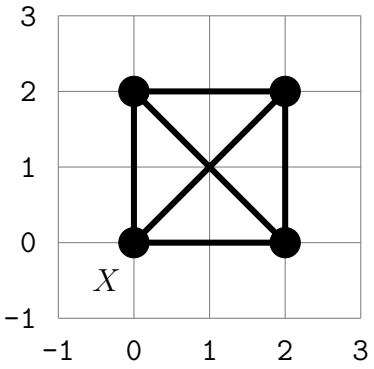
CISS450: Artificial Intelligence
Assignment 8

OBJECTIVES

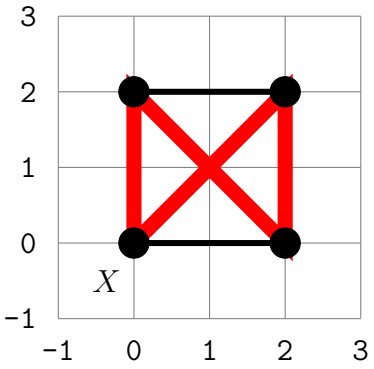
1. Solve AI (optimization) problems using local search techniques.
2. To be acqainted with standard AI problem(s).

Q1. Local search and TSP

The traveling salesman program is a very famous problem in computer science. Briefly, a salesman need to visit a list of cities or houses (starting at say X) and he wants to minimize his travel distance. For instance here are 4 cities (the dots) and there the 6 roads (lines) are as shown:

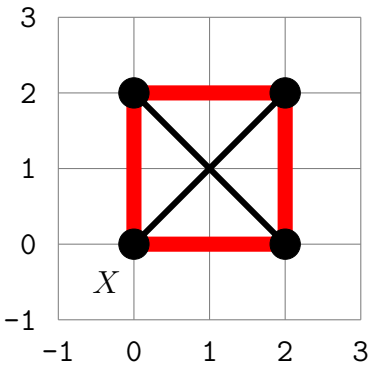


Here's a path that visits all 4 cities (starting and ending at X):



$$[(0,0),(2,2),(2,0),(0,2),(0,0)]$$

But that's not the shortest. This is the shortest:



$$[(0,0),(0,2),(2,0),(2,0),(0,0)]$$

Note that the last city visited in also the first: the path is closed.

The above is a simplification of TSP since we are assuming that there's a road between any two points. Also, we're assuming that the roads are straight lines since there are roads between any two cities. In a more general problem, you might have fewer roads so that the saleman has lesser freedom and the roads might not be straight so that we would have to attach distance to every pair of points (instead of computing the distance by the Euclidean distance formula). For the general TSP, we might need to revisit a dot more than once.

For this assignment, we will assume:

1. There are N cities or points in a 2D space.
2. You have maximum freedom in terms of roads: you can go from one point to another through a straight line.
3. The distance between two points is measure by the "straight line distance", i.e., the Euclidean distance formula.
4. The goal of the salesman is find a path of the shortest total distance that visits all cities exactly once (except for the first which also appears as the last.)

Here are some videos. Watch them. For each entry you will see the local search method and the successor generation method. The local search methods are hc (hillclimbing), fchc (first choice hill climbing), sa (simulated annealing), rrhc (random restart hillclimbing), shc (stochastic hillclimbing). The successor generation methods are: p2 (permute 2) and o2 (opt-2).

Local search method	Successor generation method	Link
hc (hillclimbing)	p2 (permute 2)	click here
fchc (first choice hillclimbing)	p2 (permute 2)	click here
rrhc (random restart hillclimbing)	p2 (permute 2)	click here

shc (stochastic hillclimbing)	p2 (permute 2)	click here
sa (stochastic annealing)	p2 (permute 2)	click here
hc (hillclimbing)	o2 (opt-2)	click here
fchc (first choice hillclimbing)	o2 (opt-2)	click here
rrhc (random restart hillclimbing)	o2 (opt-2)	click here
shc (stochastic hillclimbing)	o2 (opt-2)	click here
sa (simulated annealng)	o2 (opt-2)	clich here

Note that the actual solution is a closed path, i.e., the last point of the path is the same as the first point. However in the given program, for the list of points, we omit the repeating last point.

