ITS336 Lecture 5.

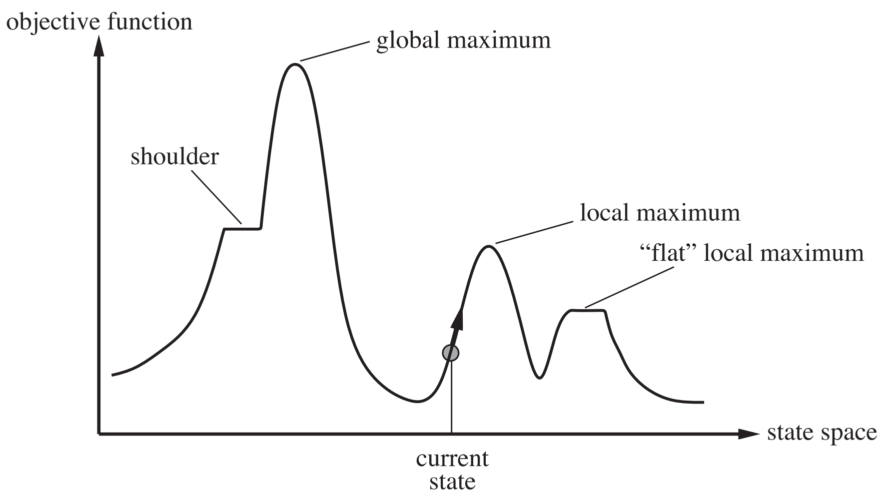
Local Search

5.1. Local Search Algorithms

In many search problems, the path from the initial state to the goal is **not** important, but the final configuration of the goal is the target of the problem, e.g., 8-queens problem. (4.1)

Local search is a group of algorithms that focus on this kind of problems. Since the path is not matter, local search algorithms typically keep only one node while operating.

Apart from searching for goal states, the algorithms can be used to solve **optimization problems**. The optimization problems aim to find the state with the best value of an **objective function**.



5.2. Hill Climbing Search

Hill climbing search algorithm repeatedly moves the current node to the direction that increases (or decreases) the objective value. It ends when the current node reaches a peak where no neighbor has a higher value. (4.1.1)

Code 5.1 Hill Climbing Algorithm implemented in Python

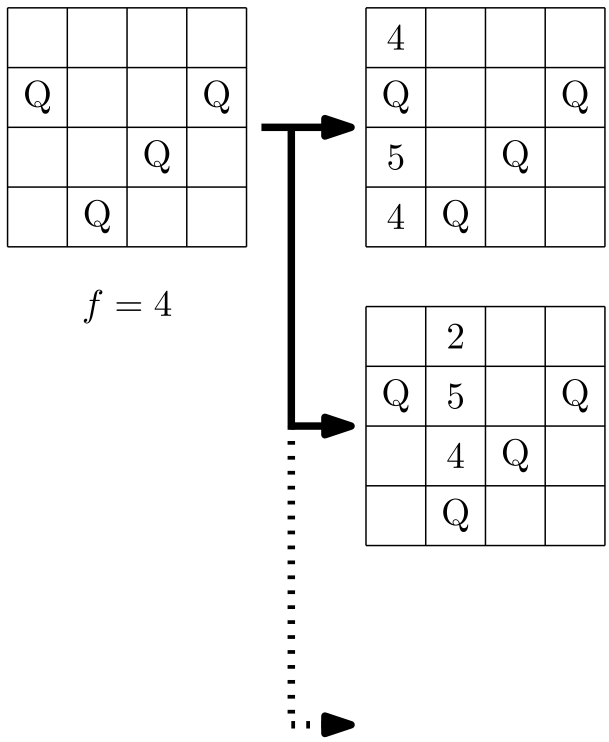
```
def hill_climbing(s0, succ, f):
    # s0   = initial state
    # succ = successor function
    # f    = evaluation function
    u = s0
    while True:
        s = succ(u)
        v = max(s, key=f)
        if f(v) > f(u):
            u = v
        else:
            return u
```

Example 5.1 To solve the 8-queens problem by the Hill Climbing algorithm, we need to define a function that evaluates each state.

We can use the number of attacks among the queens on the board as an evaluation value of a state. Thus, $f(s)$ returns the number of attacks given a state s. The search can start from a randomly created board with one queen placed in each column.

A successor state can be generated by selecting one queen and change its row position.

Here, we search for the state with $f = 0$.



Exercise 5.1 A local search problem represents a state by an ordered pair. The initial state is $(0,0)$. The successors of a state (x,y) are $(x-1,y)$, $(x+1,y)$, $(x,y-1)$, and $(x,y+1)$. The objective function is shown in the following figure. Use the hill climbing algorithm to search for the state with maximum objective value.

