Multiple Sequence Alignment

Xin Xu

Dynamic Programming

Dynamic programming: Break up a problem into a series of overlapping problems, and build up solutions to larger and larger sub-problems. It's a little like greedy algorithm, but contrast to greedy, it can override past optimal local solution because it records all local solutions so that dynamic programming can always find the global optimal solution while greedy may result in local optimal solution.

Dynamic programming has three important parts: state, state transfer equation and boundary conditions, and I will explain my implementation in these three aspects.

Pairwise Alignment

State

The procession to go through each sequence is the state. In pairwise alignment, I use a 2D matrix to denote state. The axis of x means it has go through x chars of the sx sequence and it's the same with y.

State Transfer Equation

The state transfer equation is:

```
for line in range(1, lenx):
    cost_matrix[0][line] = line * GAP
    for row in range(1, leny):
        cost_matrix[row][line] = min([PairCost(sx[line], sy[row]) +
cost_matrix[row - 1][line - 1], GAP + cost_matrix[row - 1][line], GAP +
cost_matrix[row][line - 1]])
```

There are three conditions:

- x_i and y_i matches. The cost is consist of $x_i - y_i$ mismatch and past least cost.
- Leaving x_i unmatched. The cost is consist of GAP and past least cost.
- Leaving y_i unmatched. The cost is consist of GAP and past least cost.

The `Pairwise` function is:

```
def PairCost(sx, sy):
    if sx == sy:
        return MATCH
    elif sx == "-":
        return GAP
    elif sy == "-":
        return GAP
    else:
        return MISMATCH
```

Boundary Conditions

The boundary conditions happen when all sx sequence chars are matched with GAP or all sy sequence chars are matched with GAP:

```
for row in range(leny):
    cost_matrix[row][0] = row * GAP
for line in range(1, lenx):
    cost_matrix[0][line] = line * GAP
```

Result

The result and running are as follows:

```
The least cost of pairwise alignment by dynamic programming is:  112
The base sequences is:  KJXXJAJKPXKJJXJKPXKJXXJAJKPXKJJXJKPXKJXXJAJKPXKJXXJAJKHXKJXXJAJKPXKJXXJAJKHXKJXX
The best match is:  XHAPXJAJXXXJAJXDJAJXXXJAPXJAHXXXJAJXDJAJXXXJAJXXXJPPXJAJXXXHAPXJAJXXXJAJXXXJAJXXXJA
KJX---XJAJKPXKJ-JX-JKPXKJXXJA--J-KPXKJ-JX-JKPXKJXXJAJKPXKJ-XXJAJ--KH--X-KJ-XXJAJKPXKJXXJAJKHXKJXX--
--XHAPXJAJ-XXXJAJXDJAJX--XXJAPXJAHXXXJAJXDJAJX--XXJAJ-XXXJPPXJAJXXXHAPXJAJXXXJAJ--X--XXJAJ--X--XXJA
The running time of pairwise alignment by dynamic programming is: 0.495641s
```

```
The least cost of pairwise alignment by dynamic programming is:  107
The base sequences is:  ILOTGJJLABWTSTGGONXJMUTUXSJHKWJHCTOQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLPO
The best match is:  IPOTWJJLABKSGZGWJJKSPPOPHTSJEWJHCTOOTHRAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJPJO
ILOTGJJLABWTSTG-GONXJ-----MUTUXSJHKWJHCT-OQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLP-O
IPOTWJJLAB-KS-GZG-WJJKSPPOPHT--SJ-EWJHCTOOTH-RAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJ-PJO
The running time of pairwise alignment by dynamic programming is: 0.800917s
```

```
The least cost of pairwise alignment by dynamic programming is:  113
The base sequences is:  IHKKKRKKKKKKXGWGKKKPKSKKKKKBKKKPKHKXXKBSKKPKWKKLKSKRKKWXKPKKBKKKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK
The best match is:  IKBSKKKKWKKKKKKKKWWKKKKPGKKKKXXGGKRKWWKWKKPKKKKXKKKRWKMKKPKKWPKKKKKPGKKKLBKWWKKJ
IHKKKRKKKKKKXGWGKKKPKSKKKK--KBKKKP-KHKKXKK--BSK-K-PKWKKLKSKRKKWXKPKK-BK-KKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK-
I---K-BSKKKK--W-KKK-K-KKKKWWK-KKKPGK-KK-KKXXGGKRKWWKWKKPK-K-KKKXK-KKRWKMKKPK--KWPKKK---KK-PGKK---KLBKW---WKKJ
The running time of pairwise alignment by dynamic programming is: 0.595409s
```

```
The least cost of pairwise alignment by dynamic programming is:  148
The base sequences is:  MPPPJPXPGPJPPPXPPPJPJPPPXPPPPSPPJJJPPXXPPPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAPJSWPPJGA
The best match is:  OPJPXJPPMJPPMXPPMJPPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGPPXXPPOMPPXXPPOXPPXJPQXPPBJPPPXPPX
MPPPJPX-PGP-JPPPXPPPJPJPPPXPPPPSPPJJJPPXXP-PPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAP-JSWPP-JGA
--OPJPXJP-PMJPPMX-PP-MJ-PPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGP--PXXP-P----OM---PPXXPPOXPPXJP-QXPPBJ-PPPXPPX
The running time of pairwise alignment by dynamic programming is: 0.572254s
```

```
The least cost of pairwise alignment by dynamic programming is:  131
The base sequences is:  IPPVKBKXWXKHSAPHVXXVOJMRAKKPJVLLJBWKOLLJKXHGXLLCPAJOBKPGXBATGXMPOMCVZTAXVPAGKXGOMJQOLJGWGKXLQ
The best match is:  ITPVKWKSKXKXUAXPVHXVOMMKHYBPABLLOBGKOLLJGXZGXLSOLAMOGKIGXBATBXMPJTCVMTAXVMPWWAWOMOUPHHZBITKKXLK
IPPVKBKXWXKHSA-PHVXXVOJMRAK--KPJVLLJBWKOLLJKXHGXL--LCPAJOBKPGXBATGXMPOMCVZTAXV-P--AGKXGOMJQO----LJGWGKXLQ
ITPVKWKSKXKXUAXP-VHXVO-M-MKHYBPABLLOBGKOLLJGXZGXLSOL--AMOGKIGXBATBXMPJTCVMTAXVMPWWA---WOM--OUPHHZBITKKXLK
The running time of pairwise alignment by dynamic programming is: 0.574570s
```