TSP: GENERATING SUCCESSOR STATES

There are many ways to generate successor states. Here are some:

Method 1: Permute two points. This one is easy. Given a list of points, we can permute two points in the list: For instance suppose

```
[..., (a,b), ..., (c,d), ...]
```

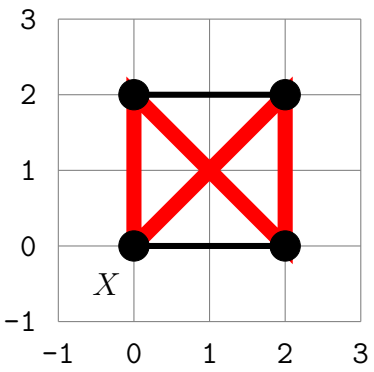
the following is a successor state:

```
[..., (c,d), ..., (a,b), ...]
```

If you permute the same point, then the path is clearly unchanged. The branching factor, if there are N points, is of course

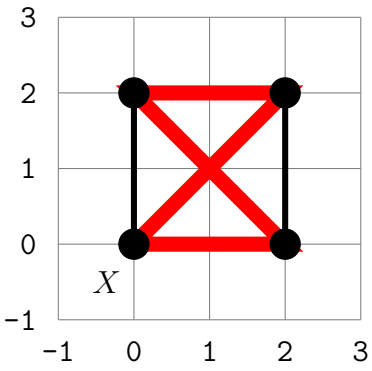
$$\binom{N}{2} = \frac{N(N-1)}{2}$$

For instance if you have this path



$$[(0,0),(2,2),(2,0),(0,2),(0,0)]$$

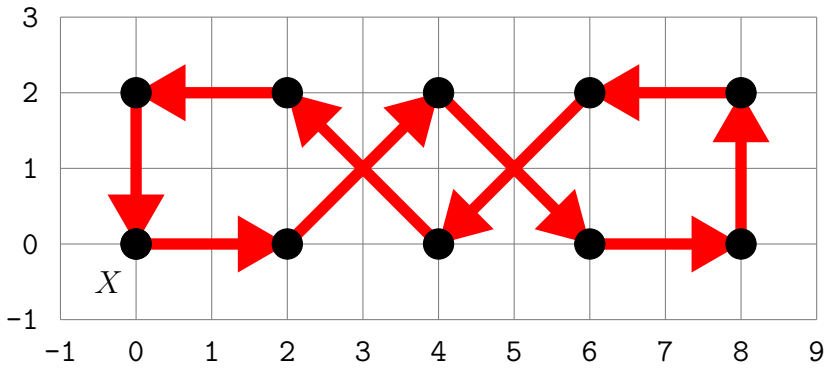
and you permute the points at index 2 and 3, you get



$$[(0,0),(2,2),(0,2),(2,0),(0,0)]$$

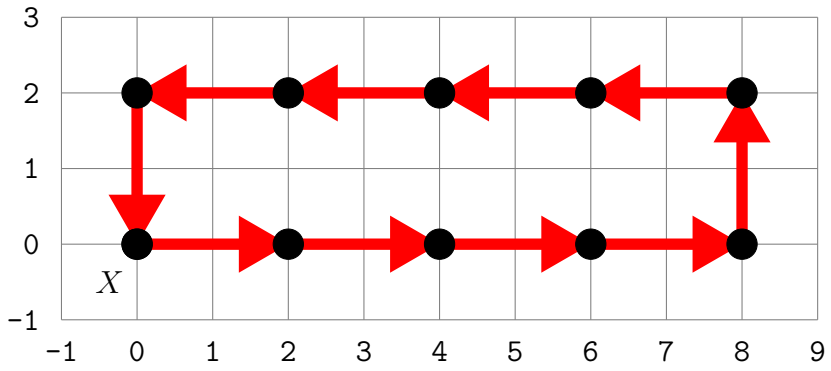
Method 2: opt-2

For this section, I will only draw the path and omit lines of all available roads. Suppose you have this this path:



