CS60045 Artificial Intelligence Autumn 2023

Local Search

Local Search

So far, we started from an initial state, and tried to reach a good goal state.

Both the goal state and the way of reaching the goal state from the initial state were important.

In many situations, the path from the initial state to the goal state is not important.

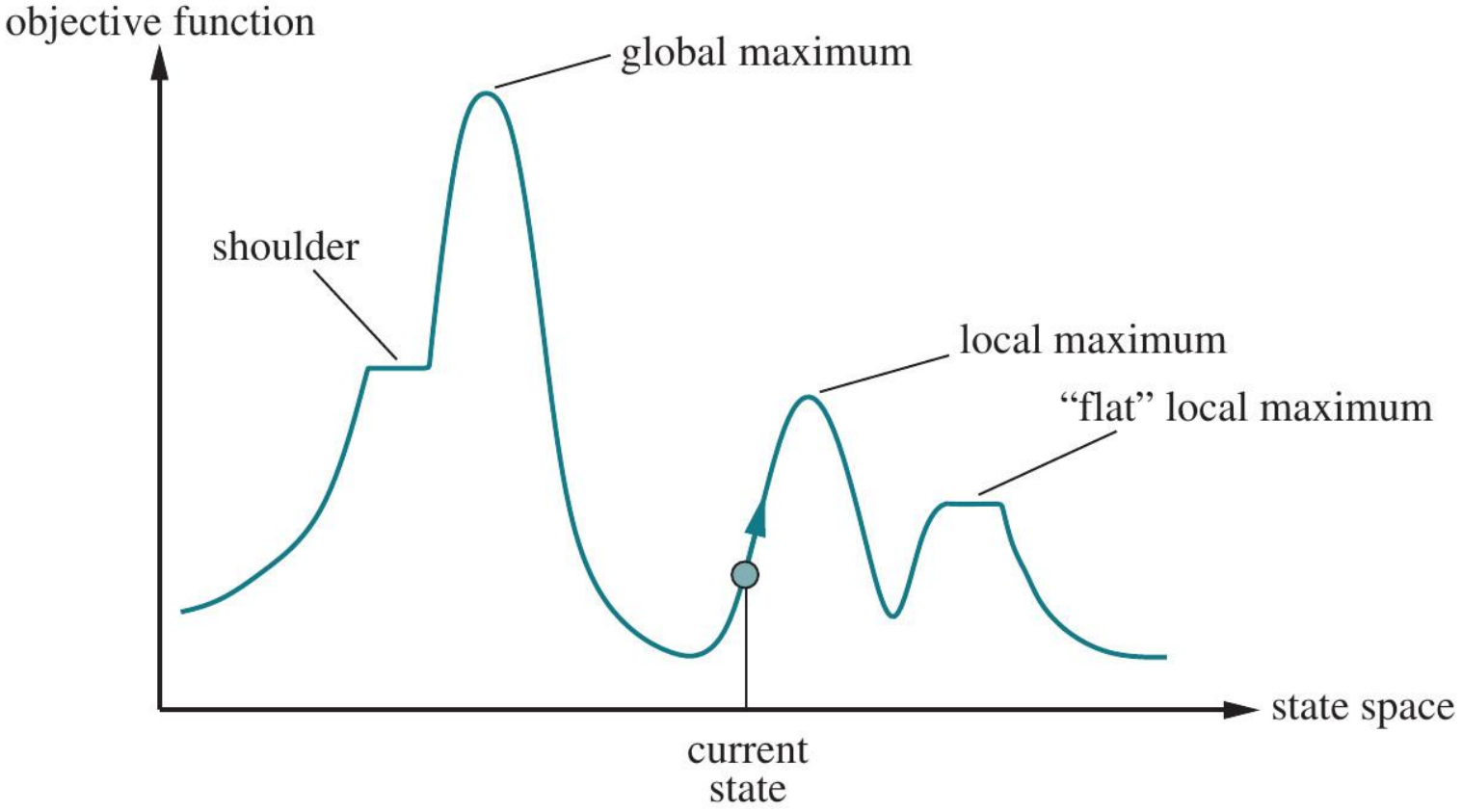
It suffices that we can find the goal state.

- Find a solution to a problem/puzzle.
  Example: The 8-queens on *n*-queens problem.
- Optimize certain objective function over a finite or infinite space.
  Example: Find an optimal TSP tour.

Local search starts at one or more initial states, and aims at moving closer and closer to the goal.

The path to reach the goal is not important, and is not remembered $\Rightarrow$ Low memory requirement.

State-space landscape (1-d, maximization) [Source: AIMA]



Moving toward the goal

Maximization problem

- Goal is to reach the global maximum
- Increase the objective function
- **Hill climbing**

Minimization problem

- Goal is to reach the global minimum
- Decrease the objective function
- **Gradient descent**

Functional problem

- Minimize distance to goal. Distance to goal is measured by a heuristic function *h* of the states.

Hill-Climbing Search (Greedy Local Search, GLS)

At every state, go to a neighboring state that maximizes/minimizes the objective function.

This is called **steepest ascent / descent**.

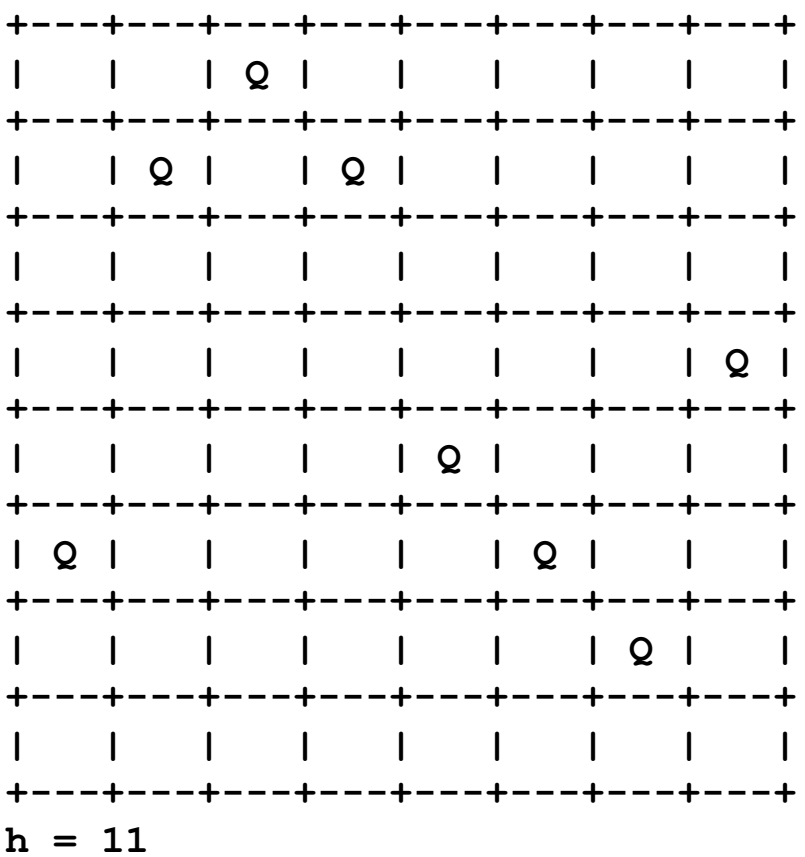
When the search reaches a peak where no neighboring states have better values for the objective function, stop.

But that peak may be a **local maximum or minimum**.