Time Complexity

Suppose the longest length of sequences sx and sy is l. The time to construct a cost matrix is $O(l^2)$ and the time to trace the match path is $O(l)$. So, the total time complexity is $O(l^2)$.

Three-Sequence Alignment

The implementation is almost the same with pairwise alignment.

State

The procession to go through each sequence is the state. In three-sequence alignment, I use a 3D matrix to denote state. The axis of x means it has go through x chars of the sx sequence and it's the same with y and z.

State Transfer Equation

The state transfer equation is:

```
for i in range(1, leni):
    for j in range(1, lenj):
        for k in range(1, lenk):
            cost_matrix[i][j][k] = min([cost_matrix[i - 1][j - 1][k - 1] +
           ThreeSequenceCost(si[i],sj[j],sk[k]), \
                    cost_matrix[i - 1][j][k - 1] + ThreeSequenceCost(si[i],
"-", sk[k]), \
                        cost_matrix[i][j - 1][k - 1] + ThreeSequenceCost("-
", sj[j], sk[k]), \
                            cost_matrix[i - 1][j - 1][k] +
ThreeSequenceCost(si[i], sj[j], "-"), \
                                cost_matrix[i][j][k - 1] +
ThreeSequenceCost("-", "-", sk[k]), \
                                    cost_matrix[i][j - 1][k] +
ThreeSequenceCost("-", sj[j], "-"), \
                                        cost_matrix[i - 1][j][k] +
ThreeSequenceCost(si[i], "-", "-")])
```

There are three conditions:

- x_i, y_i and z_i matches. The cost is consist of x_i, y_i and z_i mismatch and past least cost.
- one of x_i, y_i and z_i unmatched. The cost is consist of one block '-' and two chars mismatch and past least cost.
- two of x_i, y_i and z_i unmatched. The cost is consist of two blocks '-' and one chars mismatch and past least cost.

