Markov Chain Monte Carlo

Simulated Annealing

Sachin Shanbhag

Department of Scientific Computing
Florida State University,
Tallahassee, FL 32306.

Introduction

Most gradient-based minimization methods yield a local minimum that is close to the starting point.

Introduction

Most gradient-based minimization methods yield a local minimum that is close to the starting point.

Simulated annealing (SA) is a heuristic MCMC method for global minimization designed to explore and overcome many local minima.

It allows tremendous flexibility in specifying constraints.

Introduction

Most gradient-based minimization methods yield a local minimum that is close to the starting point.

Simulated annealing (SA) is a heuristic MCMC method for global minimization designed to explore and overcome many local minima.

It allows tremendous flexibility in specifying constraints.

The underlying idea of SA is to treat the cost function to be minimized as the reciprocal of the PDF. It can then be explored using standard MCMC techniques.

- generate new trial point based on current point
- evaluate the cost function at the two locations
- accept new value (with Metropolis or MH criterion) if it improves solution

Basic Algorithm

Idea of a fictitious "temperature", which also explains the "annealing" in SA.

The method is fashioned after crystal formation from molten materials by cooling.

If the rate of cooling is gradual, molecules move freely at high temperature (explore space), and slide into low-energy crystal lattices (stuck in minimas) at low temperatures.

Basic Algorithm

Idea of a fictitious "temperature", which also explains the "annealing" in SA.

The method is fashioned after crystal formation from molten materials by cooling.

If the rate of cooling is gradual, molecules move freely at high temperature (explore space), and slide into low-energy crystal lattices (stuck in minimas) at low temperatures.

Note that "annealing" is different from "quenching" in which the temperature is decreased very rapidly. Quenching freezes the structure in a high-energy state.

Basic Algorithm

Idea of a fictitious "temperature", which also explains the "annealing" in SA.

The method is fashioned after crystal formation from molten materials by cooling.

If the rate of cooling is gradual, molecules move freely at high temperature (explore space), and slide into low-energy crystal lattices (stuck in minimas) at low temperatures.

Note that "annealing" is different from "quenching" in which the temperature is decreased very rapidly. Quenching freezes the structure in a high-energy state.

In SA, at high temperatures, we explore parameter space jumping over (cost) energy barriers; at lower temperatures, exploration is restricted.

Basic Algorithm

Suppose the cost function is $E(\mathbf{s})$, where $\mathbf{s} = (s_1, s_2, ..., s_N)$ is a point in the multidimensional space.

We wish to find:

$$\arg\min_{\mathbf{s}} E(\mathbf{s})$$

Basic Algorithm

Suppose the cost function is $E(\mathbf{s})$, where $\mathbf{s} = (s_1, s_2, ..., s_N)$ is a point in the multidimensional space.

We wish to find:

$$\arg\min_{\mathbf{s}} E(\mathbf{s})$$

1. Start with an initial guess $\mathbf{s}_0$ and a high temperature T_0.

Basic Algorithm

Suppose the cost function is $E(\mathbf{s})$, where $\mathbf{s} = (s_1, s_2, ..., s_N)$ is a point in the multidimensional space.

We wish to find:

$$\arg\min_{\mathbf{s}} E(\mathbf{s})$$

1. Start with an initial guess $\mathbf{s}_0$ and a high temperature T_0.
2. Consider a gradually decreasing sequence of temperatures T_i, $i = 0, 1, \cdots, M$.

Basic Algorithm

Suppose the cost function is $E(\mathbf{s})$, where $\mathbf{s} = (s_1, s_2, ..., s_N)$ is a point in the multidimensional space.

We wish to find:

$$\arg\min_{\mathbf{s}} E(\mathbf{s})$$

1. Start with an initial guess $\mathbf{s}_0$ and a high temperature T_0.
2. Consider a gradually decreasing sequence of temperatures T_i, $i = 0, 1, \cdots, M$.
3. At each temperature T_i, perform Metropolis MCMC:
   - propose an update and evaluate function
   - preferentially accept updates that improve solution

$$a(\mathbf{s} \to \mathbf{s}') = \min\left\{1, \exp\left(-\frac{\Delta E}{T_i}\right)\right\},$$

where $\Delta E = E(\mathbf{s}') - E(\mathbf{s})$.

Notes

When $T \gg \Delta E$, the acceptance probability $a(\mathbf{s} \to \mathbf{s}') \approx 1$. High temperatures allow you to climb over "hills" in the cost function.

Notes

When $T \gg \Delta E$, the acceptance probability $a(\mathbf{s} \to \mathbf{s}') \approx 1$. High temperatures allow you to climb over "hills" in the cost function.

If cooling is sufficiently slow, the global minimum will be reached

Notes

When $T \gg \Delta E$, the acceptance probability $a(\mathbf{s} \rightarrow \mathbf{s}') \approx 1$. High temperatures allow you to climb over "hills" in the cost function.

If cooling is sufficiently slow, the global minimum will be reached

Typical choices one has to make are:

- initial temperature
- final temperature
- rate of temperature decrease

These choices are in addition to the typical Metropolis MCMC choices, such as the form of the proposal function, starting points etc.