Example: 8-Queens Problem

> We plan to minimize the count of pairs of queens that can attack each other.This is equivalent to maximizing the negative of this count of pairs.
>
> The goal is to reach a state where this count reaches zero.
>
> In each column, we put a single queen. The state consists of the positions (row numbers) of the eight queens.

Solving the 8-Queens Problem: Initial board



Solving the 8-Queens Problem: Iteration 1

15	7	Q	10	11	8	9	11
12	Q	11	Q	11	9	10	13
12	6	14	10	10	6	10	10
11	8	11	14	9	10	9	Q
12	8	10	10	Q	7	13	11
Q	8	11	12	10	Q	10	15
11	8	11	10	11	7	Q	11
10	7	10	11	8	7	8	14

hmin = 6, imin = 2, jmin = 1

		Q					
			Q				
	Q						
							Q
				Q			
Q					Q		
						Q	

h = 6

Solving the 8-Queens Problem: Iteration 2

9	7	Q	7	7	4	5	6
7	11	7	Q	6	4	5	7
7	Q	10	7	7	3	7	6
7	8	8	9	5	6	5	Q
7	8	6	7	Q	3	9	6
Q	8	7	8	7	Q	6	10
6	8	7	6	7	4	Q	6
5	7	6	7	4	3	5	8

hmin = 3, imin = 2, jmin = 5

		Q					
			Q				
	Q				Q		
							Q
				Q			
Q							
						Q	

h = 3

Solving the 8-Queens Problem: Iteration 3

6	3	Q	5	5	4	3	5
5	6	4	Q	5	4	4	5
6	Q	7	5	6	Q	6	5
5	4	5	5	4	6	4	Q
5	4	3	5	Q	3	6	5
Q	3	4	4	4	6	3	7
4	5	4	3	4	4	Q	4
4	3	3	3	2	3	3	5

hmin = 2, imin = 7, jmin = 4

		Q					
			Q				
	Q				Q		
							Q
Q							
						Q	
					Q		

h = 2

Solving the 8-Queens Problem: Iteration 4

4	2	Q	4	5	3	3	4
4	4	3	Q	5	3	4	3
5	Q	5	4	6	Q	5	4
5	3	4	3	4	4	4	Q
3	3	1	3	3	1	5	4
Q	2	4	2	4	4	4	6
3	4	2	3	4	4	Q	3
4	2	3	3	Q	3	4	4

hmin = 1, imin = 4, jmin = 2

			Q				
	Q				Q		
							Q
		Q					
Q							
						Q	
				Q			

h = 1

Solving the 8-Queens Problem: Iteration 5

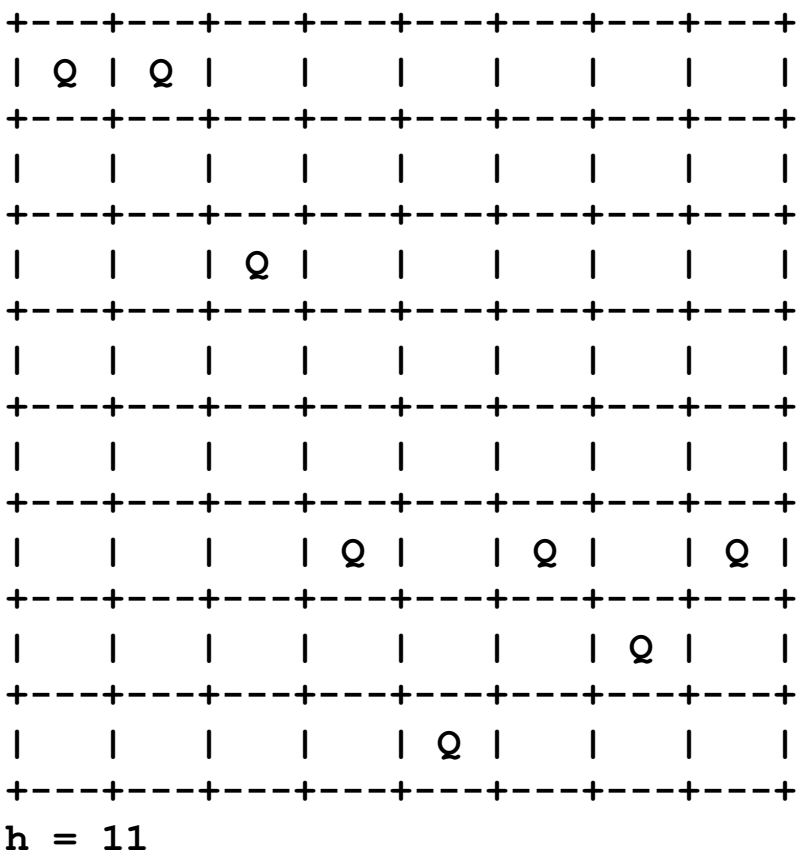
2	0	2	3	3	1	2	2
3	2	3	Q	4	3	3	2
4	Q	5	4	5	Q	4	3
4	3	4	4	3	2	3	Q
3	3	Q	4	3	1	4	4
Q	2	4	3	3	3	3	4
3	3	2	3	4	3	Q	2
3	1	3	3	Q	3	3	3

hmin = 0, imin = 0, jmin = 1

	Q						
			Q				
					Q		
							Q
			Q				
Q							
						Q	
				Q			

h = 0

A failed attempt to solve the 8-Queens Problem: Initial board



A failed attempt to solve the 8-Queens Problem: Iteration 1

Q	Q	11	11	14	8	9	10
8	14	9	11	11	6	7	9
9	11	Q	11	13	7	9	9
8	12	8	13	12	8	7	9
8	11	9	9	16	7	9	9
10	13	12	Q	14	Q	12	Q
8	11	10	11	14	8	Q	10
8	12	9	11	Q	10	8	13

hmin = 6, imin = 1, jmin = 5

Q	Q						
					Q		
		Q					
			Q				Q
						Q	
				Q			

h = 6

A failed attempt to solve the 8-Queens Problem: Iteration 2

Q	Q	7	7	10	8	6	6
5	9	6	8	7	Q	4	6
5	6	Q	7	9	7	6	5
4	7	4	9	7	8	3	5
4	6	6	5	10	7	4	5
5	8	7	Q	8	11	7	Q
5	6	6	7	8	8	Q	6
4	7	5	6	Q	10	4	8

hmin = 3, imin = 3, jmin = 6

Q	Q						
					Q		
		Q					
						Q	
			Q				Q
				Q			

h = 3

A failed attempt to solve the 8-Queens Problem: Iteration 3

Q	Q	5	5	7	5	6	4
3	5	4	5	5	Q	4	4
3	3	Q	4	6	5	6	4
3	5	3	6	5	6	Q	4
2	3	4	2	6	5	4	4
3	5	5	Q	6	7	7	Q
2	2	3	4	4	4	6	3
2	4	4	3	Q	6	4	5