The `ThreeSequenceCost` function is:

```
def ThreeSequenceCost(c1, c2, c3):
    return PairCost(c1, c2) + PairCost(c1, c3) + PairCost(c2, c3)
```

Boundary Conditions

Suppose the search space of three-sequence alignment in dynamic programming is a cube. The boundary conditions happen in the face of the cube: it reduces to the situation in 2D. So, the boundary condition is:

```
for i in range(leni):
    cost_matrix[i][0][0] = i * GAP * 2
for j in range(lenj):
    cost_matrix[0][j][0] = j * GAP * 2
for k in range(lenk):
    cost_matrix[0][0][k] = k * GAP * 2

for i in range(1, leni):
    for j in range(1, lenj):
        cost_matrix[i][j][0] = min(ThreeSequenceCost(si[i], sj[j], "-") +
cost_matrix[i - 1][j - 1][0], ThreeSequenceCost(si[i], "-", "-") +
cost_matrix[i - 1][j][0], ThreeSequenceCost("-", sj[j], "-") +
cost_matrix[i][j - 1][0])
for i in range(1, leni):
    for k in range(1, lenk):
```

```
        cost_matrix[i][0][k] = min(ThreeSequenceCost(si[i], "-", sk[k]) +
cost_matrix[i - 1][0][k - 1], ThreeSequenceCost(si[i], "-", "-") +
cost_matrix[i - 1][0][k], ThreeSequenceCost("-", "-", sk[k]) +
cost_matrix[i][0][k - 1])
for k in range(1, lenk):
    for j in range(1, lenj):
        cost_matrix[0][j][k] = min(ThreeSequenceCost("-", sj[j], sk[k]) +
cost_matrix[0][j - 1][k - 1], ThreeSequenceCost("-", "-", sk[k]) +
cost_matrix[0][j][k - 1], ThreeSequenceCost("-", sj[j], "-") +
cost_matrix[0][j - 1][k])
```

Result

The result and running time are as follows:

```
The least cost of three-sequence alignment by dynamic programming is:  456
The base sequences are:  ['IPZJJLMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWSLUKOBSOPLOOKUKSARPPJ', 'IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJPKSTJJUWXHGP
HGALKLPJTPJPGVXPLBJHHJPKWPPDJSG']
The best match is:  IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH
I----PZJ-J--LMLT-K------JULOS-TK--T-----J-OGLKJ-OBLTXGK-TPLUWWKOMOYJBGALJUKLGLOSVHWBPG-WSLUKOBSOPLOOKUKSARPP-J--
IWTJBGTJGJTWGBJTPKHAXHAGJ--JSJJPPJAPJHJHJHJHJHJHJHJHJPKSTJJU-W--XHGPHGAL--K--L-PJTPJPGVXPL---B---JHHJPK--WPPDJSG
IWTJBGTJGJTWGBJTPKHAXHAGJ--JXJJKPJTPJHJHJHJHJHJHJHJHJHKUTJJU-W--XHGHHGAL--K--L-PJTPJPGVXPL---B---JHH-----------
The running time of three-sequence alignment by dynamic programming is: 401.121597s
```

Time Complexity

Suppose the longest length of sequences sx, sy and sz is l. The time to construct a cost matrix is $O(l^3)$ and the time to trace the match path is $O(l)$. So, the total time complexity is $O(l^3)$.

A-star Algorithm

A-star algorithm is a little like BFS search. But it uses heuristic funtion $h(n)$ to estimate the total cost by equation $f(n) = g(n) + h(n)$. So, it can cut off many unnecessary branches to shrink search space.

I will explain my implementation from heuristic function and search order.

Pairwise Alignment

Heuristic Function

The heuristic function is:

```
def Heuristic(self, current, goal):
    return abs((goal[0] - current[0]) - (goal[1] - current[1])) * GAP
```

This heuristic function is definitely true because the least cost situations happens when all chars in shortest sequence are matched.

Search Order

I use a priority queue to decide the search order: the least cost, the earlier to be searched. In order to avoid loop search, the algorithm should record points it has passed through. And the update happens when a new point to be visited or the cost points already in queue decreases.

