Simulated Annealing

CE 377K

February 3, 2015

REVIEW

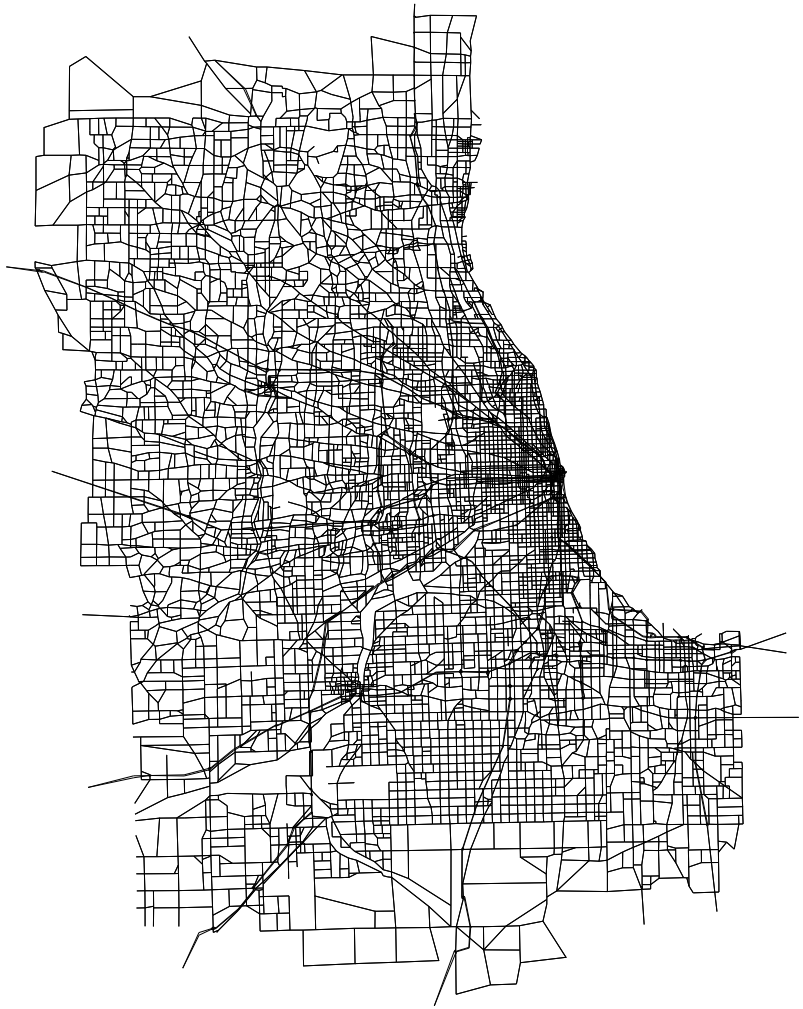
HW 1 posted, due February 10

Unconstrained optimization

Line search: Newton's method and bisection

MORE COMPLICATED PROBLEMS AND HEURISTICS

Last week we saw techniques for solving simple problems (no constraints, or a single decision variable.)



Of course, most real-world problems are much more complicated.

Think back to the problems we formulated in the first week and imagine the number of decision variables and constraints involved.

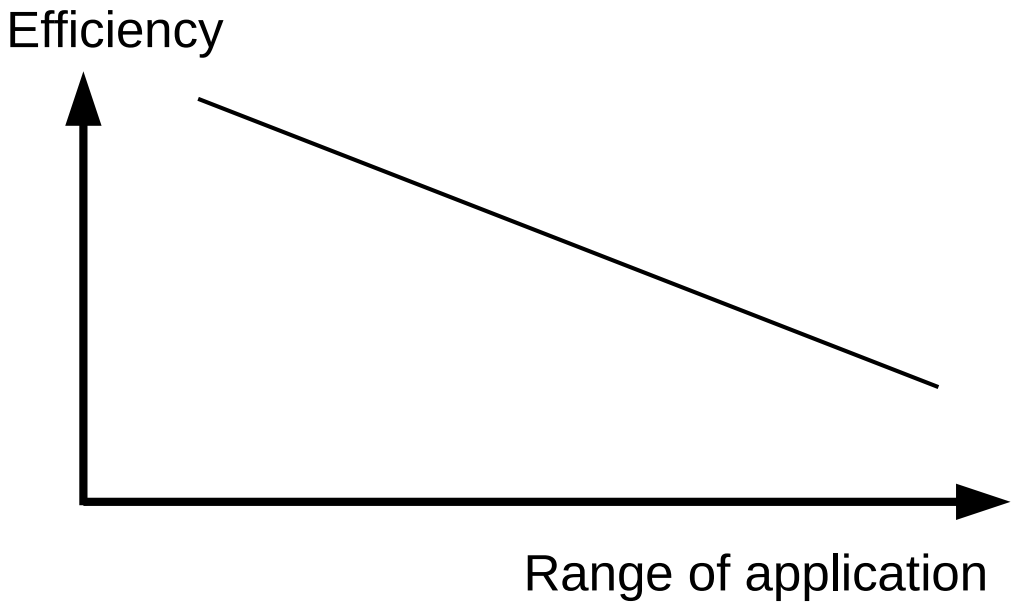
Transit frequency setting: 89 decision variables, 189 constraints