hmin = 2, imin = 4, jmin = 0

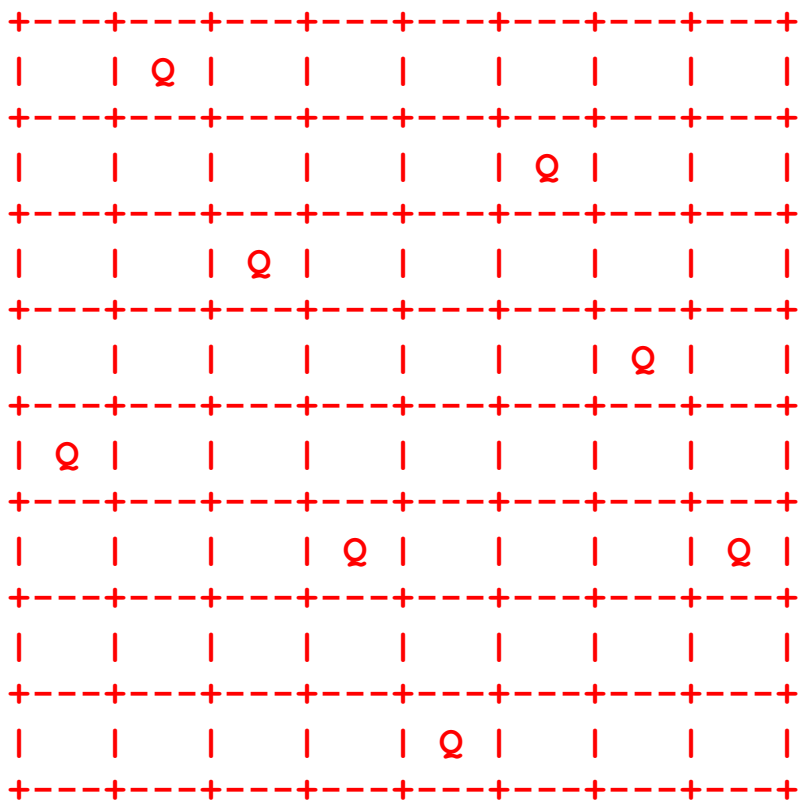
	Q						
					Q		
		Q					
						Q	
Q							
			Q				Q
				Q			

h = 2

A failed attempt to solve the 8-Queens Problem: Iteration 4

3	Q	3	3	6	3	4	2
3	4	3	5	4	Q	3	3
3	3	Q	3	5	4	5	3
3	6	2	4	4	5	Q	3
Q	4	4	2	5	5	4	4
3	6	4	Q	5	5	6	Q
2	2	3	3	3	3	4	2
2	4	3	3	Q	5	3	3

hmin = 2, imin = 0, jmin = 7



h = 2

Steepest Ascent (Greedy Local Search): Algorithm

Start from an initial state s (may be randomly chosen).

Repeat the following steps:

- If s is a goal, return s.
- Find all the states s_1, s_2, …, s_k that can be reached from s in a **single** step.
- Compute the objective function values f_i for all s_i.
- Let s_{i^*} denote a successor of s with the highest objective-function value (among the successors).
- If f_{i^*} is less than or equal to the objective-function value of s, return failure.
- Set $s = s_{i^*}$.

Properties of greedy local search

Usually very efficient in terms of time and memory requirement.

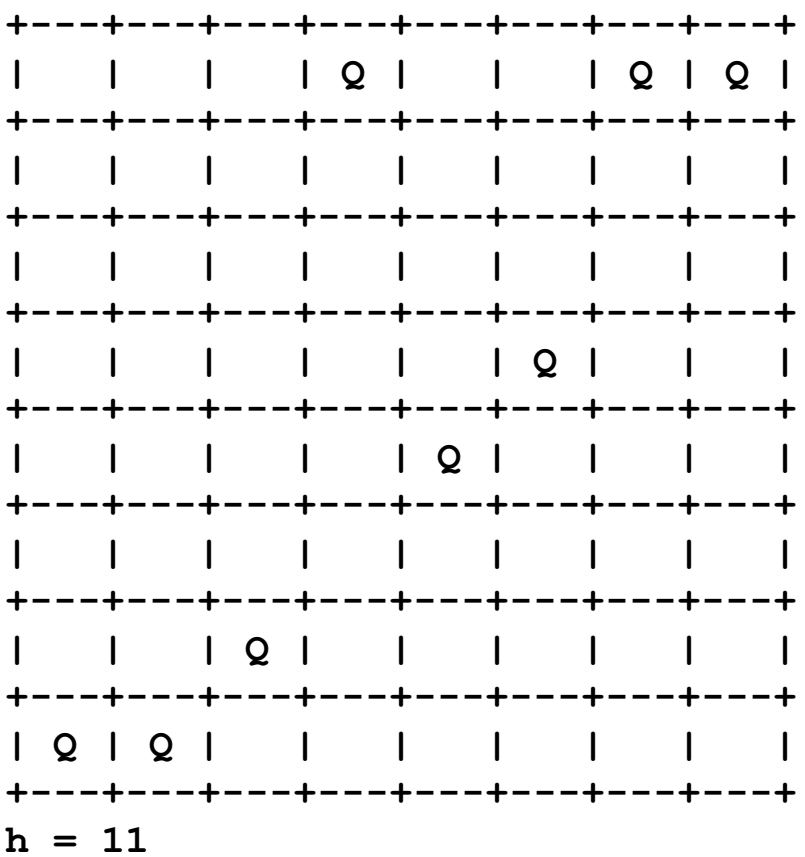
But failure is frequent because of

- Local maximum
- Plateaus and shoulders

Some ways to solve these problems

- **Sideways move:** Get out of shoulders (requires memory to prevent ending up in a cycle)
- **Stochastic hill climbing:** Choose an ascent (not always the steepest) with some probability.
- **Random restart:** Keep on trying with new randomly chosen initial states.
- **Local beam search:** Keep *k* states. Generate all the neighbors of all these *k* states. Keep the best *k* among all these neighbors for the next iteration. Not same as random restart. May lead to concentration in the same hole(s).

Solving the 8-Queens Problem with sideways moves: Initial board



Solving the 8-Queens Problem with sideways moves: Iteration 1

13	10	12	Q	11	11	Q	Q
9	8	9	10	9	9	11	13
9	8	9	9	10	11	13	8
11	8	9	12	11	Q	11	9
11	8	10	12	Q	9	11	10
9	9	10	13	8	9	9	9
12	10	Q	10	9	9	11	9
Q	Q	10	12	10	10	11	11

hmin = 8, imin = 1, jmin = 1

			Q			Q	Q
	Q						
					Q		
				Q			
		Q					
Q							

h = 8

Solving the 8-Queens Problem with sideways moves: Iteration 2

12	10	11	Q	8	9	Q	Q
8	Q	8	8	7	8	9	10
8	8	8	6	7	9	9	5
9	8	7	10	8	Q	8	6
9	8	8	9	Q	7	8	7
7	9	8	9	5	8	6	6
9	10	Q	7	6	7	9	6
Q	11	7	8	6	7	7	8

hmin = 5, imin = 2, jmin = 7

			Q			Q	
	Q						
							Q
					Q		
				Q			
		Q					
Q							

h = 5

Solving the 8-Queens Problem with sideways moves: Iteration 3

9	6	7	Q	4	6	Q	8
6	Q	5	6	4	5	7	10
7	6	6	5	5	6	8	Q
7	5	4	8	4	Q	7	6
7	5	5	6	Q	5	6	7
5	6	4	7	3	5	4	6
7	6	Q	6	3	4	7	6
Q	8	5	6	3	4	5	8

hmin = 3, imin = 5, jmin = 4

			Q			Q	
	Q						
							Q
					Q		
				Q			
		Q					
Q							

h = 3

Solving the 8-Queens Problem with sideways moves: Iteration 4

6	5	6	Q	4	5	Q	5
5	Q	4	4	4	4	5	6
5	6	4	3	5	5	5	Q
5	4	4	5	4	Q	6	3
4	3	3	4	5	4	3	3
4	6	4	5	Q	4	3	4
5	5	Q	5	3	4	4	3
Q	6	5	4	3	3	4	4

hmin = 3, imin = 2, jmin = 3

			Q			Q	
	Q						
							Q
					Q		
				Q			
		Q					
Q							

h = 3 [Stuck at a shoulder]