Initial Temperature

A suitable T_0 is one that results in an average initial acceptance probability of $a_0(\Delta E > 0) \approx 0.8$. (Kirkpatrick)

Initial Temperature

A suitable T_0 is one that results in an average initial acceptance probability of $a_0(\Delta E > 0) \approx 0.8$. (Kirkpatrick)

T_0 is problem-specific; it can be estimated by an initial exploratory search in which all *increases* in the cost function E are accepted.

Initial Temperature

A suitable T_0 is one that results in an average initial acceptance probability of $a_0(\Delta E > 0) \approx 0.8$. (Kirkpatrick)

T_0 is problem-specific; it can be estimated by an initial exploratory search in which all *increases* in the cost function E are accepted.

Conduct an initial exploratory search by setting T_0 very large (10^{300}) and calculate the average increase in the cost function:

$$\overline{\Delta E}^{+} = \frac{1}{n} \sum_{\Delta E > 0} \Delta E,$$

where n is the number of terms in the summation.

Initial Temperature

A suitable T_0 is one that results in an average initial acceptance probability of $a_0(\Delta E > 0) \approx 0.8$. (Kirkpatrick)

T_0 is problem-specific; it can be estimated by an initial exploratory search in which all *increases* in the cost function E are accepted.

Conduct an initial exploratory search by setting T_0 very large (10^{300}) and calculate the average increase in the cost function:

$$\overline{\Delta E}^{+} = \frac{1}{n} \sum_{\Delta E > 0} \Delta E,$$

where n is the number of terms in the summation.

Using $a_0 = 0.8 = \exp(-\overline{\Delta E}^{+}/T_0)$, for production run set,

$$T_0 = -\frac{\overline{\Delta E}^{+}}{\ln a_0}$$

Final Temperature

The final temperature may be fixed during runtime.

It may be provisionally defined as the temperature around which the simulation ceases to make further progress.

Lack of progress can be defined as:

- the cost function stops decreasing further
- acceptance ratio falls below a certain threshold
- a combination of the two
- Or a certain (fixed) number of steps at a specified cooling rate.

Cooling Schedule

After every L trials (corresponding to $\approx 10 - 100$ MCS), the temperature may be decreased *exponentially*,

$$T_{k+1} = \alpha T_k,$$

with $\alpha \approx 1$. Often $\alpha = 0.95 - 0.999$.

Cooling Schedule

After every L trials (corresponding to $\approx 10 - 100$ MCS), the temperature may be decreased *exponentially*,

$$T_{k+1} = \alpha T_k,$$

with $\alpha \approx 1$. Often $\alpha = 0.95 - 0.999$.

It may also be decreased *linearly*, after every L trials with

$$T_{k+1} = T_k - \Delta T.$$

In either case α and ΔT control the rate of cooling.

Cooling Schedule

After every L trials (corresponding to $\approx 10 - 100$ MCS), the temperature may be decreased *exponentially*,

$$T_{k+1} = \alpha T_k,$$

with $\alpha \approx 1$. Often $\alpha = 0.95 - 0.999$.

It may also be decreased *linearly*, after every L trials with

$$T_{k+1} = T_k - \Delta T.$$

In either case α and ΔT control the rate of cooling.

We want to pick the slowest cooling rate that allows us to perform the calculation in a reasonable time.

Reseeding

In many practical simulations, reseeding may be necessary.

Reseeding involves moving back to the best solution found so far, and continuing the calculation.

Reseeding

In many practical simulations, reseeding may be necessary.

Reseeding involves moving back to the best solution found so far, and continuing the calculation.

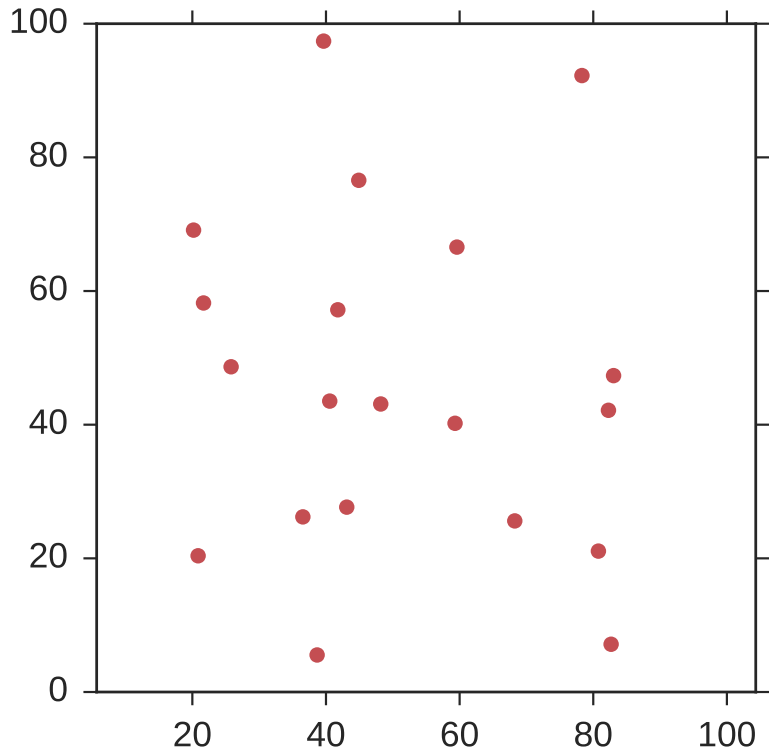
In our algorithm, we will choose to reseed with a small probability.