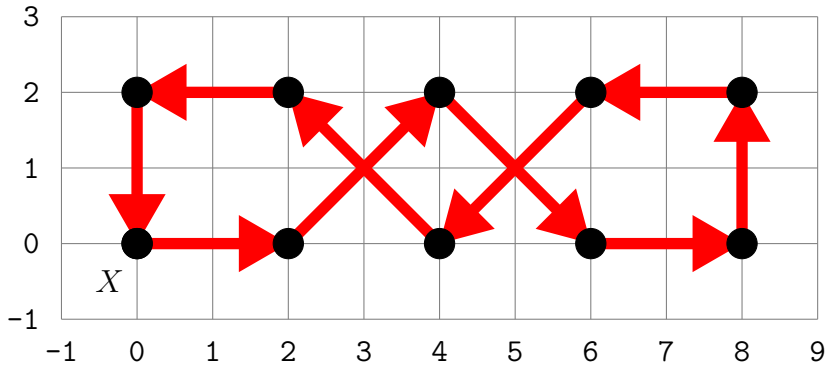
[(0, 0), (2, 0), (4, 2), (6, 0), (8, 0), (8, 2), (6, 2), (4, 0), (2, 2), (0, 2), (0, 0)]

If you permute the points of the path at index 2 and 7, you would get this:



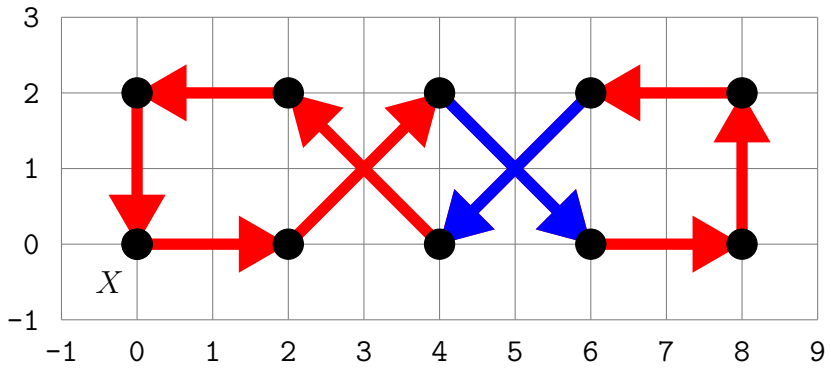
[(0, 0), (2, 0), (4, 0), (6, 0), (8, 0), (8, 2), (6, 2), (4, 2), (2, 2), (0, 2), (0, 0)]

This permute two points is from the previous section. There's another way to generate a successor. Pay attention to this because it's a very important idea recently discovered (around the late 50s). Here's the original path again:



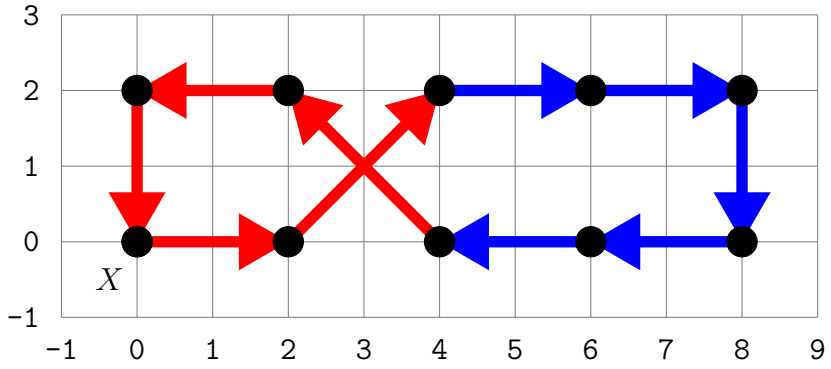
[(0, 0), (2, 0), (4, 2), (6, 0), (8, 0), (8, 2), (6, 2), (4, 0), (2, 2), (0, 2), (0, 0)]

