Local Search Methods for Discrete Optimization

The topic of this lecture is local search methods for discrete optimization—the problem of finding a global optimum in very large search spaces. Optimization problems arise in many fields, and their solutions have real-world impact. Here are a few examples:

- **Facility Location** Suppose a city is deciding where to build its fire stations, knowing that fire trucks will be dispatched from these stations in emergencies. Intuitively, they should be spread out across the city, but how can we assess what the best locations would be? First, we need to quantify how good a set of locations actually is. Maybe we care most about average response time, e.g., how far each building in the city is—on average—from the nearest fire station. Using this objective function, we can evaluate the quality of different arrangements of fire stations in search of an optimal solution. This is the facility location problem.
- **Vehicle Routing** Every day, delivery companies, like Fedex and UPS, are contracted to deliver a collection of packages to a variety of locations. The vehicle routing problem is the problem of assigning packages to trucks, and arranging routes for the truck drivers, so as to minimize the amount of time it takes to make all the deliveries, or total fuel consumption, or distance traveled, etc.
- **Antenna Design** Antennae are used to broadcast information over radio waves. TV and radio stations use antennae to broadcast their programming over "airwaves." Satellites and spacecraft need very specific antennae with two properties: 1) low energy usage and 2) a strong enough signal. NASA has spent a great deal of effort optimizing the antenna design of their satellites and space craft. They used a local search algorithm based on of principles of biological evolution to design their antennae.

A **discrete** optimization problem is one in which the state space is discrete (e.g., the integers, paths through a graph, etc.).[1] Informally, the following features are typical of discrete optimization problems:

- combinatorial (and hence, exponential) state space
- cost (reward) function to be minimized (maximized)
- similar solutions have similar costs ("continuous" objective function)
- exhaustive, blind, and heuristic search are intractable
- reasonable solutions are satisfactory

In contrast to typical search problems (e.g., Rubik's cube),[2] the solution to a *pure* optimization problem, such as facility location, does not include the path to a goal, but is only the goal itself. In particular, every state represents a complete solution to the problem, although not necessarily an optimal one. This observation suggests the use of **iterative improvement** algorithms to solve optimization problems: initialize the algorithm at some (random) state, and iteratively perturb the current solution, updating it if any of the tested perturbations yields an improvement.

[1] The opposite of *discrete* is *continuous*. Any real-valued vector space, including the reals themselves, is an example of a continuous domains.

[2] which might be more aptly named **planning** problems, since their solutions involve sequences of actions, *a.k.a.* **plans**

Local search methods—the focus of this lecture—are algorithms that seek iterative improvements. The term "local" refers to the fact that algorithms in this class only consider perturbations in a local neighborhood of the current solution. We study: (i) hill-climbing algorithms, which greedily improve upon the current solutio and (ii) simulated annealing, which temporarily explores changes worse than the current solution, but ultimately only exploits improvements in the current solution.

1 Problem Definition

An **optimization problem** (X, f) comprises a state space X and an objective function $f : X \to \mathbb{R}$, in which the goal is to identify a state x^* that is either a (global) minimum or a maximum: i.e., it is an optimum. Given an optimization problem, $x^* \in X$ is a minimum iff $f(x^*) \leq f(x)$, for all $x \in X$; similarly, $x^* \in X$ is a maximum iff $f(x^*) \geq f(x)$, for all $x \in X$. This lecture is tailored toward minimization problems and algorithms; maximization problems are handled analogously.

As the search spaces in optimization problems can be exceedingly large, one approach to solving them is to search only locally: i.e., within the neighborhood of a given state. Given an optimization problem (X, f), it is the job of the AI practitioner to define a **neighborhood function** $\mathcal{N} : X \rightrightarrows X$, which typically returns simple perturbations of x for which $f(x)$ can be computed efficiently. A state $x_l^* \in X$ is a local minimum iff $f(x_l^*) \leq f(y)$, for all $y \in \mathcal{N}(x_l^*)$. (And likewise, for a local maximum.)

The objective function of an optimization problem gives rise to a cost (or reward) landscape. Local search algorithms move around this landscape seeking its lowest troughs (or highest peaks).