We will consider a specific example to demonstrate simulated annealing.

This example is the famous traveling salesman problem.

Traveling Salesman Example

Consider $Q = 20$ cities on a 100 by 100 grid.

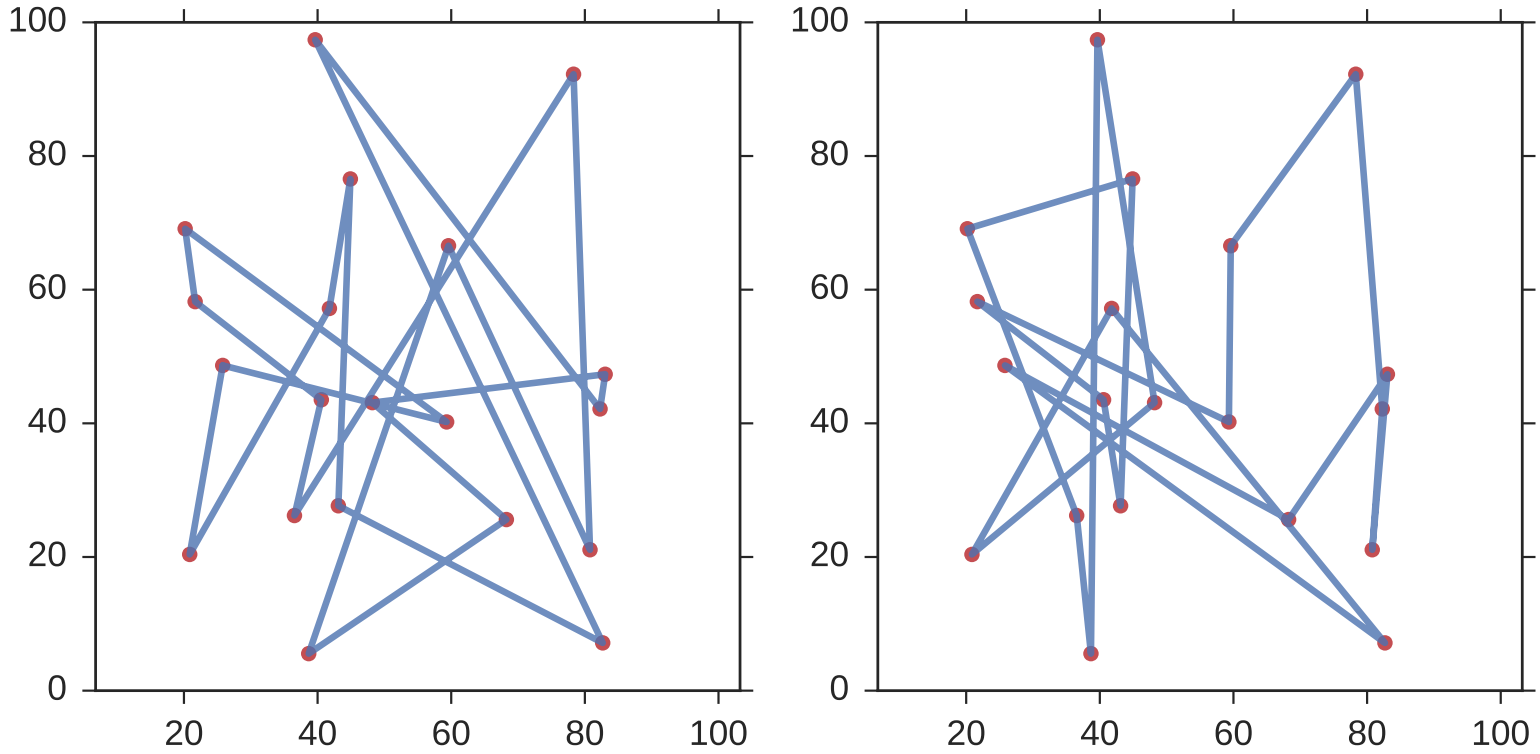


Suppose you are a salesman, who wants to pay a sales visit to all the Q cities, and return to the one you started from.

You want to come up with an itinerary that minimizes the distance you have to travel.

Traveling Salesman Example

Consider two possible routes:



The total distance traveled is 856.6 (left) and 813.4 (right).

Let's formalize things a little bit!

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Let d_{ij} denote the distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Let d_{ij} denote the distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Let a path be some permutation of the vector $[1, 2, ..., Q]$.

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Let d_{ij} denote the distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Let a path be some permutation of the vector $[1, 2, ..., Q]$.

In the examples above:

$$p_{\text{left}} = [1, 12, 13, ..., 19, 8, 1]$$

$$p_{\text{right}} = [4, 15, 6, ..., 20, 13, 4]$$

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Let d_{ij} denote the distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Let a path be some permutation of the vector $[1, 2, ..., Q]$.