The code is as follows:

```
def Astar_Pairwise(self):
    frontier = PriorityQueue()
    frontier.put(self.start, 0)
    self.came_from[self.start] = None
```

```
    self.cost_so_far[self.start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == self.goal:
            return self.cost_so_far[current]

        for next in self.neighbors(current):
            new_cost = self.cost_so_far[current] + self.Cost(current, next)
            if next not in self.cost_so_far or new_cost <
self.cost_so_far[next]:
                self.cost_so_far[next] = new_cost
                priority = new_cost + self.Heuristic(next, self.goal)
                frontier.put(next, priority)
                self.came_from[next] = current
```

Result

The result and running time are as follows:

```
The least cost of pairwise alignment by Astar Algorithm is:  112
The base sequences is:  KJXXJAJKPXKJJXJKPXKJXXJAJKPXKJJXJKPXKJXXJAJKPXKJXXJAJKHXKJXXJAJKPXKJXXJAJKHXKJXX
The best match is:  XHAPXJAJXXXJAJXDJAJXXXJAPXJAHXXXJAJXDJAJXXXJAJXXXJPPXJAJXXXHAPXJAJXXXJAJXXXJAJXXXJA
KJX---XJAJKPXKJ-JX-JKPXKJXXJA--J-KPXKJ-JX-JKPXKJXXJAJKPXKJ-XXJAJ--KH--X-KJ-XXJAJKPXKJXXJAJKHXKJXX--
--XHAPXJAJ-XXXJAJXDJAJX--XXJAPXJAHXXXJAJXDJAJX--XXJAJ-XXXJPPXJAJXXXHAPXJAJXXXJAJ--X--XXJAJ--X--XXJA
The running time of pairwise alignment by A-star algorithm is: 8.726464s
```

```
The least cost of pairwise alignment by Astar Algorithm is:  107
The base sequences is:  ILOTGJJLABWTSTGGONXJMUTUXSJHKWJHCTOQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLPO
The best match is:  IPOTWJJLABKSGZGWJJKSPPOPHTSJEWJHCTOOTHRAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJPJO
ILOTGJJLABWTSTG-GONXJ-----MUTUXSJHKWJHCT-OQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLP-O
IPOTWJJLAB-KS-GZG-WJJKSPPOPHT--SJ-EWJHCTOOTH-RAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJ-PJO
The running time of pairwise alignment by A-star algorithm is: 9.747117s
```

```
The least cost of pairwise alignment by Astar Algorithm is:  113
The base sequences is:  IHKKKRKKKKKKXGWGKKKPKSKKKKKBKKKPKHKXXKBSKKPKWKKLKSKRKKWXKPKKBKKKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK
The best match is:  IKBSKKKKWKKKKKKKKWWKKKKPGKKKKXXGGKRKWWKWKKPKKKKXXKKRWKMKKPKKWPKKKKKPGKKKLBKWWKKJ
IHKKKRKKKKKKXGWGKKKPKSKKKK--KBKKKP-KHKXXKK--BSK-K-PKWKKLKSKRKKWXKPKK-BK-KKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK-
I---K-BSKKKK--W-KKK-K-KKKKWWK-KKKPGK-KK-KXXXGGKRKWWKWKKPK-K-KKKXK-KKRWKMKKPK--KWPKKK---KK-PGKK---KLBKW---WKKJ
The running time of pairwise alignment by A-star algorithm is: 10.221536s
```

```
The least cost of pairwise alignment by Astar Algorithm is:  148
The base sequences is:  MPPPJPXPGPJPPPXPPPJPJPPPXPPPPSPPJJJPPXXPPPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAPJSWPPJGA
The best match is:  OPJPXJPPMJPPMXPPMJPPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGPPXXPPPOMPPXXPPOXPPXJPQXPPBJPPPXPPPX
MPPPJPX-PGP-JPPPXPPPJPJPPPXPPPPSPPJJJPPXXP-PPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAP-JSWPP-JGA
--OPJPXJP-PMJPPMX-PP-MJ-PPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGP--PXXP-P----OM---PPXXPPOXPPXJP-QXPPBJ-PPPXPPX
The running time of pairwise alignment by A-star algorithm is: 9.700976s
```

```
The least cost of pairwise alignment by Astar Algorithm is:  131
The base sequences is:  IPPVKBKXWXKHSAPHVXXVOJMRAKKPJVLLJBWKOLLJKXHGXLLCPAJOBKPGXBATGXMPOMCVZTAXVPAGKXGOMJQOLJGWGKXLQ
The best match is:  ITPVKWKSKXKXUAXPVHXVOMMKHYBPABLLOBGKOLLJGXZGXLSOLAMOGKIGXBATBXMPJTCVMTAXVMPWWAWOMOUPHHZBITKKXLK
IPPVKBKXWXKHSA-PHVXXVOJMRAK--KPJVLLJBWKOLLJKXHGXL--LCPAJOBKPGXBATGXMPOMCVZTAXV-P--AGKXGOMJQO----LJGWGKXLQ
ITPVKWKSKXKXUAXP-VHXVO-M-MKHYBPABLLOBGKOLLJGXZGXLSOL--AMOGKIGXBATBXMPJTCVMTAXVMPWWA---WOM--OUPHHZBITKKXLK
The running time of pairwise alignment by A-star algorithm is: 9.351278s
```

