energy state that comes with better physical properties.

In each iteration, simulated annealing considers a neighbor y of the current solution x and moves to the new solution with probability

$$p_{xy} = \min\left(1, \exp\left(-\frac{c(y) - c(x)}{T}\right)\right),$$

where $T \geqslant 0$ is a parameter, the *temperature*, that can vary over time. With the remaining probability the solution stays the same. When T is large, the method allows transitions even when $c(y)$ exceeds $c(x)$ by a certain amount. As T approaches zero, so does the probability of moving to a solution with larger cost.

It can further be shown that with a suitable *cooling schedule* that decreases T sufficiently slowly from iteration to iteration, the probability of reaching an optimal solution after t iterations tends to 1 as t tends to infinity. To motivate this claim, consider the special case where every solution has k neighbors and a neighbor of the current solution is chosen uniformly at random. The behavior of the algorithm can then be modeled as a *Markov chain* with transition probabilities

$$P_{xy} = \begin{cases} p_{xy}/k & \text{if } y \in N(x), \\ 1 - \sum_{z \in N(x)} p_{xz}/k & \text{if } y = x, \\ 0 & \text{otherwise.} \end{cases}$$

This Markov chain has a unique *stationary distribution* π, i.e., a distribution over X such that for all $x \in X$, $\pi_x = \sum_{y \in X} \pi_y P_{yx}$. In addition it can be shown that π must satisfy the *detailed balance* condition that $\pi_x P_{xy} = \pi_y P_{yx}$ for every pair of solutions $x, y \in X$. In fact, detailed balance is not only necessary but also sufficient for stationary, because it implies that $\sum_{x \in X} \pi_x P_{xy} = \sum_{x \in X} \pi_y P_{yx} = \pi_y \sum_{x \in X} P_{yx} = \pi_y$. It is not hard to show that π with

$$\pi_x = \frac{e^{-c(x)/T}}{\sum_{z \in X} e^{-c(z)/T}}$$

for every $x \in X$ is a distribution and satisfies detailed balance, and must therefore be the stationary distribution. Letting $Y \subseteq X$ be the set of solutions with minimum cost and $\pi_Y = \sum_{x \in Y} \pi_x$, we conclude that $\pi_Y/(1 - \pi_Y) \to \infty$ as $T \to 0$.

The idea now is to decrease T slowly enough for the Markov chain to be able to reach its stationary distribution. A common cooling schedule is to set $T = c/\log t$ in iteration t, for some constant c.