Maintenance scheduling: 795,350 decision variables; 1,590,710 constraints

Shortest path: 18,961 decision variables; 26,349 constraints

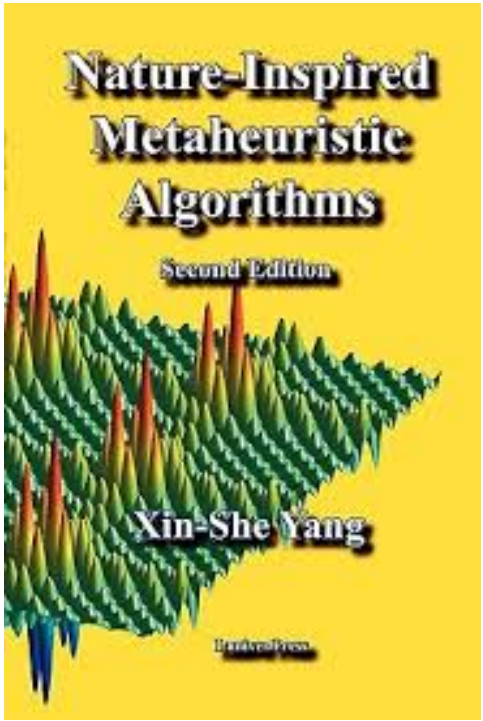
It is not at all unusual to have tens of thousands, or even millions, of decision variables and constraints.

Being able to efficiently solve these kinds of problems requires specialized techniques, which we will see as the course goes on.



Remember our rule of thumb: there is a tradeoff between a method's effectiveness and its generality.

Some of the most broadly applicable methods are called *heuristics* (or sometimes *metaheuristics*).

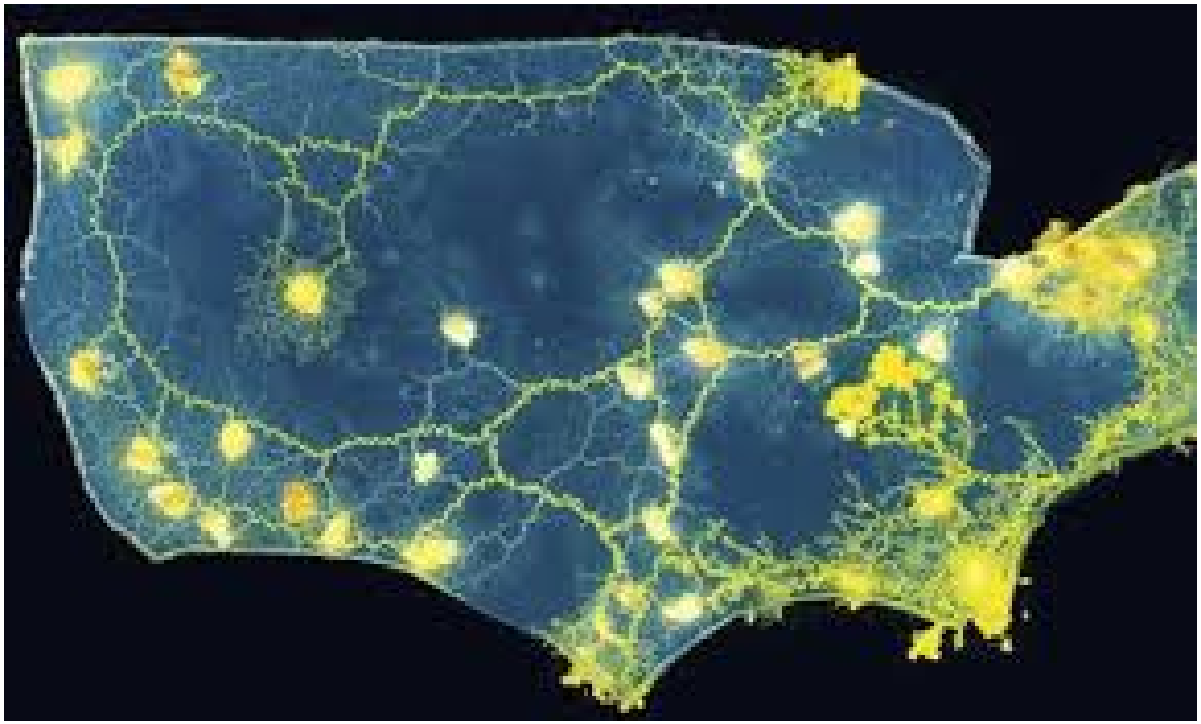


A heuristic is not guaranteed to find the global optimum, but often works well in practice.