Suppose I pick the points at index 2 and at index 6. I notice that the edge coming out of the point at index 2 (i.e. from index 2 to index 3) and the edge coming out of the point index 6 (i.e., from index 6 to index 7) crosses each other:



[(0, 0), (2, 0), (4, 2), (6, 0), (8, 0), (8, 2), (6, 2), (4, 0), (2, 2), (0, 2), (0, 0)]

I swap the points at index 3 and 6 *and* I reverse the path from index 3 to 6:



[(0, 0), (2, 0), (4, 2), (6, 2), (8, 2), (8, 0), (6, 0), (4, 0), (2, 2), (0, 2), (0, 0)]

(The intuition is that the operation attempts to disentangle one crossing at a time – it's a meaningful but a greedy heuristic.)

This is the second way of generating successors.

In the above, I pick two indices `i` and `j` and look at the two line segments of the path, from the point at index `i` to `i + 1` and from the point at index `j` to `j + 1`. I do need to check if these two line segments intersect. To check if one line segment containing points (a, b), (c, d) intersects another line segment containing points (A, B), (C, D) is pretty easy. (This is roughly Math106.) If they do intersect, you can compute the point of intersection, say (x, y). You still need to check that this point is on both line segments: the point could be *outside* one or both line segments. (This is the east way – there are other methods.)

Here are more details for the above hint. Given two distinct points, say, (a, b) and (c, d),

you can always find the equation of the line contains the above points. One way to write the equation of a line is like this:

$$y = mx + n$$

Of course m is the slope:

$$m = \frac{d-b}{c-a}$$

i.e.,

$$y = \frac{d-b}{c-a} \cdot x + n$$

If you put (a, b) into the equation, you'll get n:

$$b = \frac{d-b}{c-a} \cdot a + n$$

i.e.,

$$n = b - \frac{d-b}{c-a} \cdot a$$

Altogether this means that the equation is

$$y = \left(\frac{d-b}{c-a}\right) x + \left(b - \frac{d-b}{c-a} \cdot a\right)$$

This is possible except for the case when the line is vertical. The line is vertical when the x–coordinate of the above two points are the same (i.e., $a = c$), in which case the equation looks like

$$x = a$$

Both cases can be converted into this form

$$\alpha x + \beta y = \gamma$$

Specifically, if $a \neq c$, the equation is

$$-\left(\frac{d-b}{c-a}\right) x + 1 \cdot y = \left(b - \frac{d-b}{c-a} \cdot a\right)$$

and if $a = c$, then the equation is

$$1 \cdot x + 0 \cdot y = a$$

So anyway, you have two lines and therefore two equations:

$$\alpha x + \beta y = \gamma$$
$$\alpha' x + \beta' y = \gamma'$$

The point of intersection (x, y) is then given by

$$x = (\beta'\gamma - \beta\gamma')/\Delta$$
$$y = (-\alpha'\gamma + \alpha\gamma')/\Delta$$

where

$$\Delta = \alpha\beta' - \alpha'\beta$$

If Δ is 0, then there is no point of intersection. This happens when the two line are parallel or the two lines are actually the same.

Once you have the point of intersection, say (x, y), you have to check that it's on the first line segment. The two endpoints of the first line segment are (a, b) and (c, d). Assuming that $a < c$ (otherwise you have to swap their values), you need to check that

$$a < x < c$$

Likewise if $b < d$ (otherwise you swap their values), you have to check that

$$b < y < d$$

You also have to do the same for the second line segment. That's it. (There's another way to do the above that is slightly more efficient.)

TSP: OBJECTIVE FUNCTION

There's nothing much to say here: we use the sum of distances between two consecutive points in the path. The distance between two points are given by the Euclidean distance formula. Period. Don't forget that for implementation, the list of points does not repeat the first point.