Time Complexity

In worst case, the A-star algorithm will pass through all points. So, suppose the longest length of sequence is l, the time complexity is $O(l^2)$.

Three-Sequence Alignment

The implementation is almost the same with pairwise alignment.

Heuristic Function

The heuristic function of three-sequence alignment is:

```
def Heuristic(self, current, goal):
    return abs((goal[0] - current[0]) - (goal[1] - current[1])) * GAP +
abs((goal[0] - current[0]) - (goal[2] - current[2])) * GAP + abs((goal[1] -
current[1]) - (goal[1] - current[1])) * GAP
```

It uses the same method with pairwise alignment. And it notices this quality: The sum of pairwise least cost is less or equal to the cost of three-sequence alignment.

Search Order

It is almost the same with pairwise alignment:

```
def Astar_ThreeAlignment(self):
    frontier = PriorityQueue()
    frontier.put(self.start, 0)
    self.came_from[self.start] = None
    self.cost_so_far[self.start] = 0

    while not frontier.empty():
        current = frontier.get()

        if current == self.goal:
            return self.cost_so_far[current]

        for next in self.neighbors(current):
            new_cost = self.cost_so_far[current] + self.Cost(current, next)
            if next not in self.cost_so_far or new_cost <
self.cost_so_far[next]:
                self.cost_so_far[next] = new_cost
                priority = new_cost + self.Heuristic(next, self.goal)
                frontier.put(next, priority)
                self.came_from[next] = current
```

Result

The result and running time are as follows:

```
The least cost of three-sequence alignment by Astar Algorithm is:  456
The base sequences are:  ['IPZJJLMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWSLUKOBSOPLOOKUKSARPPJ', 'IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJPKSTJJUWXHGP
HGALKLPJTPJPGVXPLBJHHJPKWPPDJSG']
The best match is:  IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH
I----PZJ-J--LMLT-K------JULOS-TK--T-----J-OGLKJ-OBLTXGK-TPLUWWKOMOYJBGALJUKLGLOSVHWBPG-WSLUKOBSOPLOOKUKSARPP-J--
IWTJBGTJGJTWGBJTPKHAXHAGJ--JSJJPPJAPJHJHJHJHJHJHJHJHJPKSTJJU-W--XHGPHGAL--K--L-PJTPJPGVXPL---B---JHHJPK--WPPDJSG
IWTJBGTJGJTWGBJTPKHAXHAGJ--JXJJKPJTPJHJHJHJHJHJHJHJHKUTJJU-W--XHGHHGAL--K--L-PJTPJPGVXPL---B---JHH-----------
The running time of three-sequence alignment by A-star algorithm is: 3108.042706s
```

Time Complexity

In worst case, the A-star algorithm will pass through all points. So, suppose the longest length of sequence is l, the time complexity is $O(l^3)$.

Genetic Algorithm

Genetic algorithm is a particular class of evolutionary algorithms that are use techniques inspired by evolutionary biology such as inheritance, mutation, selection and crossover. By using this algorithm, we don't need to know how to get a best solution; we just need to follow several steps which randomly transfer one solution to other solutions and reserve the best individuals to

reproduce the next generation until end condition happens. It's random to get an optimal solution.

In MSA, I use the position of $'-'$ in a sequence to denote a match. For example, $(0, 3, 5, 8)$ means $-CT-C-CT-AG$.

I have used randomly initialization and greedy initialization in start. But in practice, there isn't much difference in performance between them.