In practice, heuristics are only used for very large problems, or problems where there is no known exact method which is efficient enough to use.

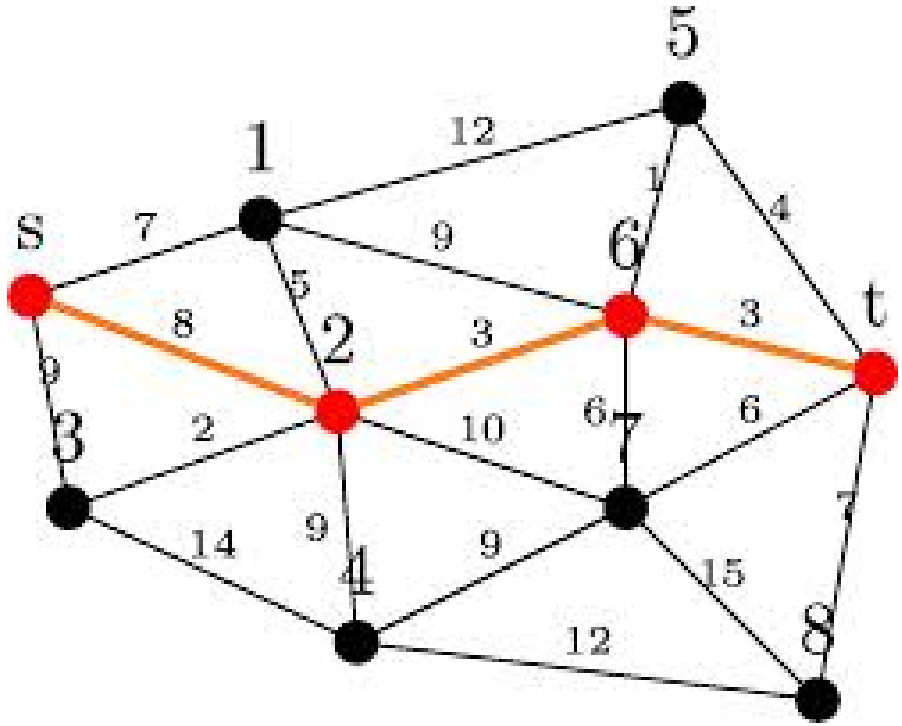
Sometimes, finding a "good enough" solution in a short amount of time is better than spending a lot of time to find the exact optimal solution. Why?

Many heuristic techniques are inspired by the natural world.