In the examples above:

$$p_{\text{left}} = [1, 12, 13, ..., 19, 8, 1]$$

$$p_{\text{right}} = [4, 15, 6, ..., 20, 13, 4]$$

The total distance traversed by a given path is:

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

Mathematical Statement

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$.

Let d_{ij} denote the distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Let a path be some permutation of the vector $[1, 2, ..., Q]$.

In the examples above:

$$p_{\text{left}} = [1, 12, 13, ..., 19, 8, 1]$$

$$p_{\text{right}} = [4, 15, 6, ..., 20, 13, 4]$$

The total distance traversed by a given path is:

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

Note that there are $Q + 1$ elements in a path.

Cost Function

In this example, the cost function is the total distance.

```
def E(p, c):
    """given a sequence p, and city coordinates c,
    finds the total distance traversed"""

    d = 0.
    for j in range(len(p)-1):
        d += np.linalg.norm((c[p[j],:] - c[p[j+1],:]))

    return d
```

Note: Python arrays are C-style; indexing starts from 0.

Proposal Design

We want to find a path p that minimizes the total distance traversed E.

Proposal Design

We want to find a path p that minimizes the total distance traversed E.

Let's think a little bit about setting up efficient proposal moves.

A trial move is going to involve taking the current path p and modifying it to suggest a new path p'.

Proposal Design

We want to find a path p that minimizes the total distance traversed E.

Let's think a little bit about setting up efficient proposal moves.

A trial move is going to involve taking the current path p and modifying it to suggest a new path p'.

One way to build p' from p may be to simply swap a pair of random selected elements.

Proposal Design

We want to find a path p that minimizes the total distance traversed E.

Let's think a little bit about setting up efficient proposal moves.

A trial move is going to involve taking the current path p and modifying it to suggest a new path p'.

One way to build p' from p may be to simply swap a pair of random selected elements.

Furthermore, since the path is cyclic, without any loss of generality, we can simply swap an **internal** pair of elements.

This is the strategy we will adopt.

Proposal Code

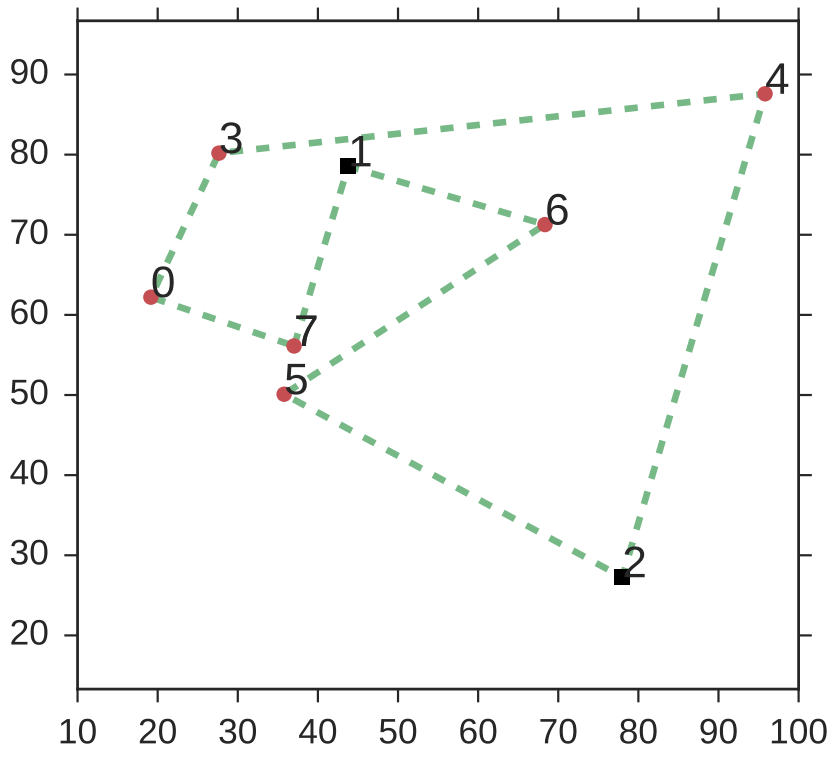
```
def proposal(oldp):
    """Given oldp, pick a pair of unique internal
       nodes i and j to swap"""

    N = len(oldp)

    i = np.random.randint(1, N-1)
    j = i

    while j == i or oldp[i] == oldp[j]:
        j = np.random.randint(1, N-1)

    return i, j
```