(There is however a tiny optimization when it comes to the computation: If you already have the sum of distances d for a list of N points, then it's easy to see that if you permute two points, you only need to change four terms in d. So delete these four terms from d and add back to d the new four terms. I'll let you think about that.)

Don't forget that for local search you are optimizing the objective function. In some cases, optimizing means maximizing while in other cases optimizing means minimizing.

For the TSP, the goal is to minimize the objective function.

LOCAL SEARCH

To fix computations, we will assume the following:

General:

- An iteration is when we compute a successor state.
- The program will allow a user to specify the maximum number of iterations. (Note that in many cases in the real world, the termination condition might not be in terms of number of iterations, but in terms of a cutoff for the objective function or when the objective function is not improving at a high enough rate, or the cutoff might be in terms of clock time, etc., or it could be a combination of the above.)
- The local search function can return a better state (which for our case is a list of points) or it returns `False` to indicate that the local search has ended). For instance hill climbing will stop (and therefore returns `False`) if it cannot find a better state.
- Occasionally you will need to randomly select (in a uniform way) from a collection of "good" successor states. To standardize solutions, this is how you should perform this random selection: First sort your list of good solutions. Second call `random.choice` on the list. For instance if the list is `good_succ_states`, then you return `random.choice(sorted(good_succ_states))`.

Specific:

- Hill climbing:
  - This is straightforward. If you do have more than one state with the same optimum objective value, sort the optimal states and return a random one using `random.choice`.
- Stochastic hill climbing.
  - Recall that you need to collect all states that achieve better objective function values than the given one. You then randomly pick one from the above collection so that one with a higher objective function value has a higher chance of being selected. This can be done, for instance, in the following way. Suppose I have a sequence of values:

    $$10, 12, 13, 17, 34$$

    Think of 10 as the base. The distance of the values from the base is

    $$0, 2, 3, 7, 24$$

Consider the following intervals:

$$[10, 10+0)$$
$$[10+0, 10+2)$$
$$[10+2, 10+2+3)$$
$$[10+2+3, 10+2+3+7)$$
$$[10+2+3+7, 10+2+3+7+24)$$

In other words the intervals are

$$[10, 10), [10, 12), [12, 15), [15, 22), [22, 46]$$

Now what I can do is this: I generate a random number x in [10, 46]. If x falls in, say, the third interval $[12, 15)$, then I pick the 13 in the list $10, 12, 13, 17, 34$. That's it.

Let me give you some hints that will help you implement the above idea.

Applying the above to any local search, we do this. Say you have states $s_0, s_1, ..., s_4$ with objective function values $10, 12, 13, 17, 34$ which are greater than the objective function value of the input state (assuming that you're maximizing the objective function.) Using the idea above, if you generate a random number that indicates you should pick the objective function value of 13, then you pick s_2. Therefore the list of successor states should be collected together with their objective function values like this:

$$[(10, s_0), (12, s_1), (13, s_2), (17, s_3), (34, s_4)]$$

It's common to scale your interval from $[10, 46]$ to $[0.0, 1.0]$ so that you generate a random float in $[0.0, 1.0]$. There are two more implementation details to think about ...

Technically speaking SHC (stochastic hill climbing) must choose a new state with strictly higher objective function value. Note that if $s_0, s_1, ..., s_4$ are states which have objective function values larger than the input state (for the case when you're maximizing the objective function), then the probabilty of choosing s_0 is almost zero. So if you think carefully about this, it's really unfair to s_0 in many cases. For instance what if your input state has objective function value of 10 and s_0 has value 1000, s_1 has value 1000.1, etc.? The obvious way of overcoming this is pretty obvious: You let s_0 be the input state, i.e., you include the input state in the set of states for your randomized selection. But if you do generate a random number of 0.0 – which means picking s_0 – you set your x to 1.0. Get it? In fact the function that generates a random float, i.e., `random.random()` produces a float in the interval `[0.0, 1.0)` (check python documentation). So by doing the above switch-0.0-to-1.0, you would be generating a

random float in `(0.0, 1.0]` which is what you want if you want to avoid picking s_0. Right?