The fitness of each solution is the same with dynamic programming.

Pairwise Alignment

The Process of Reproduce

```
def Reproduce(self):
    for i in range(ceil(self.population * self.cross_rate)):
        # select two individuals at random
        father = randint(0, len(self.group) - 1)
        mother = randint(0, len(self.group) - 1)
        while father == mother:
            mother = randint(0, len(self.group) - 1)

        x = self.group[father] # individual
        y = self.group[mother]
        # crossover these two individuals
        self.Crossover(x, y)

        # mutation
        self.Mutation(x)
        self.Mutation(y)

        # add new children to the mate pool
        self.group.append(x)
        self.group.append(y)

        self.group_fitness.append(self.Individual_Fitness(x))
        self.group_fitness.append(self.Individual_Fitness(y))
```

Crossover

```
def Crossover(self, x, y):
    for i in range(2):
        point = self.CrossoverRange(x[i], y[i]) # get the range to crossover
        pos = 0
        while x[i][pos] < point: # crossover between x and y
            tmp = x[i][pos]
            x[i][pos] = y[i][pos]
            y[i][pos] = tmp
            pos += 1
            if pos == len(x[i]):
                break
        x[i].sort() # sort the position of '-'
        y[i].sort()
```

And the $CrossoverRange$ is to find the range to perform crossover. Because the length of expanded sequence(expanded by '-') is fixed, I request the number of chars to perform crossover between x and y is the same. The function is as follows:

```
def CrossoverRange(self, father_block, mother_block):
    point = 1

    # find the point in the left of which there are the same number of block '-' between father and mother
    count_left_father = sum(i < point for i in father_block)
    count_left_mother = sum(i < point for i in mother_block)
    while count_left_father != count_left_mother or count_left_father == 0:
        point +=1
        count_left_father = sum(i < point for i in father_block)
        count_left_mother = sum(i < point for i in mother_block)

    return point
```

Mutation

```
def Mutation(self, individual):
    for seq in range(2):
        for i in range(len(individual[seq])): # for every block
            if random() < self.mutation_rate: # mutate in a probability of mutation_rate
                new_block = randint(0, self.seq_len - 1)
                while new_block in individual[seq]:
                    new_block = randint(0, self.seq_len - 1)
                individual[seq][i] = new_block

        individual[seq].sort()
```