Solving the 8-Queens Problem with sideways moves: Iteration 4

6	5	6	Q	4	5	Q	5
5	Q	4	4	4	4	5	6
5	6	4	3	5	5	5	Q
5	4	4	5	4	Q	6	3
4	3	3	4	5	4	3	3
4	6	4	5	Q	4	3	4
5	5	Q	5	3	4	4	3
Q	6	5	4	3	3	4	4

hmin = 3, imin = 2, jmin = 3

						Q	
	Q						
			Q				Q
					Q		
				Q			
		Q					
Q							

h = 3 [Trying a sideways move]

Solving the 8-Queens Problem with sideways moves: Iteration 5

5	5	5	3	3	5	Q	3
5	Q	4	4	4	4	6	5
6	6	5	Q	6	5	7	Q
4	4	5	5	5	Q	6	2
4	4	3	4	5	5	4	1
5	6	4	5	Q	4	5	3
5	5	Q	5	3	4	5	3
Q	6	5	4	3	3	5	3

hmin = 1, imin = 4, jmin = 7

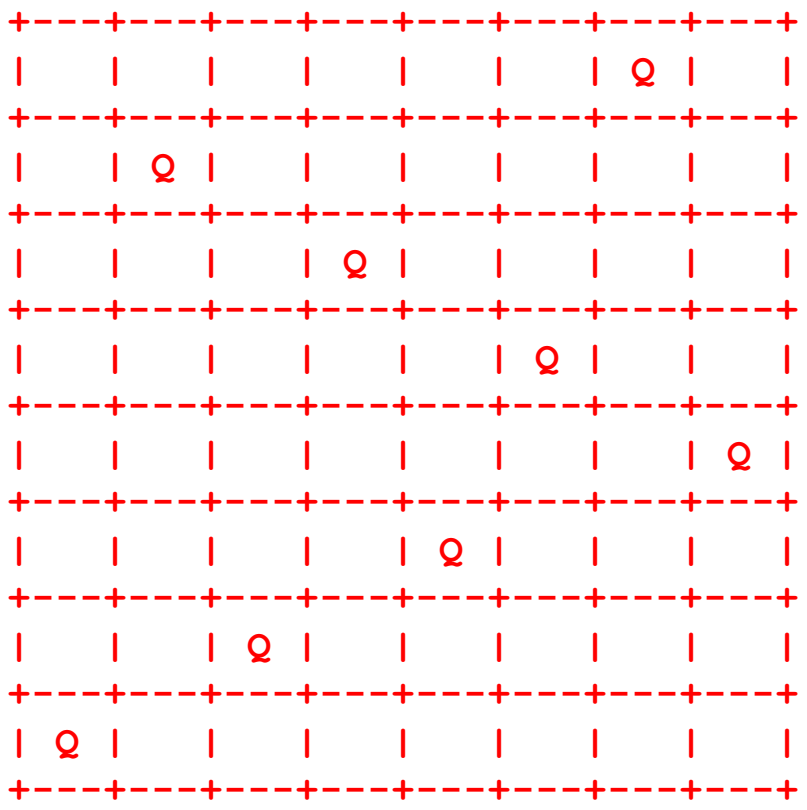
						Q	
	Q						
			Q				
					Q		
							Q
				Q			
		Q					
Q							

h = 1

Solving the 8-Queens Problem with sideways moves: Iteration 6

3	3	3	3	2	2	Q	3
3	Q	2	3	4	2	3	5
3	3	2	Q	4	3	4	3
2	2	3	4	4	Q	4	2
3	3	2	4	5	3	3	Q
3	4	2	4	Q	2	4	3
3	3	Q	3	2	3	3	3
Q	4	2	3	3	1	3	3

hmin = 1, imin = 7, jmin = 5



h = 1 [Stuck again at a shoulder]

Solving the 8-Queens Problem with sideways moves: Iteration 6

3	3	3	3	2	2	Q	3
3	Q	2	3	4	2	3	5
3	3	2	Q	4	3	4	3
2	2	3	4	4	Q	4	2
3	3	2	4	5	3	3	Q
3	4	2	4	Q	2	4	3
3	3	Q	3	2	3	3	3
Q	4	2	3	3	1	3	3

hmin = 1, imin = 7, jmin = 5

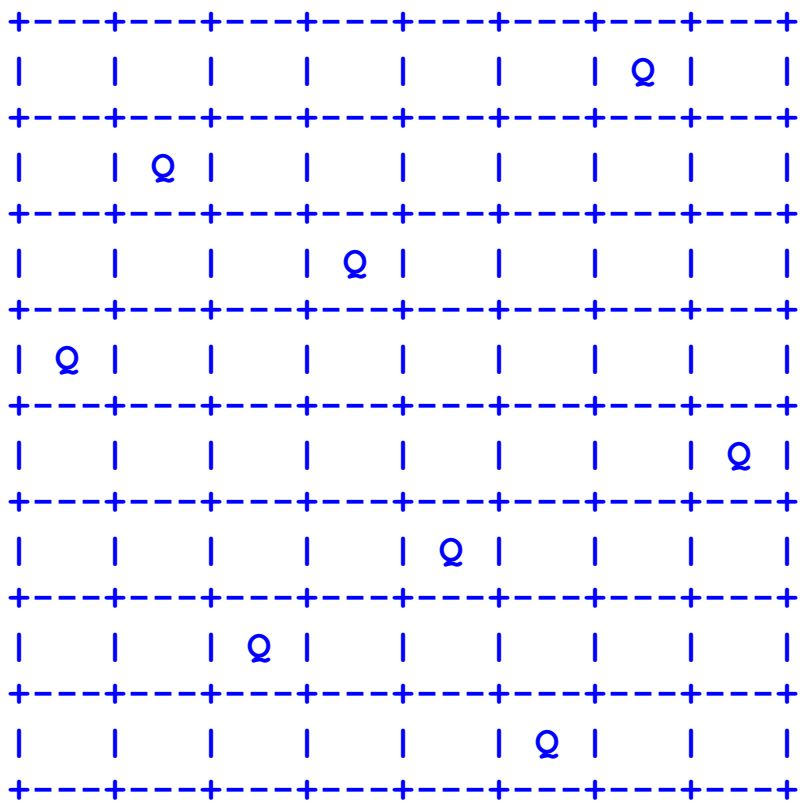
						Q	
	Q						
			Q				
							Q
				Q			
		Q					
Q					Q		

h = 1 [Sideways move again]

Solving the 8-Queens Problem with sideways moves: Iteration 7

2	3	3	3	2	2	Q	3
2	Q	3	2	4	2	3	4
3	3	3	Q	3	3	3	3
0	2	3	3	3	1	3	1
2	3	4	4	4	3	2	Q
2	4	3	4	Q	2	4	3
2	3	Q	3	3	3	4	3
Q	4	4	4	4	Q	4	4

hmin = 0, imin = 3, jmin = 0



h = 0

Simulated Annealing (SA)

Sideways moves can let the greedy search out of a shoulder.

But for the greedy algorithm, there is no way out of a local maximum / minimum.

In order to solve this problem, we need to make move(s) in the **wrong** direction.

Sometimes, we need to go to neighbors after degrading the objective-function value.

But such movements in the reverse direction must be carefully controlled.