	0	1	2	3	4	5	6	7	8	9
0	3	4	4	4	5	4	5	5	5	4
1	4	5	5	5	4	4	5	6	5	3
2	5	6	7	6	4	3	6	7	6	2
3	5	6	6	6	4	2	6	6	6	2
4	5	5	5	4	4	1	6	5	5	3
5	3	3	4	3	7	7	7	6	4	4
6	4	3	4	3	7	8	7	8	7	5
7	4	5	5	4	6	7	9	8	7	5
8	4	5	5	4	6	8	8	8	7	6
9	3	3	3	4	6	6	7	7	7	5

5.2.1. Random-restart Hill Climbing

To avoid getting stuck in local maxima, the hill climbing search can be conducted multiple times. In each iteration, it starts from different randomly generated initial states.

Exercise 5.2 Conduct the random-restart hill climbing on the following space.

	0	1	2	3	4	5	6	7	8	9
0	3	4	4	4	5	4	5	5	5	4
1	4	5	5	5	4	4	5	6	5	3
2	5	6	7	6	4	3	6	7	6	2
3	5	6	6	6	4	2	6	6	6	2
4	5	5	5	4	4	1	6	5	5	3
5	3	3	4	3	7	7	7	6	4	4
6	4	3	4	3	7	8	7	8	7	5
7	4	5	5	4	6	7	9	8	7	5
8	4	5	5	4	6	8	8	8	7	6
9	3	3	3	4	6	6	7	7	7	5

Code 5.2 Random Restart Hill Climbing Algorithm

```
from hillclimbing import *


def random_restart_hill_climbing(succ, f, rnd, n):
    # succ = successor function
    # f    = evaluation function
    # rnd  = function that randomly generates state
    # n    = number of iterations
    best_value = -float("inf")
    for i in range(n):
        s0 = rnd()
        u = hill_climbing(s0, succ, f)
        if f(u) > best_value:
            best_state = u
            best_value = f(u)
    return best_state
```

5.2.2. Stochastic Hill Climbing Search

The hill climbing search basically goes directly to the closest local maximum with exploring the search space.

To allow the search to explore the space, we use the probability p to judge if a newly generated state should be accepted.

$$p = \frac{1}{1 + e^{\frac{current - new}{T}}}$$

where T is the parameter to control how easy the worse state is selected.

Code 5.3 Stochastic Hill Climbing Algorithm

```
import random
import math


def stochastic_hill_climbing(s0, succ, f, n, t):
    # succ = successor function
    # f    = objective function
    # n    = number of iterations
    # t    = control parameter
    best_value = -float("inf")  # negative infinity
    u = s0                      # u = currrent state
    for i in range(n):
        s = succ(u)             # generate a set of successors
        v = random.choice(s)    # randomly choose one to be the next state
        p = 1/(1+math.exp((f(u)-f(v))/t))   # calculate probability
        r = random.uniform(0.0, 1.0)        # randomly pick "r"
        if r < p:               # move to "v" if r < p
            u = v
        if f(u) > best_value:   # keep the best state up to now
            best_state = u
            best_value = f(u)
    return best_state
```

Consider the following values of p based on the different values of T

T	$e^{\frac{-13}{T}}$	p	$e^{\frac{13}{T}}$	p
1	0.00000226	1.000	442413.39	0.00000226
5	0.0498	0.953	13.5	0.0691
10	0.273	0.786	3.67	0.214
20	0.522	0.657	1.92	0.343
50	0.771	0.565	1.30	0.435
10^{10}	1.00	0.500	1.00	0.500

5.3. Simulated Annealing

Simulated annealing is based on an idea from thermodynamics.

- To grow a crystal, we heat the materials to a molten state. Then, cool it down until the crystal structure is frozen in.
- If the cooling is done too quickly, the materials become brittle.