A local search approach to optimization starts at some initial state, evaluates that state, looks around the neighborhood of that state for local improvements, possibly accepts a local improvement, and repeats. As these algorithms only search in local neighborhoods, they can only ever be guaranteed to find locally optima.

Note that local search is initialized at some initial state. One common approach to finding an initial state is—drum roll—heuristic search! In general, heuristic search can be slow. But it is also possible to run a quick-and-dirty heuristic search, such as one guided by a greedy heuristic, without backtracking.[3]

Random restarts Alternatively, local search can be initialized at random. This approach can be very useful, because then local search can be restarted multiple times (or run multiple times in parallel), from random initial states, increasing the likelihood of eventually landing on a global optimum.

2 Traveling Salesperson Problem

Figure 1 presents an instance of the traveling salesperson problem (TSP), a classic optimization problem. Providence, Boston, Hartford, New York, and Washington D.C., and their respective distances from one another (as the crow flies), are depicted. A path through the graph that visits all cities *exactly once* and returns to its origin is called a **tour**. The objective in TSP is to find a tour of minimal distance. The optimal tour in Figure 1 is PBHWNP, for which the total distance is 795.

Given your experience with search so far, it would be quite natural for you to propose that we tackle TSP by formulating it as a search problem: i.e., by building up tours incrementally. In Appendix 4, we describe two ways in which to formulate TSP as search. The difficulty with this approach is the number of tours: starting in Providence, there are 4 possible next cities, and from there, 3 possibilities, and so on. In general, on a complete graph of n nodes, there are $n!$ paths from any one node to all the others and back again.[4]

[3] Best-h heuristic search maintains a priority queue, and therefore employs backtracking.

[4] without repeats, i.e., without visiting the same city twice

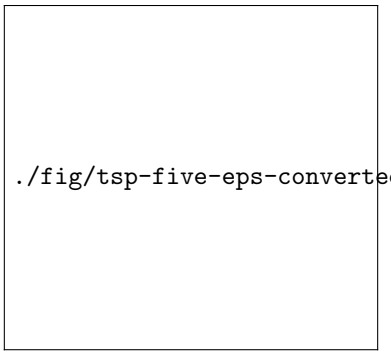


Figure 1: This graph encodes an instance of TSP in which states are tours, and the objective function f is defined as a tour's total distance, as indicated by the weights along its edges.

Instead of formulating TSP as search, we will take the point of view of optimization, where states are tours, and our job is to define a neighborhood operation based on which we can run local search.

Before proceeding, however, we remind you that local search methods must start somewhere! They require an initial state. Heuristic search (e.g., A* with backtracking is too onerous for this purpose. Instead, we suggest using greedy search (without backtracking) to generate a quick-and-dirty tour that can be used as an initial state before embarking on a local search. For example, a greedy search might proceed by applying the **nearest-neighbor heuristic**: visit the nearest city next, as long as that city has not already been visited, until all cities have been visited, and then return home.

There are many possible neighborhood operations. One simple example is inversion: given a tour, consider swapping the order of visiting each consecutive pair of cities on the tour (except the origin). More generally, an operation called 2OPT eliminates any two edges, say (x_1, x_2) and (y_1, y_2), and reconnects the nodes in the opposite way, as (x_1, y_2) and (y_1, x_2). Inversion and 2OPT are useful in practice because their computation is cheap—just subtract the distances of the deleted edges and add the distances of the new edges.

How might local search behave on our sample TSP, with inversion as the neighborhood operation? Suppose the initial state is PWBHNP of distance 1099. Applying inversion yields PBWHNP of distance 991, PWHBNP of distance 1097, and PWBNHP, of distance 1106. Most likely, a local search algorithm would accept PBWHNP, since $991 < 1099$; but, it could also reasonably accept PNBHWP since $1097 < 1099$. Assume PBWHNP is accepted. Applying inversion again yields PWBHNP (the original tour), PBHWNP of distance 795, and PBWNHP of distance 804. Now, if the tour PBHWNP is accepted, the algorithm verifies that it encountered a local (in fact, global) minimum and halts. (See Figure 2.)