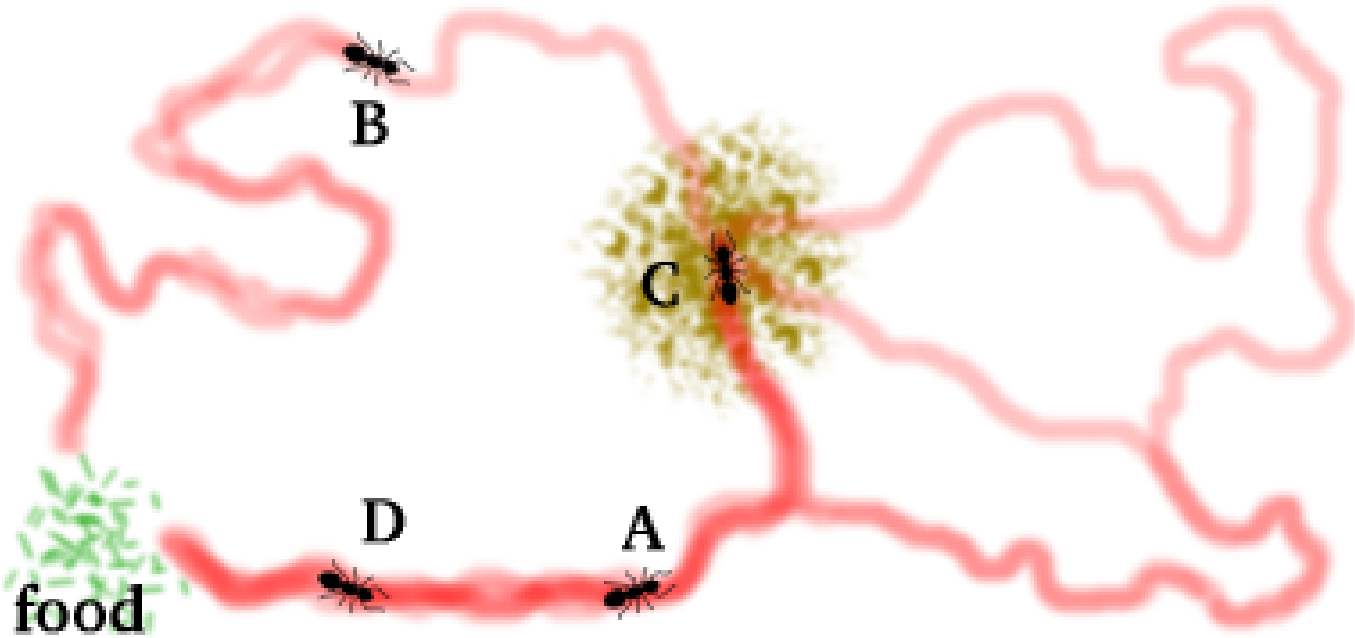


Very commonly living systems successfully (if not optimally) accomplish complicated tasks with relatively simple rules

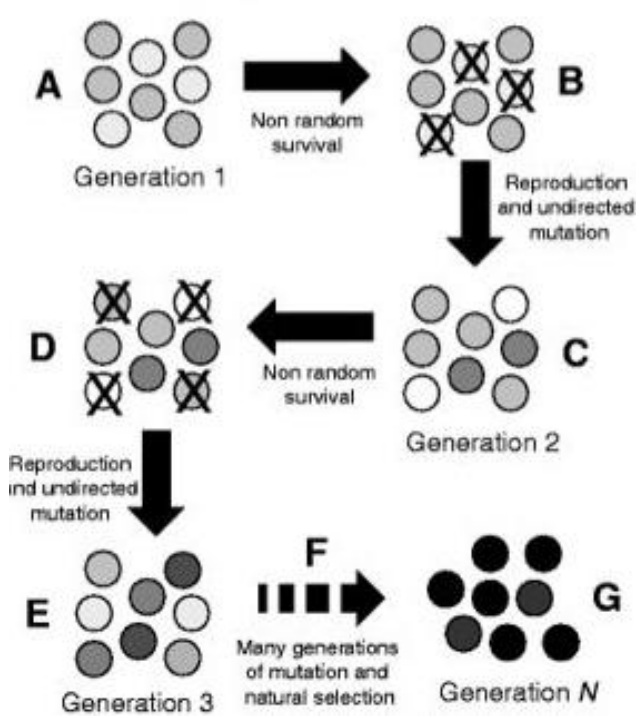
How do humans find shortest paths?



How does an ant colony find food sources?



How does natural selection work?



This week, we will see two heuristic methods.

- *Simulated annealing*, which searches through the feasible region in a half-systematic, half-random way.
- Genetic algorithms, which maintain a "population" of feasible solutions which are selectively "bred" with each other.

Unlike last week, we are making *no* assumptions on the number of decision variables or constraints.

SIMULATED ANNEALING

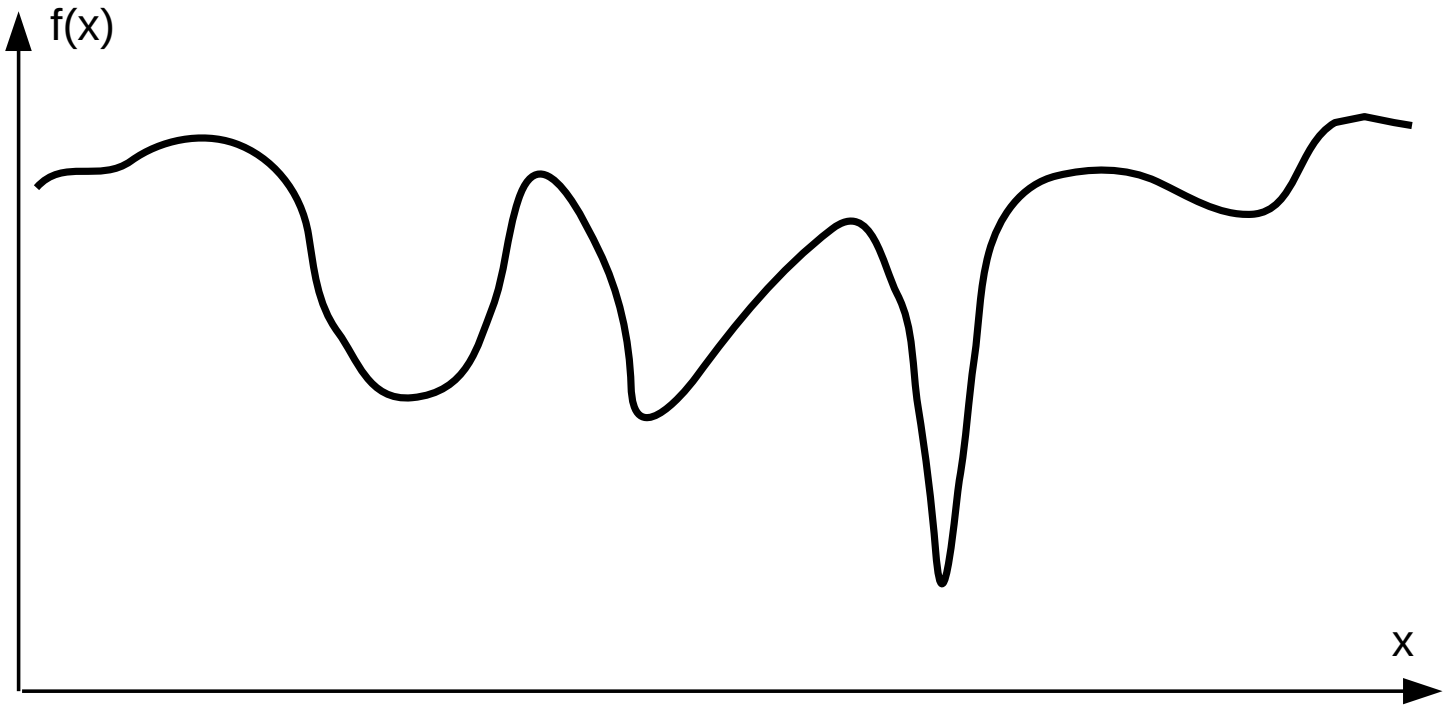
Imagine a hiker walking through a mountainous park area.