One way is to use an analog of **annealing** in metallurgy.

> Heat a substance to high temperature.
> At high temperatures, the molecules are in high-energy states.
> As the substance is gradually cooled down, the molecules settle down in deep holes (stable low-energy states).

SA Algorithm for minimizing f ( )

Start with an initial state s (may be randomly chosen) and an initial temperature Θ.

Repeat until some termination condition is satisfied:

Choose a random neighbor s' of s.
Let $\Delta f = f(s') - f(s)$.
If $\Delta f < 0$ set $s = s'$, /* always accept good moves */
else do the following: /* accept a bad move according to the Boltzmann distribution */

Generate a real-valued random number r in the range [0, 1].

If $r \leqslant \exp\left[-\Delta f / \Theta\right]$, then set $s = s'$.

Reduce the temperature Θ.

Return s.

Note: The terminating condition may be: Θ is 0 or very small, or s is a goal node, or timeout, or …

SA gives statistical guarantees

If the cooling process is sufficiently slow, then SA will output the global optimum with probability $\longrightarrow 1$.

This is often impractical (too many iterations).

Under a feasible cooling schedule, we should be happy with a good enough suboptimal solution.

The Traveling Salesperson Problem (TSP)

There are n cities 0, 1, 2, …, n – 1 with a (symmetric) distance $d(i, j)$.

The state space consists of all n-tuples of the form 0, c_1, c_2, …, c_{n-1} , where c_1, c_2, …, c_{n-1} is a permutation of 1, 2, 3, …, n – 1.

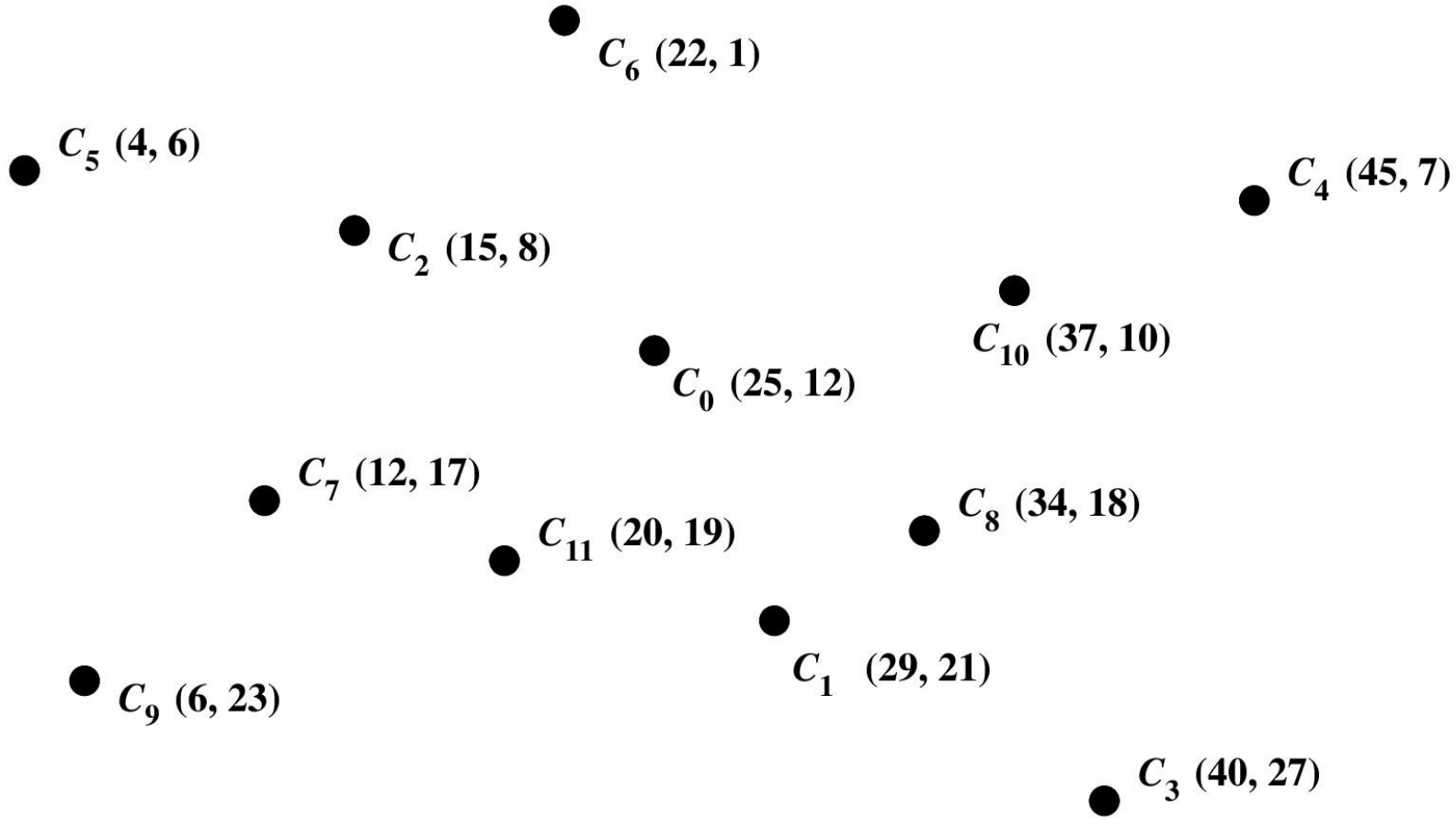
The size of the state space is $(n – 1)!$.

For n = 100, this size is $9.332\ldots \times 10^{155}$.

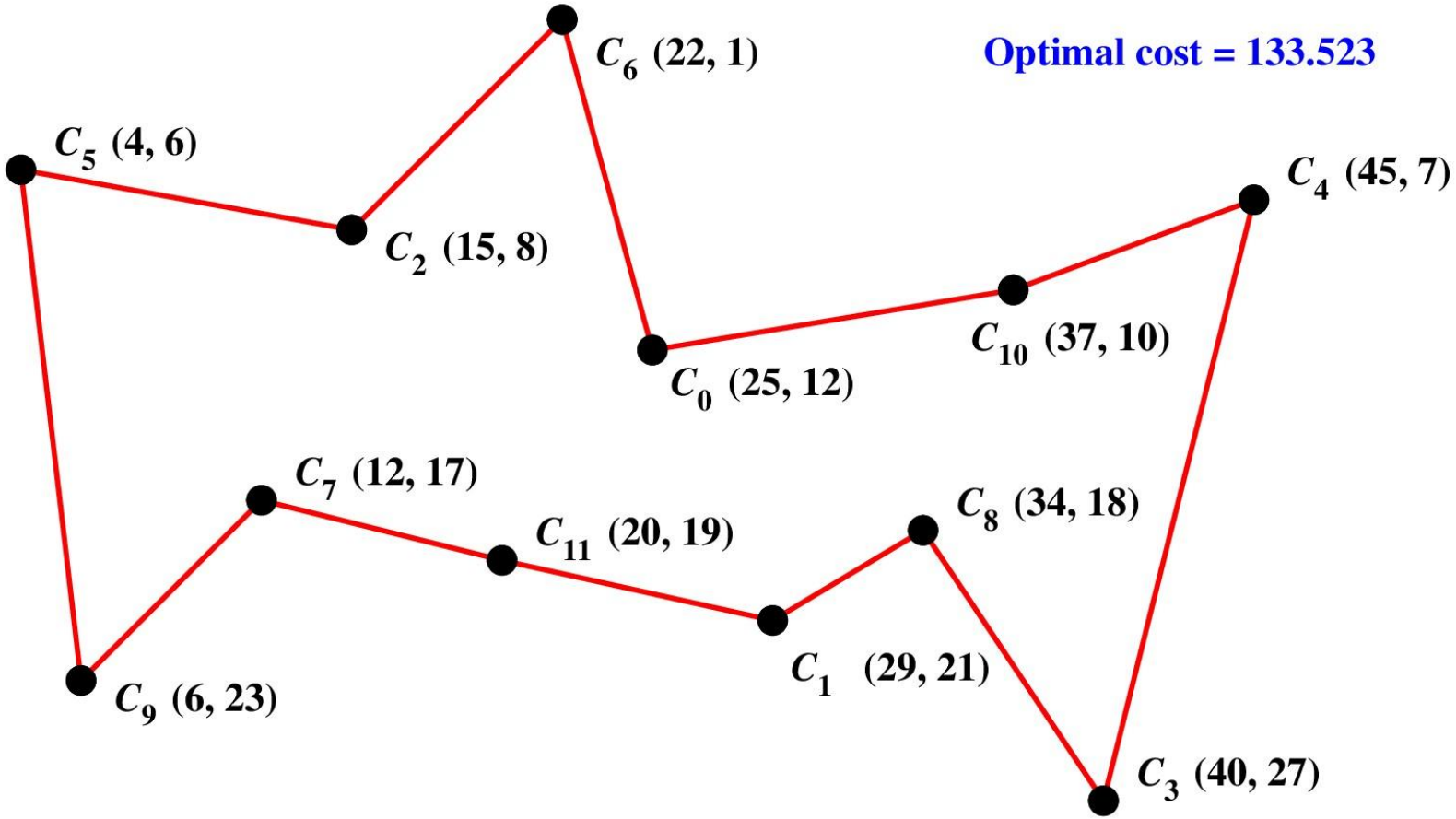
An exhaustive search for the optimal TSP tour is infeasible except for small values of n.