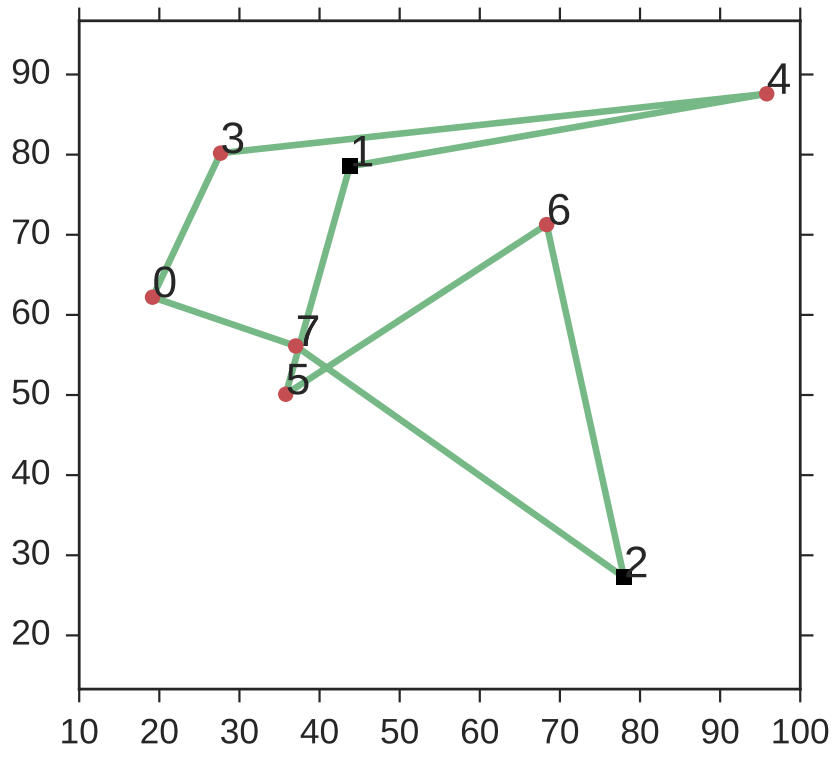
Let's test this using an example with $Q = 8$ cities.

```
# Location of the cities
Q = 8
c = np.random.uniform(0., 100., size=(Q,2))

# Pick a cyclic path
p = np.random.permutation(Q); p = np.append(p, p[0])
[7 1 6 5 2 4 3 0 7]
```

```
# Location of the cities
Q = 8
c = np.random.uniform(0., 100., size=(Q,2))

# Pick a cyclic path
p = np.random.permutation(Q); p = np.append(p, p[0])
[7 1 6 5 2 4 3 0 7]
# Generate new path pn
i,j = proposal(p)
4, 1

pn = p.copy(); pn[i] = p[j]; pn[j] = p[i]
[7 2 6 5 1 4 3 0 7]
```

Test Proposal





p = [7 1 6 5 2 4 3 0 7] pn = [7 2 6 5 1 4 3 0 7]

Efficiency: Observation 1

Note: We don't have to recompute the total E again, since most of the distances in the sum are unchanged.

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

Efficiency: Observation 1

Note: We don't have to recompute the total E again, since most of the distances in the sum are unchanged.

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

In this case before the swap, `p = [7 1 6 5 2 4 3 0 7]`:

$$E_{\text{old}} = d_{71} + d_{16} + d_{65} + d_{52} + d_{24} + d_{43} + d_{30} + d_{07}.$$

Efficiency: Observation 1

Note: We don't have to recompute the total E again, since most of the distances in the sum are unchanged.

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

In this case before the swap, `p = [7 1 6 5 2 4 3 0 7]`:

$$E_{\text{old}} = d_{71} + d_{16} + d_{65} + d_{52} + d_{24} + d_{43} + d_{30} + d_{07}.$$

After the proposed swap, `pn = [7 2 6 5 1 4 3 0 7]`

$$E_{\text{new}} = d_{72} + d_{26} + d_{65} + d_{51} + d_{14} + d_{43} + d_{30} + d_{07}.$$

Efficiency: Observation 1

Note: We don't have to recompute the total E again, since most of the distances in the sum are unchanged.

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

In this case before the swap, `p = [7 1 6 5 2 4 3 0 7]`:

$$E_{\text{old}} = d_{71} + d_{16} + d_{65} + d_{52} + d_{24} + d_{43} + d_{30} + d_{07}.$$

After the proposed swap, `pn = [7 2 6 5 1 4 3 0 7]`

$$E_{\text{new}} = d_{72} + d_{26} + d_{65} + d_{51} + d_{14} + d_{43} + d_{30} + d_{07}.$$

In general only 4 distance pairs are altered. This is especially useful if Q is large!

Efficiency: Observation 2

Recognize that we compute distances between cities over and over again.

Efficiency: Observation 2

Recognize that we compute distances between cities over and over again.

A lookup table of inter-city distances may be useful.

```
def CreateDistanceTable(c):
    """Given city coordinates c, create Q by Q matrix K,
       with pairwise distances"""

    Q = len(c)
    K = np.zeros((Q, Q))

    for i in range(Q):
        for j in range(Q):
            K[i,j] = np.linalg.norm((c[i,:] - c[j,:]))
    return K
```

Now, we can compute $\Delta E = E_{new} - E_{old}$ relatively cheaply

```
def dE(p, i, j, K):
    """change in cost function due to a swap"""

    ip = j; jp = i

    Enew = K[p[i-1], p[ip]] + K[p[ip], p[i+1]] +
           K[p[j-1],p[jp]]  + K[p[jp],p[j+1]]
    Eold = K[p[i-1], p[i]]  + K[p[i], p[i+1]]  +
           K[p[j-1],p[j]]   + K[p[j],p[j+1]]

    return Enew - Eold
```

Now, we can compute $\Delta E = E_{new} - E_{old}$ relatively cheaply

```
def dE(p, i, j, K):
    """change in cost function due to a swap"""

    ip = j; jp = i

    Enew = K[p[i-1], p[ip]] + K[p[ip], p[i+1]] +
           K[p[j-1],p[jp]]  + K[p[jp],p[j+1]]
    Eold = K[p[i-1], p[i]]  + K[p[i], p[i+1]]  +
           K[p[j-1],p[j]]   + K[p[j],p[j+1]]

    return Enew - Eold
```

In addition to the current path, we will also store the best path explored until now p^*.