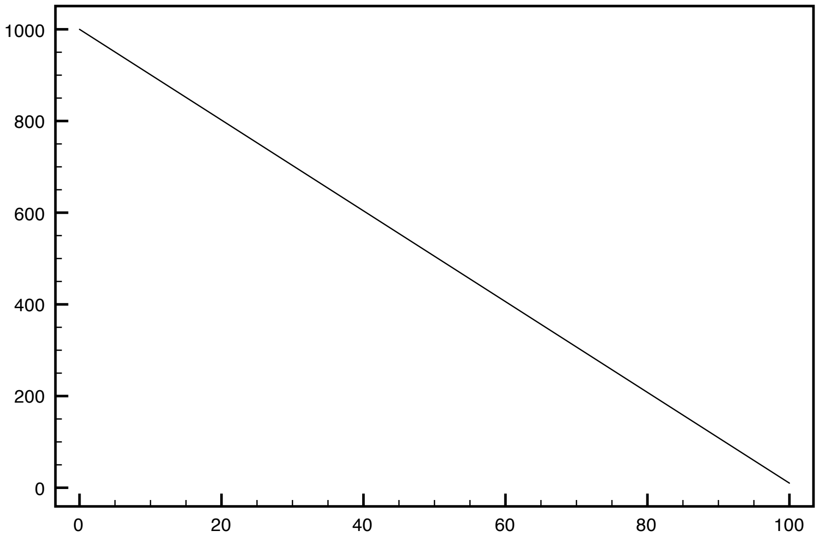
The parameter T in the stochastic hill-climbing can be compared to the temperature. We start with T in a high value to allow random walks, and we decrease T when the search is going on. This allows the exploration of the search space at the beginning, then it goes to the optimal state in the end.

Code 5.4 Simulated Annealing Algorithm

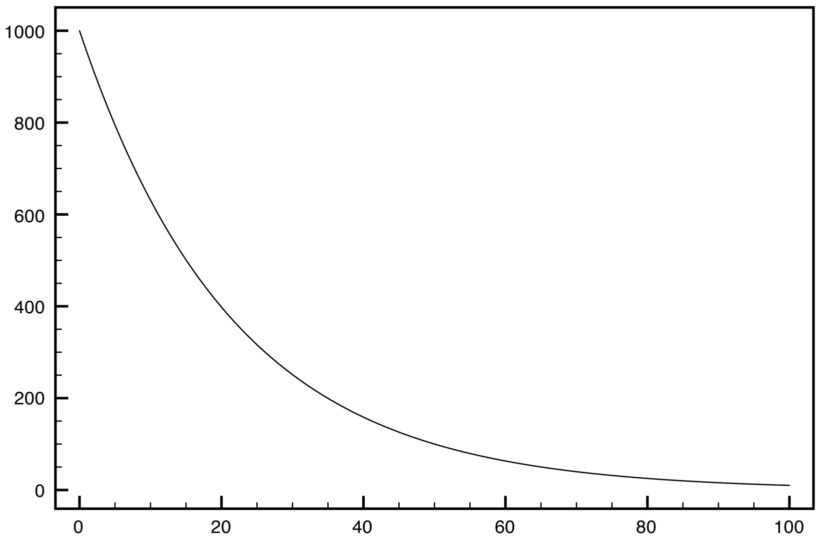
```
import random
import math


def simulated_annealing(s0, succ, f, t0, cooling):
    # succ = successor function
    # f    = evaluation function
    # rnd  = function that randomly generates state
    # n    = number of iterations
    best_value = -float("inf")
    u = s0
    t = t0
    i = 0
    while t > 0:
        s = succ(u)
        v = random.choice(s)
        if f(u) > f(v):
            u = v
        else:
            p = 1/(1+math.exp((f(u)-f(v))/t))
            r = random.uniform(0.0, 1.0)
            if r < p:
                u = v
        i += 1
        t = cooling(t, i)
        if f(u) > best_value:
            best_state = u
            best_value = f(u)
    return best_state
```

5.3.1. Cooling Schedule

$$T_i = T_0 - i\frac{T_0 - T_N}{N}$$



$$T_i = T_0 \left(\frac{T_N}{T_0}\right)^{\frac{i}{N}}$$



5.4. Local Beam Search

Instead of keeping track of only 1 node, **local beam search** algorithm keeps track of k states while conducting the search. It starts from randomly generated k states. (4.1.3)

The algorithm selects the k best neighbors. It may select all of the neighbors if there are less than k states, and randomly select in the case of ties. Then, these k states are used in the next iteration.

Local beam search with k states is different from conducting k random restarts hill climbing search.

Exercise 5.3 Conduct the local beam search when the beam size is 3, and the initial states are $(0,0); (1,8); (7,4)$

	0	1	2	3	4	5	6	7	8	9
0	3	4	4	4	5	4	5	5	5	4
1	4	5	5	5	4	4	5	6	5	3
2	5	6	7	6	4	3	6	7	6	2
3	5	6	6	6	4	2	6	6	6	2
4	5	5	5	4	4	1	6	5	5	3
5	3	3	4	3	7	7	7	6	4	4
6	4	3	4	3	7	8	7	8	7	5
7	4	5	5	4	6	7	9	8	7	5
8	4	5	5	4	6	8	8	8	7	6
9	3	3	3	4	6	6	7	7	7	5