The hiker is trying to find the lowest elevation point in the park. (The park is the feasible region, the elevation profile is the objective function.)

However, there is a dense fog and the hiker has no map, so they have to rely on what they can see nearby.

What strategy should they use to try to find the lowest point?

One approach is to always walk in a downhill direction.



The downside of this "local search" strategy is that it is very easy to get trapped in a local minimum which is not global.

So, sometimes we need to walk uphill... but we can't do it all the time.

The *simulated annealing* algorithm incorporates randomization into local search.

A typical step of simulated annealing works as follows:

- Given a current feasible solution $\mathbf{x}$, generate a new feasible solution $\mathbf{x}'$ which is close to $\mathbf{x}$.
- Compare $f(\mathbf{x})$ and $f(\mathbf{x}')$. If $\mathbf{x}'$ is better, accept $\mathbf{x}'$ as the new current solution and repeat.
- If $\mathbf{x}'$ is *worse*, accept $\mathbf{x}'$ with some probability and repeat. Otherwise, generate another $\mathbf{x}'$.

How should this probability be chosen? We run into problems if the probability is too high or too low.

Simulated annealing uses a parameter T called the *temperature* to reflect the probability of moving uphill. When the temperature is high, uphill moves are more likely to be chosen. When the temperature is low, uphill moves are unlikely to be chosen. The algorithm uses a *cooling schedule* to gradually decrease the temperature over time.

The hope is that having an initially high temperature allows a large portion of the feasible space to be explored, but that lowering the temperature ensures that over time there is more and more preference for lower points.

One common formula (not the only choice) for the probability of moving uphill is $\exp(-[f(\mathbf{x}') - f(\mathbf{x})/T)$.

One common cooling schedule is to start with an initial temperature T_0, and multiply it by a common factor $k \in (0, 1)$ every n iterations.

Finally, since there is no guarantee that the algorithm will end up in the lowest point found during the search, you should keep track of the best solution found so far (call it $\mathbf{x}^*$).

This is like the hiker keeping a record of their journey, and at the end of their search returning to the lowest point found so far.

Then, the entire algorithm is:

1. Choose some initial feasible solution $\mathbf{x} \in X$, and calculate the value of the objective function $f(\mathbf{x})$.
2. Initialize the best solution to the initial one $\mathbf{x}^* \leftarrow \mathbf{x}$.
3. Set the temperature to the initial temperature: $T \leftarrow T_0$
4. Repeat the following steps n times:
   (a) Randomly generate a new feasible solution $\mathbf{x}'$ which neighbors $\mathbf{x}$.
   (b) If $f(\mathbf{x}') < f(\mathbf{x}^*)$, it is the best solution found so far, so update $\mathbf{x}^* \leftarrow \mathbf{x}'$.
   (c) If $f(\mathbf{x}') \leq f(\mathbf{x})$, it is a better solution than the current sone, so update $\mathbf{x} \leftarrow \mathbf{x}'$.
   (d) Otherwise, update $\mathbf{x} \leftarrow \mathbf{x}'$ with probability $\exp(-[f(\mathbf{x}') - f(\mathbf{x})]/T)$
5. If $T > T_f$, then reduce the temperature ($T \leftarrow kT$) and return to step 4.
6. Report the best solution found $\mathbf{x}^*$

Some questions:

- **How should the cooling schedule be chosen?** Hard to give general guidance, very problem-specific. Trial and error is common.
- **How should the initial solution be chosen?** If you can come up with a good rule of thumb *quickly*, use it. Otherwise, just randomize.
- **What is a neighboring solution?** Again problem-specific; should be similar to the current solution, but should eventually be able to reach any other feasible point.
- **How can I perform an action with probability p?**

  Generate a uniform random number between 0 and 1, perform the action if that number is less than p.