Result

The result and running time are as follows:

```
The least cost of pairwise alignment by Genetic Algorithm is:  167
The base sequences is:  KJXXJAJKPXKJJXJKPXKJXXJAJKPXKJJXJKPXKJXXJAJKPXKJXXJAJKHXKJXXJAJKPXKJXXJAJKHXKJXX
The best match is:  IPHPJJHHJJPJJKPJJPHJJJKHJJJPPAJJJPPJJJKHJJJPPJJJKPJJJKPJJJKPJJJK
KJXXJ-AJKP-XKJJXJ--KPX-K-JXXJAJ-KPXKJJ-X-JKPXKJXXJAJKPXKJXX-JAJ-KH--XKJXXJAJKPXKJXXJA-JKHXK-JXX-
I--PH-PJJH-HJ-JPJ--J---KPJJ-PH-JJ-JKHJ-JJ-PPA-J-JJ-P--PJJ-J-KHJJJ-PP-JJJ--KPJJJ--K-PJJJKP-J-J-JK
The running time of pairwise alignment by genetic algorithm is: 174.507281s
```

```
The least cost of pairwise alignment by Genetic Algorithm is:  177
The base sequences is:  ILOTGJJLABWTSTGGONXJMUTUXSJHKWJHCTOQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLPO
The best match is:  IPOTWJJLABKSGZGWJJKSPPOPHTSJEWJHCTOOTHRAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJPJO
ILOTGJ-JL-AB-W-TSTGGONXJ-MUTU-XSJHKW--JHCTOQ-HWGAGIWLZHW--PKZULJ-TZW-AKBWHX-MIKLZJGLXBPAHOHVO-LZ-W-OSJJLPO-
IPOTWJJLABKS-GZGWJ-JKSPPOP-HTSJEWJH-CTOOTHR-AX--BKLBHWPKZU--LJP-ZK---AKVKHXUIKLZJGLX-BT--GHOHBJL-ZPPSJJ-PJO
The running time of pairwise alignment by genetic algorithm is: 152.765932s
```

```
The least cost of pairwise alignment by Genetic Algorithm is:  172
The base sequences is:  IHKKKRKKKKKXGWGKKKPKSKKKKKBKKKPKHKXXKKBSKKPKWKKLKSKRKKWXKPKKBKKKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK
The best match is:  IKBSKKKKWKKKKKKKKWWKKKKPGKKKKKXXGGKRKWWKWKKPKKKKKXXKKRWKMKKPKKWPKKKKKPGKKKLBKWWKKJ
IHKKKRKKKKKKXGWG--K-K-KPKSK-KK-KKBKKKPKHKK-XKKBSKKPKWKKLKS-KRKKWXKPKKBKKKPK-TSKHKK-KK-LADK--K-Y-PKKKOPH--K--KBWWLPPWK-K
-IKBSKKK-KWKKKKKK-K-K--W-WK-KKK---PGK-KK-KKXX-GG-KRKWWKWKK--PKKK-KKXKK-KR-W-KM----KKPKK-W---P-KKKKKP-GKK-K--LBKWW--KKJ-
The running time of pairwise alignment by genetic algorithm is: 193.140494s
```

```
The least cost of pairwise alignment by Genetic Algorithm is:  196
The base sequences is:  MPPPJPXPGPJPPPXPPPJPJPPPXPPPPSPPJJJPPXXPPPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAPJSWPPJGA
The best match is:  OPJPXJPPMJPPMXPPMJPPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGPPXXPPPOMPPXXPPOXPPXJPQXPPBJPPPXPPX
MPPPJP-XPG-PJPPPXPPPJPJPPP--XPPPPS-PP-JJJPPXXP-PP-PPJP-PPXPPXI-PJMMMX-PKPSVG-ULMHHZPAWH-T-HKAA-HHU-PAO-NA-P-JSW-PPJGA
-OP--JP--XJPPM-J-PPMXPPM-JPP-XJPPOXPPXJJP--JXX-PXJ-PP-O--J-PPM-XP-POG-PPX-XP-PO-MPPX-X-PPOX-PP--X--JPQX-PPB-JPPPXPPX-
The running time of pairwise alignment by genetic algorithm is: 173.687169s
```

```
The least cost of pairwise alignment by Genetic Algorithm is:  188
The base sequences is:  IPPVKBKXWXKHSAPHVXXVOJMRAKKPJVLLJBWKOLLJKXHGXLLCPAJOBKPGXBATGXMPOMCVZTAXVPAGKXGOMJQOLJGWGKXLQ
The best match is:  ITPVKWKSKXKXUAXPVHXVOMMKHYBPABLLOBGKOLLJGXZGXLSOLAMOGKIGXBATBXMPJTCVMTAXVMPWWAWOMOUPHHZBITKKXLK
IPPV-K-BKX--WXKHSA-PHVXX---VOJM-RAKKPJVLLJBWKOL-L-JKXHGX-LLCPA-JO-BKPGXBATGXMPOMC--VZT-AXVPAGKX-GOMJQOLJ-GWGKXL-Q-
ITPV--KWKSKXKXUAXPVHX-V-OMMK-HYBPABLLOB---GKO-L-LJGXZGXL-SO--LAMOG-K-IGXBATBX-MPJTCVM-TA-XVMPWWAWOMOU-PHHZBITKKXLK
The running time of pairwise alignment by genetic algorithm is: 168.689891s
```

We can compare these results to dynamic programming and A-star algorithm, it shows that although genetic algorithm cannot get the best match, it can find the best-matched sequence.

Time Complexity

Suppose the population of one generation is p, the longest length of sequence is l, the expansion rate is r_{sp}, the generation times is g, the crossover rate is r_c, the mutation rate is r_m. The time complexity in initialization is $O(pl)$. The time complexity in evolution is $O(g*(p*l+p\log p))$. So, the total time complexity is $O(g*(p*l+p\log p))$.

Three-Sequence Alignment

The implementation of three-sequence alignment is almost the same with pairwise alignment. So, I'm not going to explain it in here.