3 Hill-Climbing

Hill-climbing is a family of local search algorithms that consider changes in the local neighborhood of a given state and accept changes that improve upon the current solution. Best-improvement and first-improvement are examples of hill-climbing algorithms. Other variants include GSAT and WalkSAT, which are special purpose algorithms for solving satisfiability.[5]

[5] Coming soon, to a lecture hall near you!

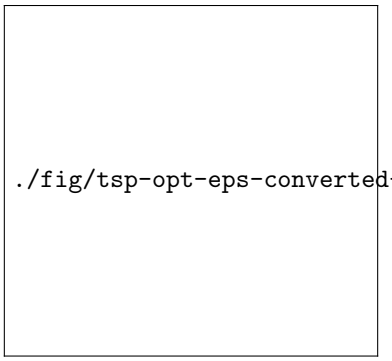


Figure 2: Local Search in TSP.

3.1 Best-Improvement Search

The main idea behind **best-improvement** search is to consider *all* states in the local neighborhood of the current state, and to accept one that best improves the objective function. Like all local search algorithms, best-improvement search terminates at a local (but not necessarily global) minimum. Nonetheless, its performance can be improved using random restarts.

BestImprovementSearch	
Inputs	optimization problem $(X, f, \mathcal{N})$
	random start state x
Output	best state visited x
Initialize	$x' \neq x$, $A = \{x\}$

`while` $(x' \neq x)$ `do`

1. $x' = x$, $A = \{x\}$
2. for all $y \in \mathcal{N}(x')$
   (a) if $f(y) < f(x)$, $x = y$, $A = \{x\}$
   (b) else if $f(y) = f(x)$, insert y in A
3. choose x in A

`return` x

Table 1: Best-Improvement Search.

3.2 First-Improvement Search

Unlike best-improvement search, **first-improvement** search does not consider all neighbors of the current solution. On the contrary, it considers its neighbors at random and accepts the first one that demonstrates an improvement. It halts if ever it loses its *patience* before finding an improvement. First-improvement search is also called **stochastic local search**.

Assuming infinite patience, stochastic local search terminates at a local optimum with probability 1. But rather than run the algorithm with too much patience, it is usually run repeatedly with random restarts.

FIRST$(X, f, \mathcal{N}, p, x, \epsilon)$	
Inputs	optimization problem
	patience time limit p
	random start state x
	rate of exploration ϵ
Output	best state visited x
Initialize	$t = 0$

`while` $(t < p)$ `do`

1. for some $y \in \mathcal{N}(x)$
   (a) if $f(y) < f(x)$, $x = y$, $t = 0$
   (b) else if $f(y) = f(x)$ and `rand`$[0, 1] \leq \epsilon$, $x = y$
   (c) else increment t

`return` x

Table 2: First-Improvement Search.

Assuming infinite patience and an infinite number of random restarts, stochastic local search terminates at a *global* optimum with probability 1.

RANDOM$(X, f, \mathcal{N}, p, n, \epsilon)$	
Inputs	optimization problem
	patience time limit p
	number of restarts n
	rate of exploration ϵ
Output	best state visited x^*
Initialize	$f(x^*) = \infty$

`for` $i = 1$ `to` n

1. choose start state $x \in X$
2. $z =$ FIRST$(X, f, \mathcal{N}, p, x, \epsilon)$
3. if $f(z) < f(x^*)$, $x^* = z$

`return` x^*

Table 3: Stochastic Local Search with Random-Restarts.

4 Simulated Annealing

Simulated annealing generalizes stochastic local search. With some probability, say p, simulated annealing accepts state changes that adversely affect the value of the objective function. In other words, in accordance with p, it encourages *exploration* of harmful states; otherwise, it *exploits* successful states.

A probability p of exploration can be determined in several ways. One approach is simply to fix p at some small value $\epsilon > 0$. Another option is to let p decrease with time: $p \sim 1/t$. Yet another idea is to let p decrease as $\Delta(x, y) = f(y) - f(x)$ increases, where x is the current solution and $y \in \mathcal{N}(x)$.