Driver

Now that we've done the initial legwork, let's do the other Monte-Carlo routines: set initial temperature, acceptance, and driver routines.

```
def driver(c, alpha=0.995, nsteps=10000):

    # initialization
    Q     = len(c)
    L     = 10    # change temperature

    T0    = EstimateT0(c, nsteps=1000, a0=0.8)
    Temp = T0

    # starting position
    p     = np.random.permutation(Q);
    p     = np.append(p,p[0])
    cost = E(p, c)
```

```
# lookup table
K    = CreateDistanceTable(c)

# best solution
pstr = p.copy()
cstr = cost
numSucc = 0

# write a log-files of samples
f = open('log.dat','w')

# main loop
for iMCS in range(nsteps):

    i, j = proposal(p)
    acc, p, dEng = acceptMetro(p, i, j, K, Temp)
```

```
        if acc:
            cost = cost + dEng
            numSucc += 1

        if iMCS % L == 0:  # time to change Temp
            cost = E(p,c)
            if cost < cstr: # update best?
                cstr = cost; pstr = p
            if np.random.rand() < 0.05: # reseed?
                p    = pstr; cost = cstr

            f.write("{0:d} {1:8.3e} {2:8.2f} {3:8.2f}\n"
                    .format(iMCS, Temp, cost, cstr))
            Temp = Temp * alpha
    f.close()

    print("acceptance ratio =", float(numSucc)/nsteps)
    return pstr, cstr
```

Acceptance Function

```
def acceptMetro(pold, i, j, K, Temp):
    """accept or reject proposed swap?"""

    accept = False
    dEnerg = dE(pold, i, j, K)
    ratio  = np.exp(-dEnerg/Temp)

    if ratio > 1.:
        accept = True
    elif np.random.rand() < ratio:
        accept = True

    # keeping track of dEnergy for efficiency
    if accept:
        pnew = pold.copy()
        pnew[i] = pold[j]; pnew[j] = pold[i]
```

```
        return accept, pnew, dEnerg
    else:
        return accept, pold, dEnerg
```

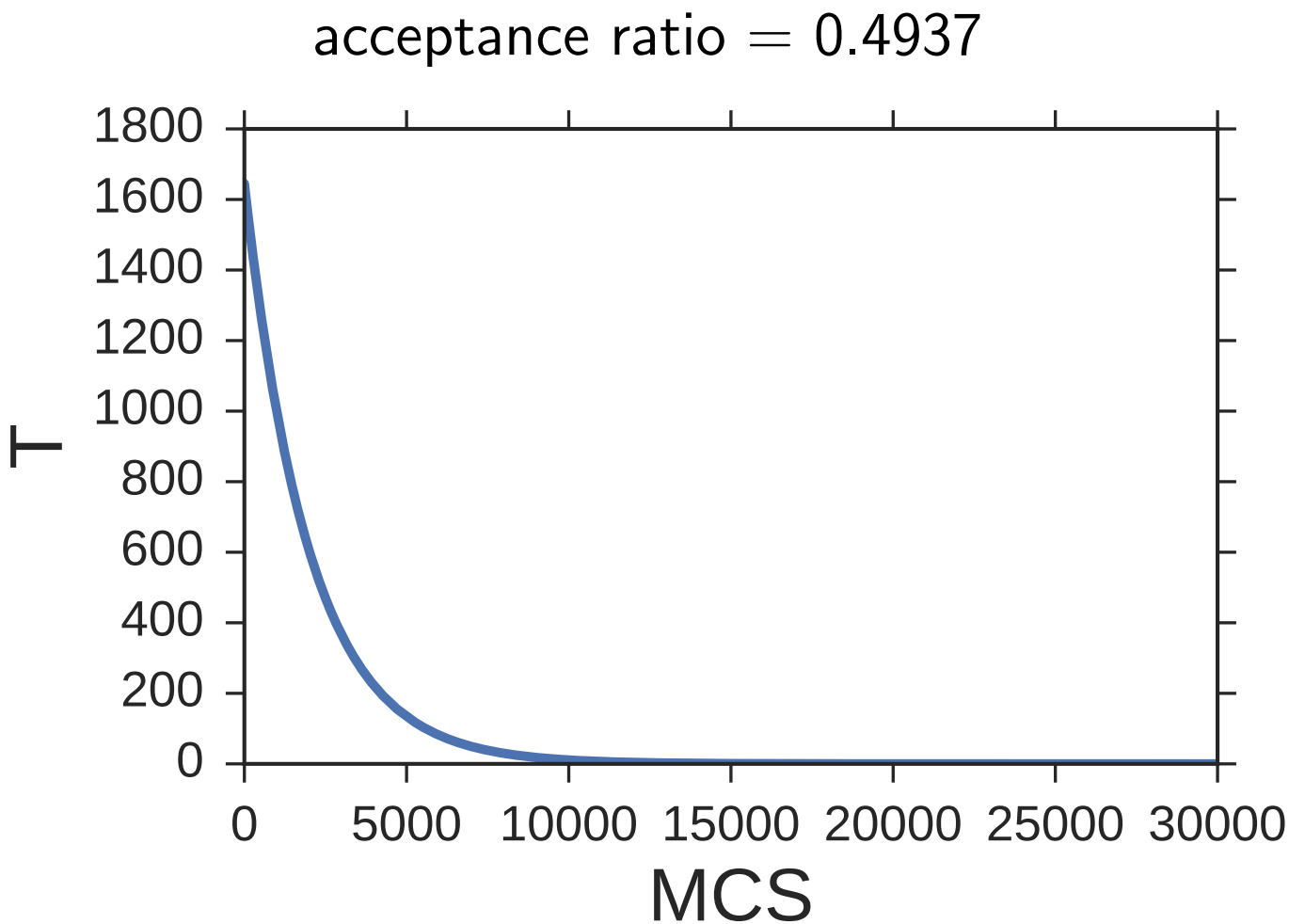
And finally, the routine to estimate T_0.

```
def EstimateT0(c, nsteps=1000, a0=0.8):

    Q     = len(c)
    Temp  = 1.0e300
    p     = np.random.permutation(Q); p = np.append(p,p[0])
    K     = CreateDistanceTable(c)

    cost = E(p, c)

    dEp   = 0.
    num   = 0
```