We work on a small sample (n = 12).

TSP: Example


C_6 (22, 1)
C_5 (4, 6)
C_4 (45, 7)
C_2 (15, 8)
C_{10} (37, 10)
C_0 (25, 12)
C_7 (12, 17)
C_8 (34, 18)
C_{11} (20, 19)
C_1 (29, 21)
C_9 (6, 23)
C_3 (40, 27)

TSP: Optimal solution of the example



Greedy Local Search: Formulation

Start with a random tour $T = (c_0, c_1, c_2, \ldots, c_{n-1})$.

For $i = 0, 1, 2, \ldots, n - 1$, generate the tour T_i by swapping c_i with c_{i+1} in T (i+1 is computed modulo n).

Compute bestcost = $\min_i \text{cost}(T_i)$.

If bestcost $\leqslant$ cost(T), replace T by the best T_i, and repeat.

If bestcost > cost(T), return T as the (suboptimal) TSP tour.

Sample runs

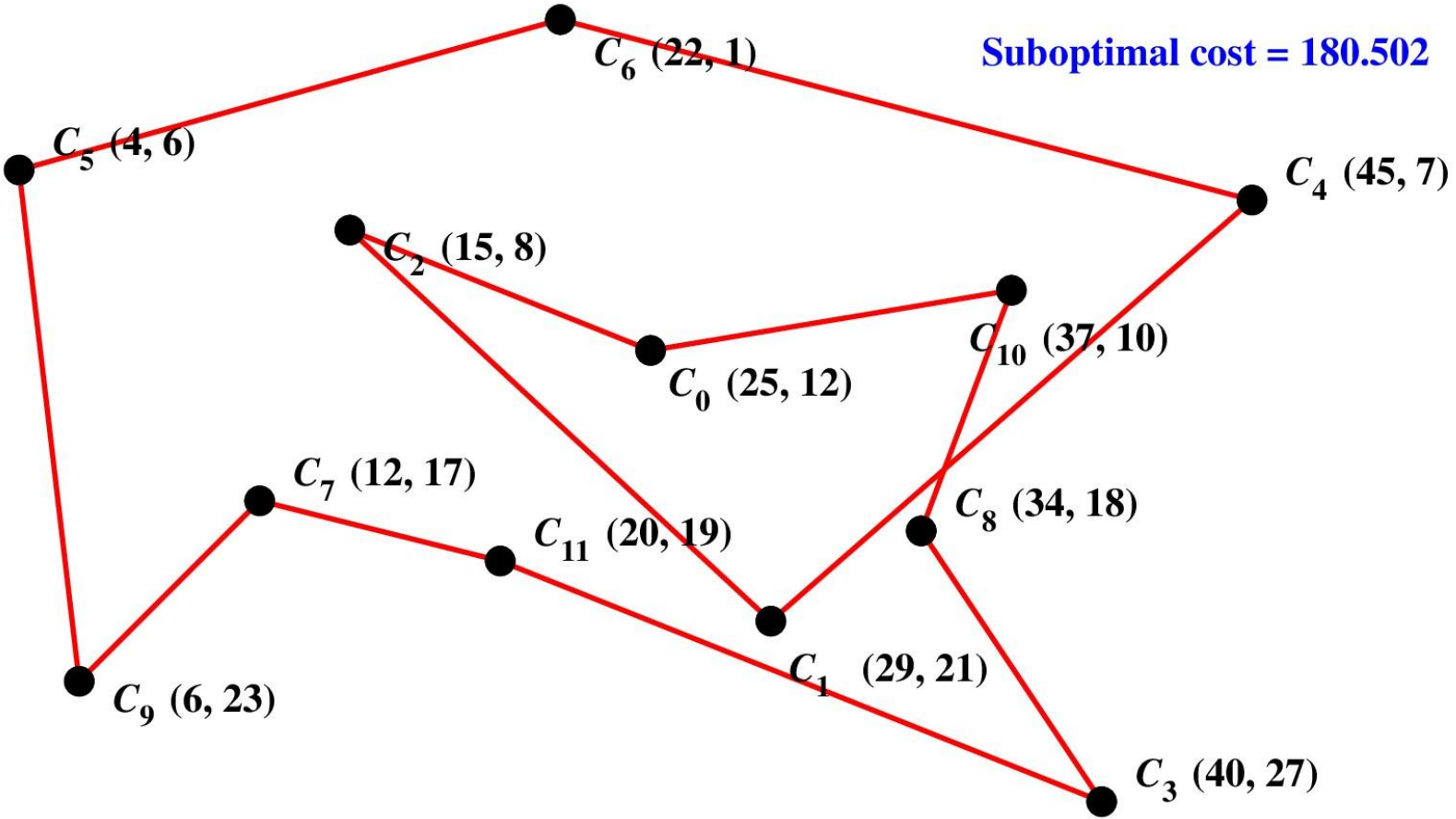
Run 1

```
[Iteration No  1] Current tour: 3 10 9 7 11 6 1 5 4 0 2 8 :: Cost = 240.750407
[Iteration No  2] Current tour: 3 10 9 7 11 6 5 1 4 0 2 8 :: Cost = 218.490427
[Iteration No  3] Current tour: 3 10 9 7 11 5 6 1 4 0 2 8 :: Cost = 213.030045
[Iteration No  4] Current tour: 3 10 9 7 11 5 6 4 1 0 2 8 :: Cost = 204.843484
[Iteration No  5] Current tour: 10 3 9 7 11 5 6 4 1 0 2 8 :: Cost = 203.189846
[Iteration No  6] Current tour: 8 3 9 7 11 5 6 4 1 0 2 10 :: Cost = 197.363635
[Iteration No  7] Current tour: 8 3 9 11 7 5 6 4 1 0 2 10 :: Cost = 196.424516
[Iteration No  8] Current tour: 8 3 11 9 7 5 6 4 1 0 2 10 :: Cost = 183.969760
[Iteration No  9] Current tour: 8 3 11 7 9 5 6 4 1 0 2 10 :: Cost = 181.171524
[Iteration No 10] Current tour: 8 3 11 7 9 5 6 4 1 2 0 10 :: Cost = 180.502442
```