Updating in the spirit of this third idea can be achieved as follows: if $\Delta(x, y) < 0$ (i.e., y is an improvement), then let $p = 1$: i.e., exploit; if $\Delta(x, y) \geq 0$ (i.e., y is not an improvement), then let $p = e^{-\Delta(x,y)}$. In this way, small increases in the value of the objective function imply large values of p (exploration is likely; exploitation is not), whereas large increases in the value of the objective function imply small values of p (exploitation is likely; exploration is not). (For this definition of p, all ties are broken in favor of y, since $e^0 = 1$.)

SA($X, f, \mathcal{N}, T, x, C$)	
Inputs	optimization problem
	number of iterations T
	random start state x
	cooling schedule C
Output	best state visited x^*
Initialize	$x^* = x$, τ

`for` $t = 1$ `to` T

1. for some $y \in \mathcal{N}(x)$
   (a) compute $\Delta(x, y) = f(y) - f(x)$
   (b) if $\Delta(x, y) < 0$, let $p = 1$
   (c) else $p = e^{-\Delta(x,y)/\tau}$
   (d) if `rand`$[0, 1] \leq p$, $x = y$
      i. if $f(x) < f(x^*)$, $x^* = x$
2. decay τ according to schedule C

`return` x^*

Table 4: Simulated Annealing.

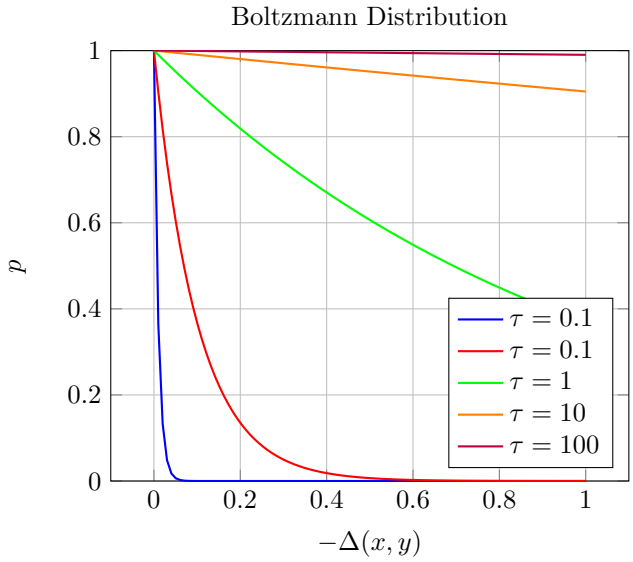
Simulated annealing is a local search algorithm based on a combination of ideas: (i) exploration decreases with time and (ii) exploration is more likely if it makes things only slightly worse, and less likely if it makes things significantly worse. Specifically, simulated annealing relies on the following exploration probabilities, given by the **Boltzmann distribution**: for $\tau > 0$,

$$p = \begin{cases} e^{-\Delta(x,y)/\tau} & \text{if } \Delta(x, y) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

Simulated annealing derives its name from the interpretation of τ as a temperature, which through slow cooling strengthens metal during its production. Initially, when $\tau \gg 0$ and p is high, simulated annealing behaves like a random walk: exploration is more likely than exploitation. Later in the search, when $\tau \to 0$ and p is low, simulated annealing reduces to stochastic local search: exploitation is more likely than exploration. (See Figure 4.)

A TSP as Search

Given a set of cities Y of cardinality m, and given the corresponding distances between cities ($d(x, y)$, for all $x, y \in Y$), we formulate TSP as a search problem in two ways. In the first formulation, states are **Hamiltonian paths**: paths that do not visit any city twice. In the second formulation, states are **Hamiltonian circuits**: cycles that do not visit any city twice.