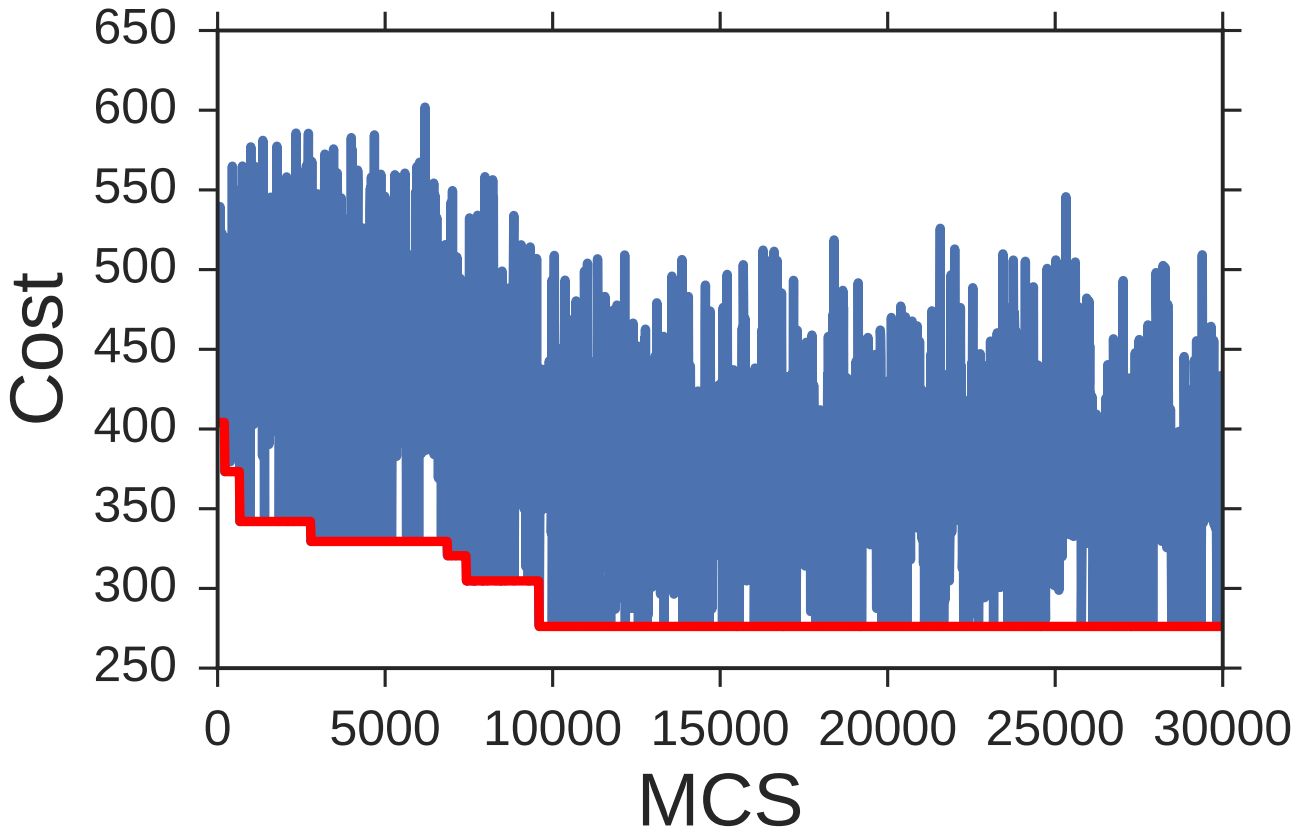
```
for iMCS in range(nsteps):

    i, j = proposal(p)
    acc, p, dEng = acceptMetro(p, i, j, K, Temp)

    if dEng > 0.:
        dEp += dEng
        num += 1

dEp = dEp/num

# factor 10 for safety
T0 = 10.0 * -dEp/np.log(a0)

return T0
```