The second thing is this: In the above we assume that each state has a unique objective function value. What if there are two states with the same objective values? The obvious thing to do is to have collect lists of states for each objective function value. The randomly generate number will pick a list of states and you randomly pick one from this set. Right? Remember to pick a random state in this way: `random.choice(sorted(set_of_states))` like before.

(Don't forget that in the above hints, I'm assuming that you're maximizing the objective function. For TSP, you are *minimizing* the function. The translation is pretty simple.)

- First choice hill climbing: This is pretty obvious and there's only one way of doing this. Just remember that when you want to pick two random points on your path, you pick their indices like this:

```
while 1:
    i = random.randrange(N)
    j = random.randrange(N)
    if i != j: break
```

Just remember that in the code for the list of points, the first point is not repeated at the end. Also, make sure that you return the state when the objective function value of the state is *strictly* greater than the objective function value of the input state.

For termination, in your algorithm, only allow the N attempts to search for a successor state where N is the number of distinct points on the path. For instance the algorithm could look something like this:

```
def first_choice_hillclimbing(obj, points, successor_method='p2'):
    N = len(points)
    if N == 1: return False

    for k in range(N):
        randomly find distinct i, j such that they are in [0,...,N-1]
        let tpoints be points but with  points[i], points[j] swapped
        if tpoints is better than points, return tpoints
    return False
```

Even better:

```
def first_choice_hillclimbing(obj, points, successor_method='p2',
                              maxtries=None):
    N = len(points)
    if N == 1: return False
    if maxtries == None:
        maxtries = N

    for k in range(maxtries):
        randomly find distinct i, j such that they are in [0,...,N-1]
        let tpoints be points but with  points[i], points[j] swapped
        if tpoints is better than points, return tpoints
    return False
```

This way, if you do not specify the maximum number of tries, it will default to the number of points in the path. In the real world, the termination condition is usually something like: if either the objective is sufficient optimized, or the change from the previous less than a certain quantity, or if the total time is reached, stop the search. For this assignment question, we are just setting the maximum number of tries to fix the output of your program.

- Random restart hill climbing. For random restart hill climbing, you need to get a new random state (i.e., random path) when you're stuck – i.e., when the state cannot be improved. Make sure do this: say your path is stored in variable `points`. Then on executing `random.shuffle(points)`, you will get a new random path. This step (of randomizing a new state) is also counted as an iteration of this local search method. Note that for this assignment the condition for restarting is when the state is stuck, i.e., the objective function value cannot be improved.

  Note that in terms of implementation, RRHC (random restart hill climbing) needs to keep track of the best previous path. (See slides/notes/AIMA.) You start off with one random path just like hill climbing. Once the path cannot be improved by hill climbing, you save the path somewhere, say as `prev_points`. You randomize `points` to get a new path and do another hill climbing. Once you cannot improve on the path `points`, you compare the `prev_points` and `points` and keep the one that's better as `prev_points`. You randomize `points` and do another round of hill climbing. Etc. RRHC requires remembering something – the best previous path – which means that it's different from the others which are basically simple functions without memory. So how do we create a random restart hill climbing function so that we can call:

```
random_restart_hillclimbing(obj, points, successor_method):
```

just like the others??? There are two ways of doing this. We can create an object that is callable (in C++–speak this means that the object has `operator()`):

```
class RRHC:
    def __init__(self):
        self.prev_points = None
    def __call__(self,
                 obj, points, successor_method='p2'):
        call hillclimbing on points
        if the objective value of points has improved, return points
        otherwise:
            if self.prev_points is None:
                store points in self.prev_points
            otherwise store the better of points and self.prev_points
                in self.prev_points
            restart by randomizing and returning points

random_restart_hillclimbing = RRHC()
```