Some neighborhood examples:

- **Transit frequency setting:** Swap two buses between their routes.
- **Maintenance scheduling:** Swap two facilities to be maintained during a given year.
- **Facility location:** Change the location of one of the three facilities.

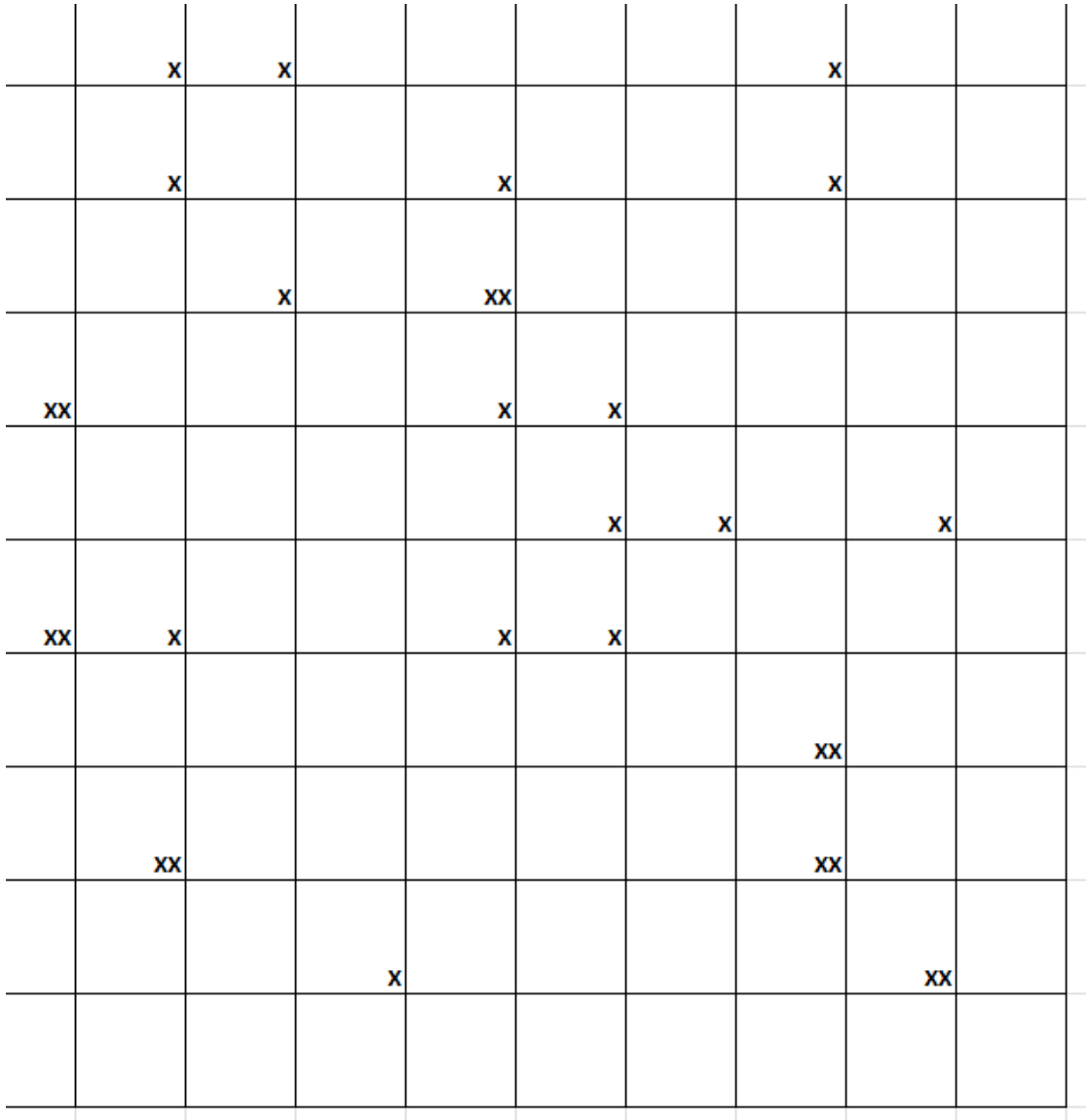
These are not the only choices! It's a good exercise to think up some others.