Result

The result and running time are as follows:

```
The least cost of three-sequence alignment by Genetic Algorithm is:  666
The base sequences are:  ['IPZJJLMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWSLUKOBSOPLOOKUKSARPPJ', 'IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJPKSTJJUWXHGP
HGALKLPJTPJPGVXPLBJHHJPKWPPDJSG']
The best match is:  IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH
IPZJJL---MLTKJUL--OSTKTJOGLKJOB-LTX-GKTP-LU-WWK-OMOY---J-B-GA-LJUK-LG-LOS-VHW--B-PG--WSLUKO-BSOPLOO-KUK-SARPPJ---
I-WTJBGTJGJT-WGBJTPKHAXHAGJJ-SJJPPJAP-JH-JHJHJHJHJ--HJHJHJPKST-JJ-U-WXHGPH----G-AL-KLPJ-TPJPGVXPLBJHHJPKWP-PDJSG-
-IWTJBGTJGJTWGBJTPK-H-AX-H-AGJJXJ-JKPJTPJ-H-JHJH----JHJHJHJ--HJ-HJH--KUTJJUWXH-GHH-GA-LK-LP--JTPJPG-V-X-PLB--J-HH
The running time of three-sequence alignment by genetic algorithm is: 348.643748s
```

Time Complexity

Use the same symbols in pairwise alignment. The time complexity in initialization is $O(pl)$. The time complexity in evolution is $O(g*(p*l+p\log p))$. So, the total time complexity is $O(g*(p*l+p\log p))$.

Reflection and Problems

- In A-star algorithm for three-sequence alignment, I use a more accurate heuristic function at first: the sum of pairwise actual cost. But after it runs for four hours, there isn't any signal to show its end. So, I pick up the old method used in pairwise alignment. It's more rough but simplier and faster. In this condition, although more accurate function results in less search space, the cost of precomputing in accurate function exceeds the save of less search space. So, it's inefficient. The first heuristic function of three-sequence is as follows:

```
1  def Heuristic(self, current, goal):
2      self.HeuristicMatrix()
3      return self.cost_matrix_xy[current[1]][current[0]] +
self.cost_matrix_yz[current[2]][current[1]] + self.cost_matrix_zx[current[0]]
[current[2]]
```

`HeuristicMatrix()` is to precomputer the dynamic cost matrix for each two sequences. It's the same with dynamic programming.

```
1  def HeuristicMatrix(self):
2      self.cost_matrix_xy = DP_MinCost_Matrix(self.sx, self.sy)
3      total_cost_xy = self.cost_matrix_xy[self.leny - 1][self.lenx - 1]
```

```
4
5       # the cost to reach goal - the cost to reach current point = needed
    least cost = h(n)
6       for row in range(len(self.cost_matrix_xy)):
7           for line in range(len(self.cost_matrix_xy[row])):
8               self.cost_matrix_xy[row][line] = total_cost_xy -
    self.cost_matrix_xy[row][line]
9
10      self.cost_matrix_yz = DP_MinCost_Matrix(self.sy, self.sz)
11      total_cost_yz = self.cost_matrix_yz[self.lenz - 1][self.leny - 1]
12
13      # the cost to reach goal - the cost to reach current point = needed
    least cost = h(n)
14      for row in range(len(self.cost_matrix_yz)):
15          for line in range(len(self.cost_matrix_yz[row])):
16              self.cost_matrix_yz[row][line] = total_cost_yz -
    self.cost_matrix_yz[row][line]
17
18      self.cost_matrix_zx = DP_MinCost_Matrix(self.sz, self.sx)
19      total_cost_zx = self.cost_matrix_zx[self.lenx - 1][self.lenz - 1]
20
21      # the cost to reach goal - the cost to reach current point = needed
    least cost = h(n)
22      for row in range(len(self.cost_matrix_zx)):
23          for line in range(len(self.cost_matrix_zx[row])):
24              self.cost_matrix_zx[row][line] = total_cost_zx -
    self.cost_matrix_zx[row][line]
```

- Although genetic algorithm cannot get the best match, it can find the best-matched sequence.
- I don't understand why my A-star algorithm runs longest. I think it would be the shortest. I suppose that the reason is STL of python. I use STL only in A-star algorithm, it makes codes clean and simple. But there may be some extra and inexplicit time for python to maintain the state of STL.
- I don't know how to configure the value of population, generation, crossover rate and mutation rate in genetic algorithm to get a better performance.