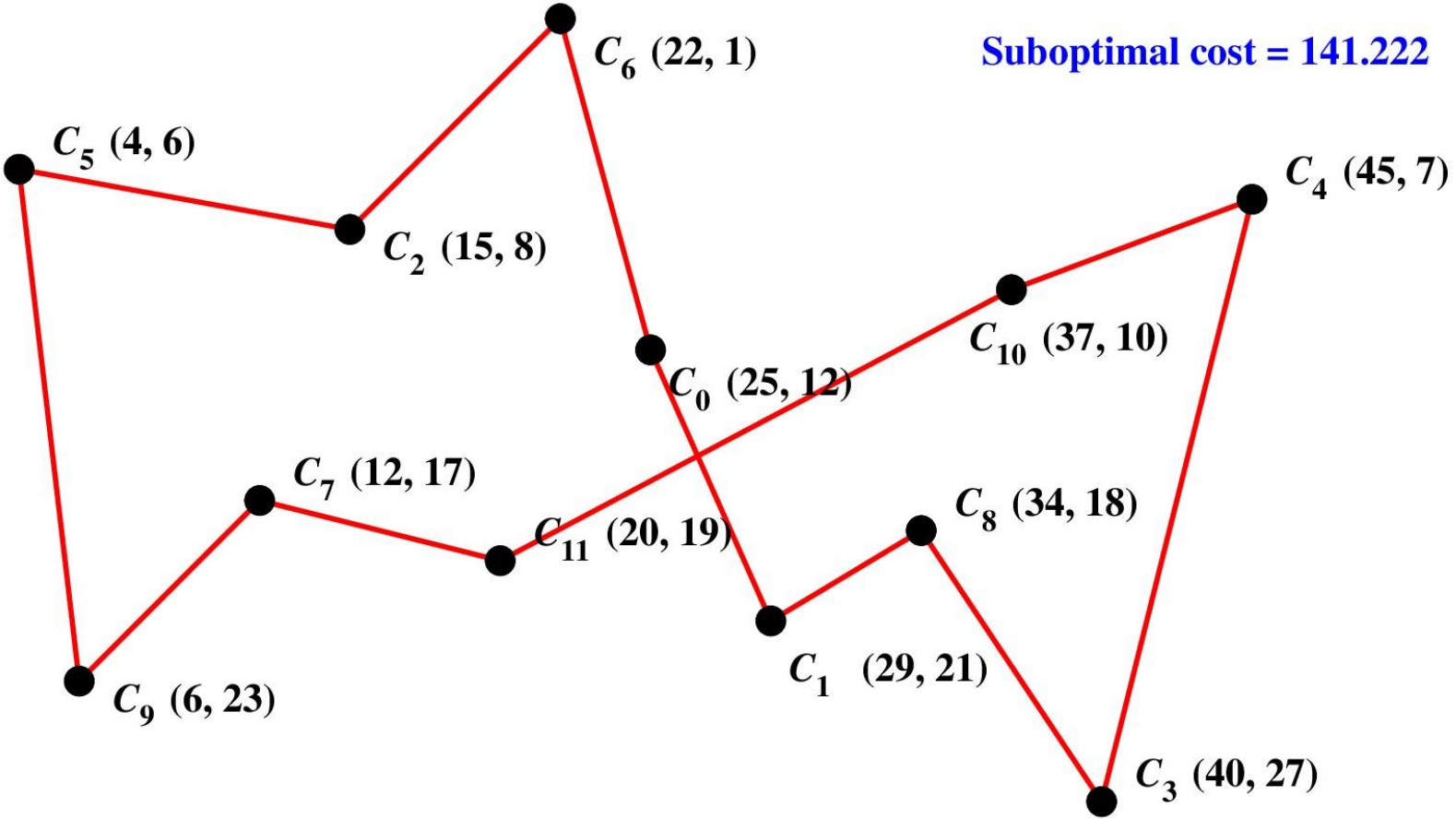
Run 2

```
[Iteration No  1] Current tour: 2 5 6 0 7 8 3 1 4 9 10 11 :: Cost = 228.910030
[Iteration No  2] Current tour: 2 5 6 0 7 8 3 1 4 10 9 11 :: Cost = 190.624391
[Iteration No  3] Current tour: 2 5 6 0 7 8 3 1 4 10 11 9 :: Cost = 181.654112
[Iteration No  4] Current tour: 5 2 6 0 7 8 3 1 4 10 11 9 :: Cost = 172.496453
[Iteration No  5] Current tour: 5 2 6 0 7 8 1 3 4 10 11 9 :: Cost = 166.865987
[Iteration No  6] Current tour: 5 2 6 0 7 1 8 3 4 10 11 9 :: Cost = 160.594211
[Iteration No  7] Current tour: 5 2 6 7 0 1 8 3 4 10 11 9 :: Cost = 160.445027
[Iteration No  8] Current tour: 5 2 7 6 0 1 8 3 4 10 11 9 :: Cost = 157.505731
[Iteration No  9] Current tour: 5 7 2 6 0 1 8 3 4 10 11 9 :: Cost = 150.958395
[Iteration No 10] Current tour: 7 5 2 6 0 1 8 3 4 10 11 9 :: Cost = 144.019940
[Iteration No 11] Current tour: 9 5 2 6 0 1 8 3 4 10 11 7 :: Cost = 141.221704
```

Output of Run 1



Output of Run 2



Simulated Annealing: Formulation

Start with a random tour $T = (c_0, c_1, c_2, \ldots, c_{n-1})$ and an initial temperature Θ.

While $\Theta > 0$, repeat:

- Randomly choose an i in the range 0 to $n - 1$.
- Obtain the tour T' by swapping c_i with c_{i+1} in T.
- Let $\Delta c = \text{cost}(T') - \text{cost}(T)$.
- If $\Delta c < 0$, set $T = T'$,
- else
  - Generate a real number r uniformly randomly in the interval [0, 1].
  - If $r \leqslant \exp\left[-\Delta c / \Theta\right]$, then set $T = T'$.
- Decrement Θ by $\Delta\Theta$.

In our example, we take the initial temperature $\Theta = 10$, and $\Delta\Theta = 0.001$ (cool it slowly).

Simulated Annealing: Experimental results

The average performance of SA is far better than that of greedy local search.

Most frequently, we get the optimal solution

0 10 4 3 8 1 11 7 9 5 2 6

of cost **133.523**.