EXAMPLE

Let's return to the facility location problem. In this instance, there are 100 eligible facility locations (costs shown below):

8.7	13.5	11.0	10.2	6.3	6.4	12.4	13.5	9.7	6.8
5.8	12.4	8.8	15.1	14.7	13.9	9.0	5.2	8.7	12.7
11.8	9.8	12.4	12.6	6.3	13.3	7.7	10.2	13.6	13.5
5.4	6.9	11.1	8.4	14.2	10.9	10.7	11.7	8.1	8.9
4.9	12.5	10.8	6.6	12.0	6.8	11.9	9.2	9.5	10.0
14.4	9.7	8.6	11.6	8.0	6.7	12.7	5.9	7.6	14.1
7.7	9.6	6.4	9.9	9.2	13.3	12.3	14.7	15.0	5.1
7.6	14.7	7.1	5.5	5.5	9.7	9.7	14.4	7.9	15.1
10.9	12.5	9.2	12.1	14.8	6.4	6.1	10.5	12.5	10.8
12.0	5.9	13.1	10.0	11.9	10.0	8.6	8.4	10.7	5.3

Here are the locations of the 30 customers.



Through trial and error, the following cooling schedule was chosen:

- Initial temperature $T_0 = 1000$
- Final temperature $T_f = 100$
- Temperature reduction factor $k = 0.75$
- Iterations before reducing temperature $n = 8$

Initial solution: locate facilities at (5,1), (8,6), and (6,1), objective is 158.

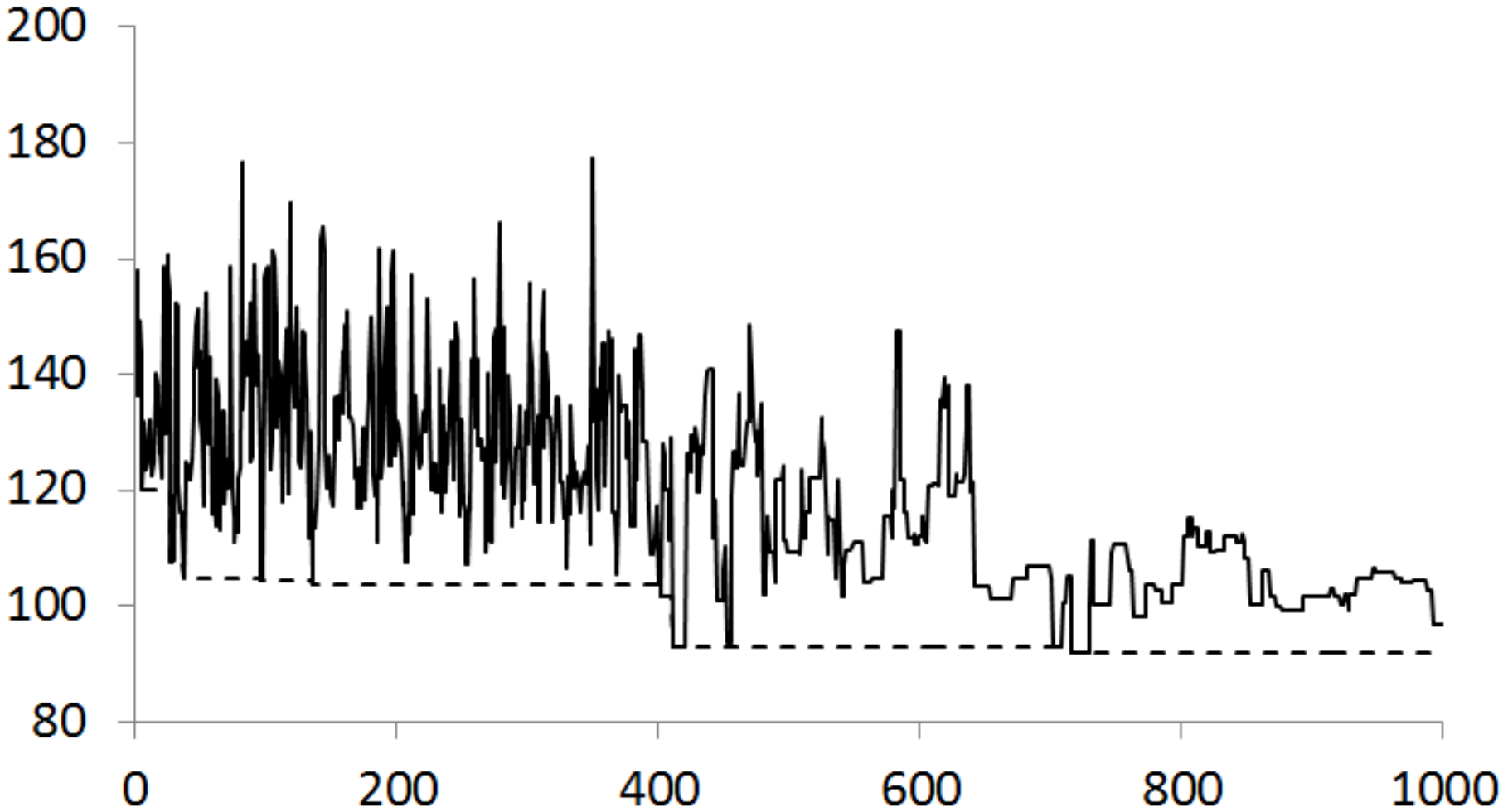
Randomly generate neighboring solution (5,1), (8,6), (1,0), which has cost 136.4