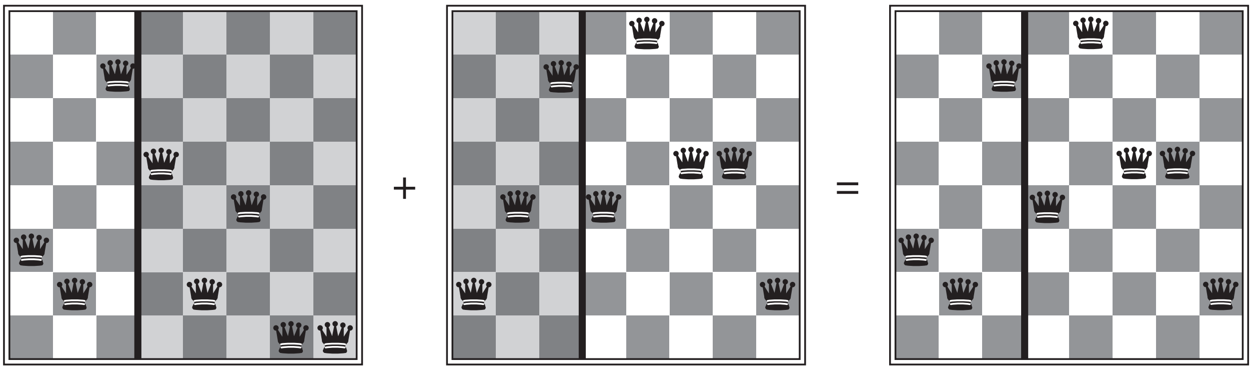
Stochastic beam search chooses k neighbors at random with higher evaluation states are more likely to be selected. Here, we define the probability p as

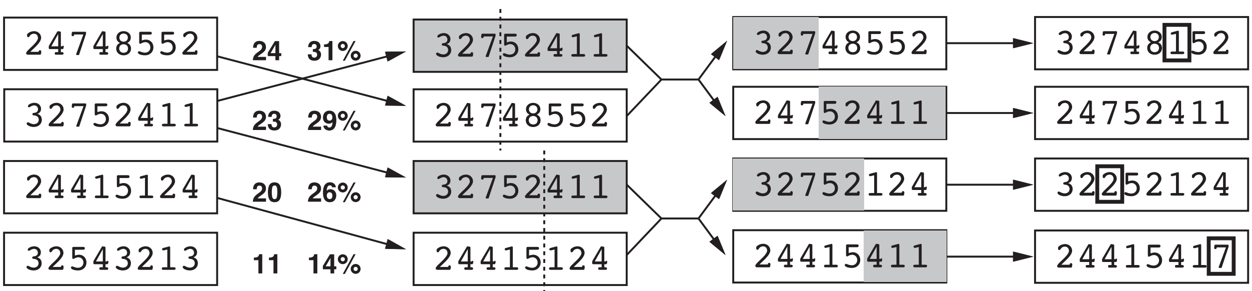
$$p = e^{-f(s)/T}$$

5.5. Genetic Algorithm

Genetic Algorithm is a variant of stochastic beam search. Different from other local search algorithms, GA generates successor states from a pair of states. (4.1.4)

Each state is denoted by a string over a finite set of alphabets, e.g. 0/1. Successor states are generated by two basic operations: crossover, and mutation. This is different from the other local search algorithms.





(a) Initial Population (b) Fitness Function (c) Selection (d) Crossover (e) Mutation

Code 5.5 Genetic Algorithm

```
import random, math

def genetic_algorithm(population, f, n):
    # population = initial set of individuals
    # f = evaluation function
    # n = number of iterations
    new_population = []
    for i in range(n):
        for j in range(len(population)):
            v = evaluate(population, f)
            x = select(population, v)
            y = select(population, v)
            child = cross_over(x, y)
            r = random.uniform(0.0, 1.0)
            if r < 0.05:
                child = mutate(child)
            new_population.append(child)
        population = new_population
    return max(population, key=f)

def evaluate(population, f):
    return [f(p) for p in population]

def select(population, val):
    m = sum(val)
    r = random.uniform(0, m)
    c = 0
    for p, v in zip(population, val):
        c += v
        if c > r:
            return p

def cross_over(x, y):
    c = random.randint(0, len(x))
    return x[0:c] + y[c+1:len(x)]

def mutate(x):
    c = random.randint(0, len(x)-1)
    x[c] = random.randint(0, 1)
    return x
```

Exercise 5.4 To find the maximum value of the following function

$$f(x, y) = (1 - x^2)e^{-x^2-y} + x^3 - y^3$$

If x and y are set to vary between -3 and 3. How can we represent an individual?

References

Russell, S. and Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd edition). Pearson/Prentice Hall.

Michalewicz, F. and Fogel, D. B. (1998). How to Solve It: Modern Heuristics. Springer.