That's it. But wait ... calling `random_restart_hillclimbing` with the relevant data might return an improved path, i.e. `points` but perhaps `random_restart_hillclimbing.prev_points` is even better. So when the search ends (for instance when the search has completed the maximum number of iterations) we need a way to get the object `random_restart_hillclimbing` to compare `points` against `random_restart_hillclimbing.prev_points` one last time and return the one that is better. So add a method called `final`:

```
class RRHC:
    def __init__(self):
        self.prev_points = None
    def __call__(self,
                 obj, points, successor_method='p2'):
        call hillclimbing on points
        if the objective value of points has improved, return points
        otherwise:
            if self.prev_points is None:
                store points in self.prev_points
            otherwise store the better of points and self.prev_points
                in self.prev_points
            restart by randomizing point and return points
    def final(self, obj, points):
```

```
        compare and return the better of points or self.prev_points
```

In your `main.py`, before you break out of the main while loop, you will need to call this `final` method:

```
...
while 1:

    ...

    iteration += 1
    if iteration > M:
        iteration = M
        if method == 'rrhc':
            points = localsearch.final(obj, points)
            graph.append((iteration, obj(points)))
            print("final %s: %s" % (iteration, obj(points)))
        break
    ...
```

(Of course you realize that this restart idea can be used with any basic local search algorithm. For instance we can restart first choice hill climbing too.)

I won't go into the second method other than saying that it involves something called closures.

- Simulated annealing. For simulated annealing, you need a time schedule. (See slides.)

  Here's how to create a successor state. Suppose the current state is s. Create a random successor say s'. If the objective function value of s' (i.e., distance of path) is $<$ the objective function value of s, then we return s' since it's better. Suppose the objective function value of s' is not better than the objective function value of s, i.e. the difference D between the new and the old objective function value is ≥ 0. (Note that the inequality here is ≥ 0 while the inequality in < 0 in my slides because the goal in the slides was to maximize the objective function. For TSP, we're trying to minimize. So we have to switch the inequality.) Consider the number $e^{(-D/T)}$ where T is some number (which represents temperature). (We use $-D$ instead of D again because we're minimizing instead of maximizing the objective function.) We generate a random number x in $[0, 1]$. If $x < e^{-D/T} = 1/e^{D/T}$ we return s'. Otherwise we return s – i.e., no change. The T represents temperature. But the only important thing is that T starts large and moves toward 0. Why? When T is extremely large, $1/e^{D/T}$ is very close to 1 and then T is extremely small, $1/e^{D/T}$ is very close to 0. This means

that the likelihood of accepting a "bad" path is higher at the beginning than near the end.

There are many ways to initialize T and many ways to modify T for each iteration so that T moves toward 0. For this assignment, you must initialize T to 10^{50} and for each iteration, replace T by $T \cdot 0.95$. In other words if t is the number of iterations, then after t iterations, $T = 10^{50} \cdot 0.95^t$.

(Different local search problems require a different way to initialize T and a different way to modify T. Finding the right way to do this requires experimentation.)

(Note: From the above, it's clear that this local search requires the iteration number, so there's one extra parameter in this local search function. Right?)

Q2. Local search and the n–queens problem

The goal is to compare the following techniques for solve the n–queens problem:

1. Graph search algorithm. Specifically, backtracking search. You can use recursive backtrack or backtrack using graph search.
2. Local search algorithm

You need not use graphics – console output is enough. When the program begins, you ask the user for n (the size of the board). You then ask the user if he/she wants to solve is using bt (backtrack) or the local search method. (Note that for local search, you also need to ask for number of iterations.) When the algorithm ends, print the final state (the board) and the time taken. For computation of time, you can use the resource module.

```
from resource import *

r = getrusage(RUSAGE_SELF)
start = r.ru_utime + r.ru_stime

for i in range(5000):
    for j in range(5000):
        k = i * j

r = getrusage(RUSAGE_SELF)
end = r.ru_utime + r.ru_stime
print(end - start)
```