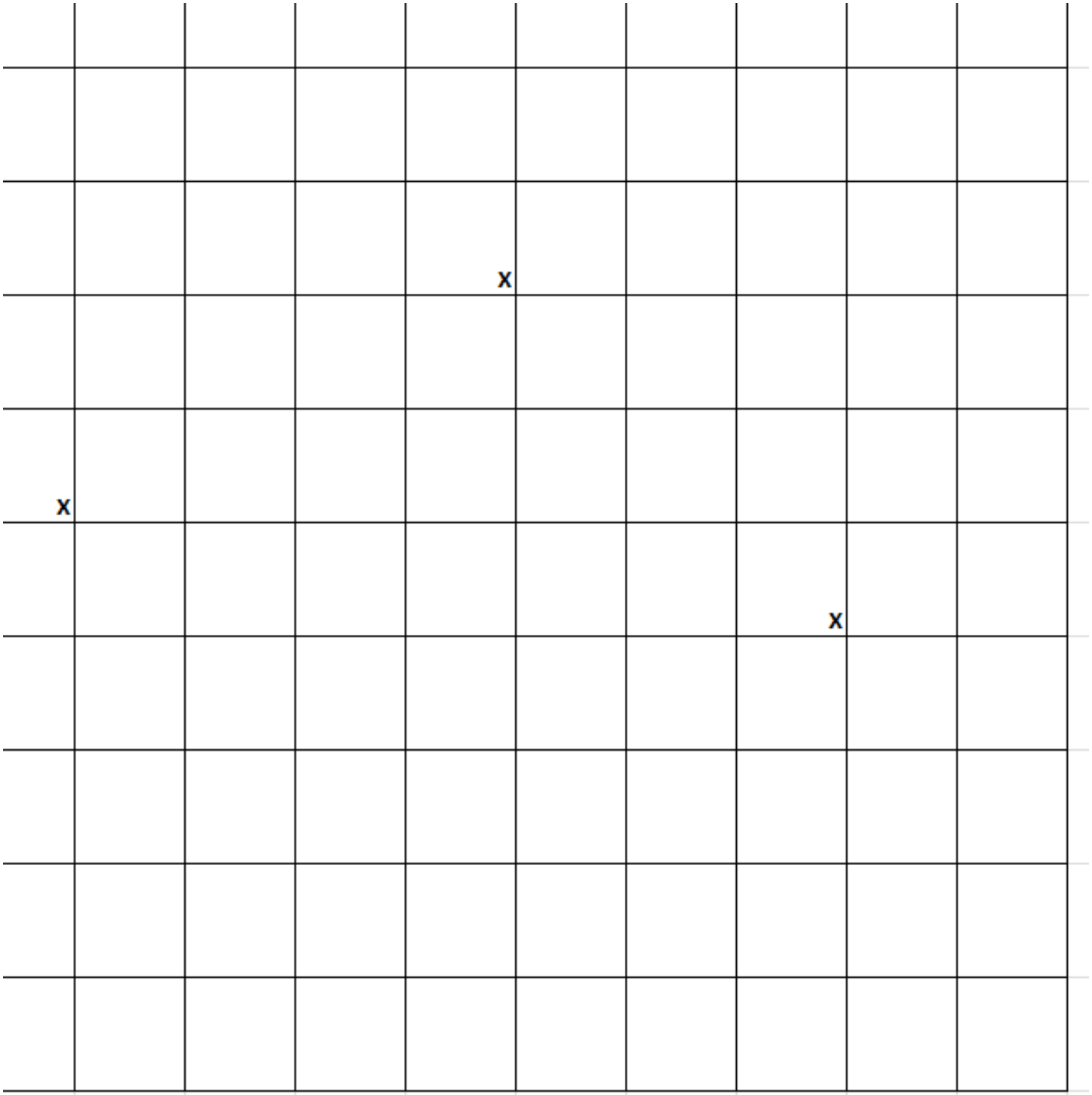
This cost is lower, so accept this as the new current solution.

Randomly generate neighboring solution (8,2), (8,6), (1,0), which has cost 149.1.

The cost is higher, the probability of accepting this move is 0.881.

Assume this move is rejected, generate another neighboring solution from the same point and repeat.


200
180
160
140
120
100
80
0
200
400
600
800
1000


X
X
X