A.1 Hamiltonian Paths

Let the set of states X consists of all Hamiltonian *paths*: i.e., paths that do not visit any city twice. Formally,

$$X = \{(x_1, \ldots, x_n) \mid n = 1, \ldots, N+1, x_i \in Y \text{ for all } 1 \leq i \leq n,$$
$$x_i \neq x_j \text{ for all } 1 \leq i \neq j \leq n, \text{ unless } i = 1,\ j = m+1\}$$

The set of goal states includes all states of length $N+1$. The set of start states includes all states of length 1. The transition function between states is defined as follows: at state $(x_1, \ldots, x_n)$,

$$\mathcal{T}(x_1, \ldots, x_n) = \{(x_1, \ldots, x_n, x_{n+1}) \mid x_{n+1} \neq x_i \in Y, \text{ for all } 1 \leq i \leq n, \}$$
$$\text{unless } i = 1, n = m$$

with cost function $c((x_1, \ldots, x_n), (x_1, \ldots, x_n, x_{n+1})) = d(x_n, x_{n+1})$. Figure 3 depicts the first three levels of the search space in this formulation for the traveling salesperson problem depicted in Figure 1.

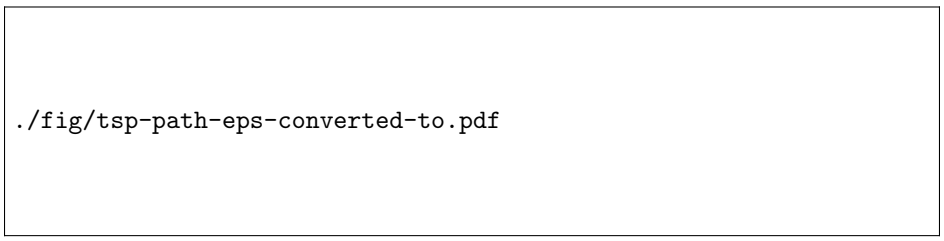


Figure 3: Search in TSP: States are Hamiltonian Paths.

One simple heuristic to solve TSP in this formulation is **nearest-neighbor**. Beginning at city x, the salesperson visits city y *s.t.* $d(x, y)$ is minimal; from city y, she visits city z *s.t.* $d(y, z)$ is minimal, *so long as doing so does not create a cycle of length less than* m, in which case she visits one of the next closest cities instead; and so on. Applying this heuristic to the search problem in Figure 3 generates the suboptimal path PBHNWP of distance 796.

A.2 Hamiltonian Circuits

Let the set of states X consists of all Hamiltonian *circuits*: i.e., cycles that do not visit any city twice—*a.k.a* tours. Formally,

$$X = \{(x, y_1, \ldots, y_n, x) \mid n = 1, \ldots, N-1, \\ x \neq y_i \neq y_j \in Y \text{ for all } 1 \leq i \neq j \leq n\}$$

In this case, the transition function between states is defined as follows: at state $(x, y_1, \ldots, y_i, y_{i+1}, \ldots, y_n, x)$,

$$\mathcal{T}(x, y_1, \ldots, y_n, x) = \{(x, y_1, \ldots, y_i, y, y_{i+1} \ldots, y_n, x) \mid y \neq y_i \neq x \in Y\}$$

at cost $d(y_i, y) + d(y, y_{i+1}) - d(y_i, y_{i+1})$. The set of goal states includes all states of length $N+1$. The set of start states includes all states of length 2.

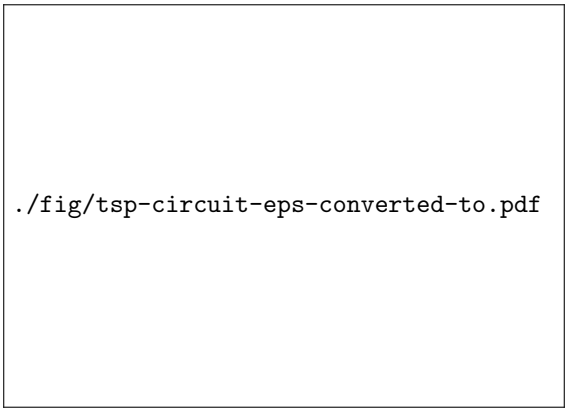


Figure 4: Search in TSP: States are Hamiltonian Circuits.

A greedy heuristic for solving TSP in this problem formulation is simply: insert a city into the tour that causes the smallest increase in the length of the tour. This heuristic finds an optimal tour in our example. (See Figure 4.)