Test

```
Q = 10
c = np.random.uniform(0., 100., 2*Q)
c = c.reshape((Q,2))
pstr, cstr = driver(c, 0.995, 30000)
```

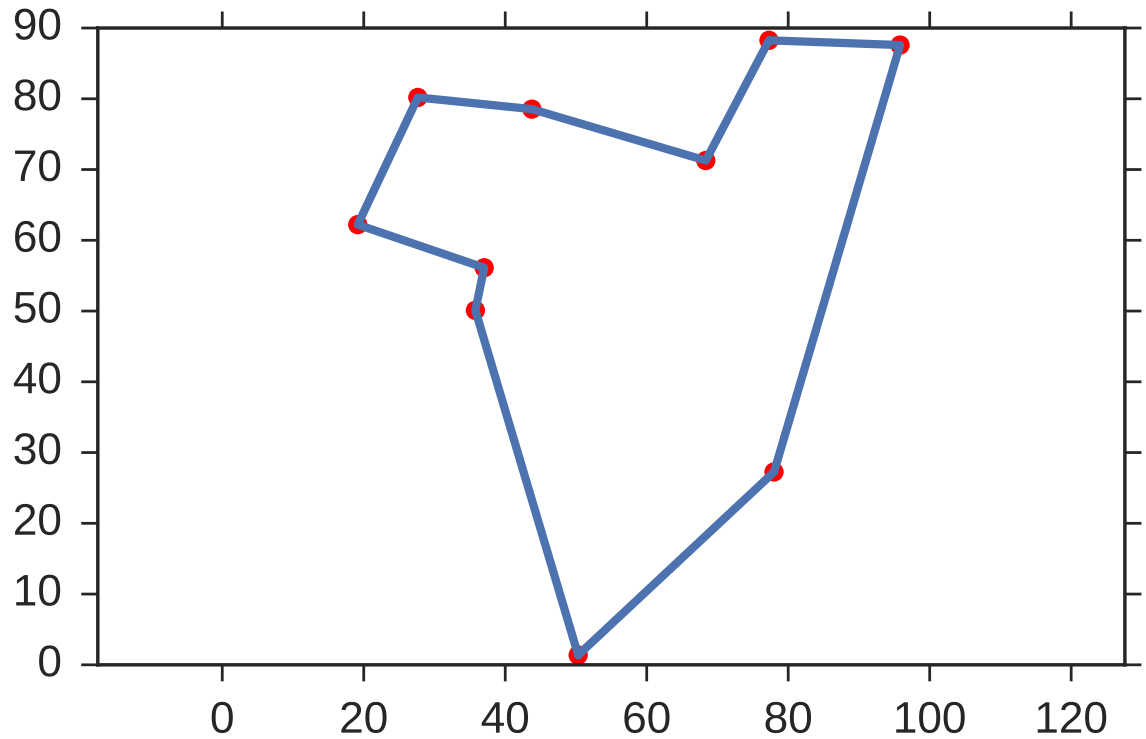


Cost



Red line corresponds to the best solution c^*, and the blue lines are the "current" cost c.

Best Solution



Postface

SA is guaranteed to converge to the global minimum if T_0 is large enough, and the cooling rate is slow enough.

This is reassuring, but doesn't tell us much about how to pick algorithm parameters to converge in a reasonable period of time.

Postface

SA is guaranteed to converge to the global minimum if T_0 is large enough, and the cooling rate is slow enough.

This is reassuring, but doesn't tell us much about how to pick algorithm parameters to converge in a reasonable period of time.

In adaptive simulated annealing, the algorithm parameters that control temperature schedule and random step selection are automatically adjusted according to algorithm progress.

This potentially makes the algorithm more efficient and less sensitive to user defined parameters than vanilla SA.

Postface

SA is guaranteed to converge to the global minimum if T_0 is large enough, and the cooling rate is slow enough.

This is reassuring, but doesn't tell us much about how to pick algorithm parameters to converge in a reasonable period of time.

In adaptive simulated annealing, the algorithm parameters that control temperature schedule and random step selection are automatically adjusted according to algorithm progress.

This potentially makes the algorithm more efficient and less sensitive to user defined parameters than vanilla SA.

In any event, it is advisable to try SA using multiple starting points to ensure that the global minimum has been sampled.

Postface

SA is guaranteed to converge to the global minimum if T_0 is large enough, and the cooling rate is slow enough.

This is reassuring, but doesn't tell us much about how to pick algorithm parameters to converge in a reasonable period of time.

In adaptive simulated annealing, the algorithm parameters that control temperature schedule and random step selection are automatically adjusted according to algorithm progress.

This potentially makes the algorithm more efficient and less sensitive to user defined parameters than vanilla SA.

In any event, it is advisable to try SA using multiple starting points to ensure that the global minimum has been sampled.

Implementations of SA are available in most compiled and interpreted languages.