Two other nearly optimal solutions produced by SA are

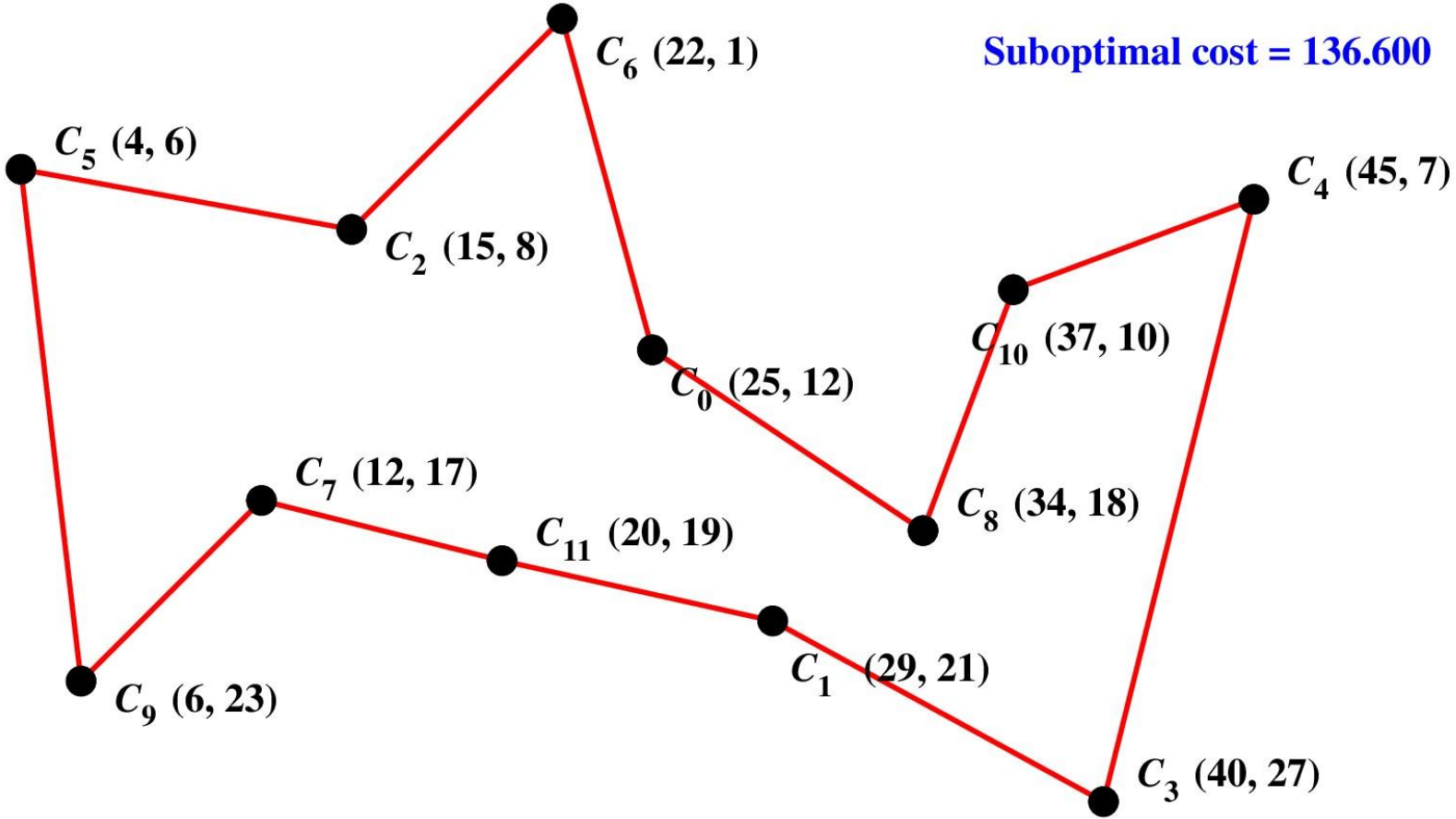
0 8 10 4 3 1 11 7 9 5 2 6

of cost **136.600**, and

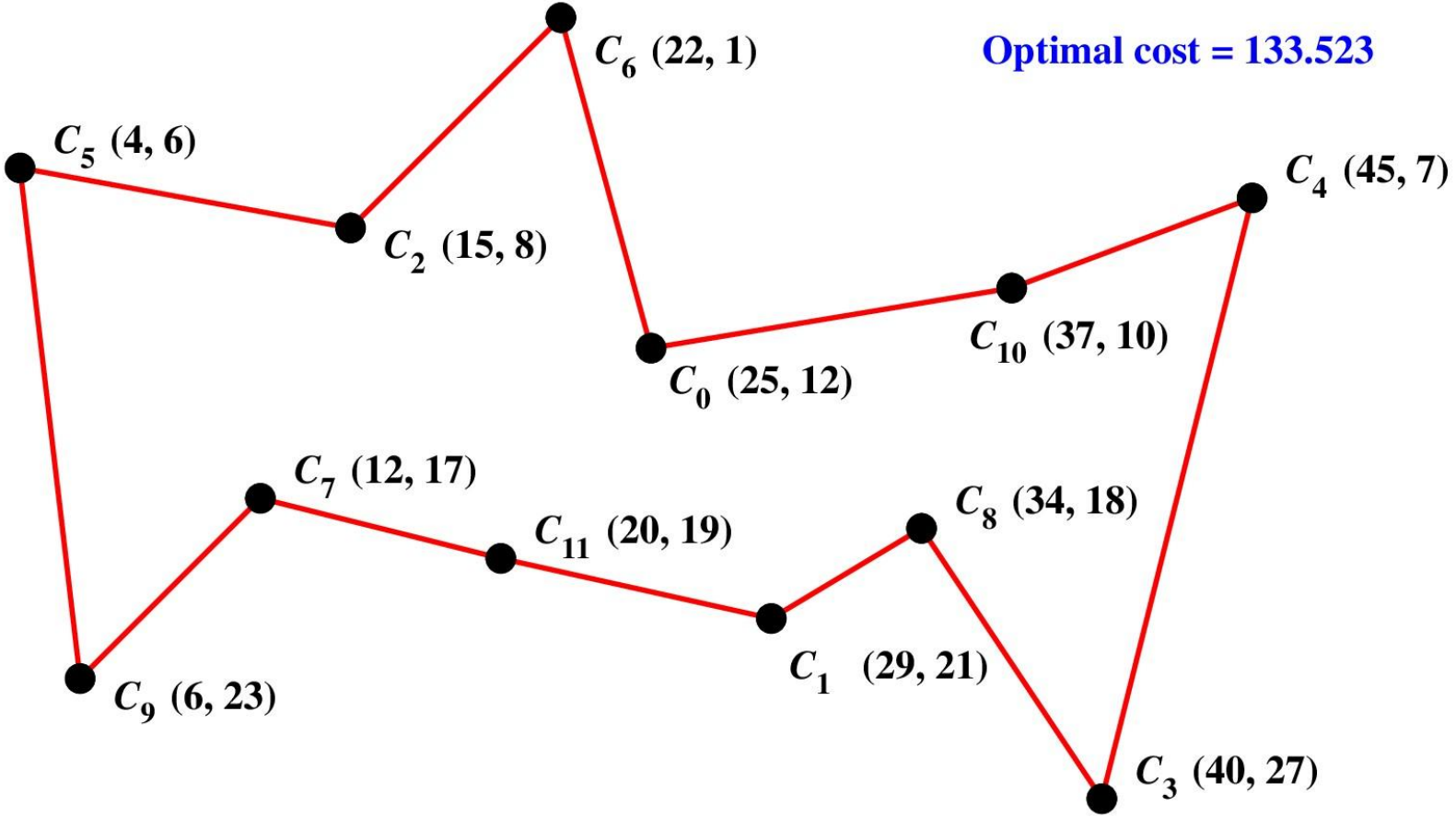
0 11 7 9 5 2 6 10 4 3 8 1

of cost **136.680**.

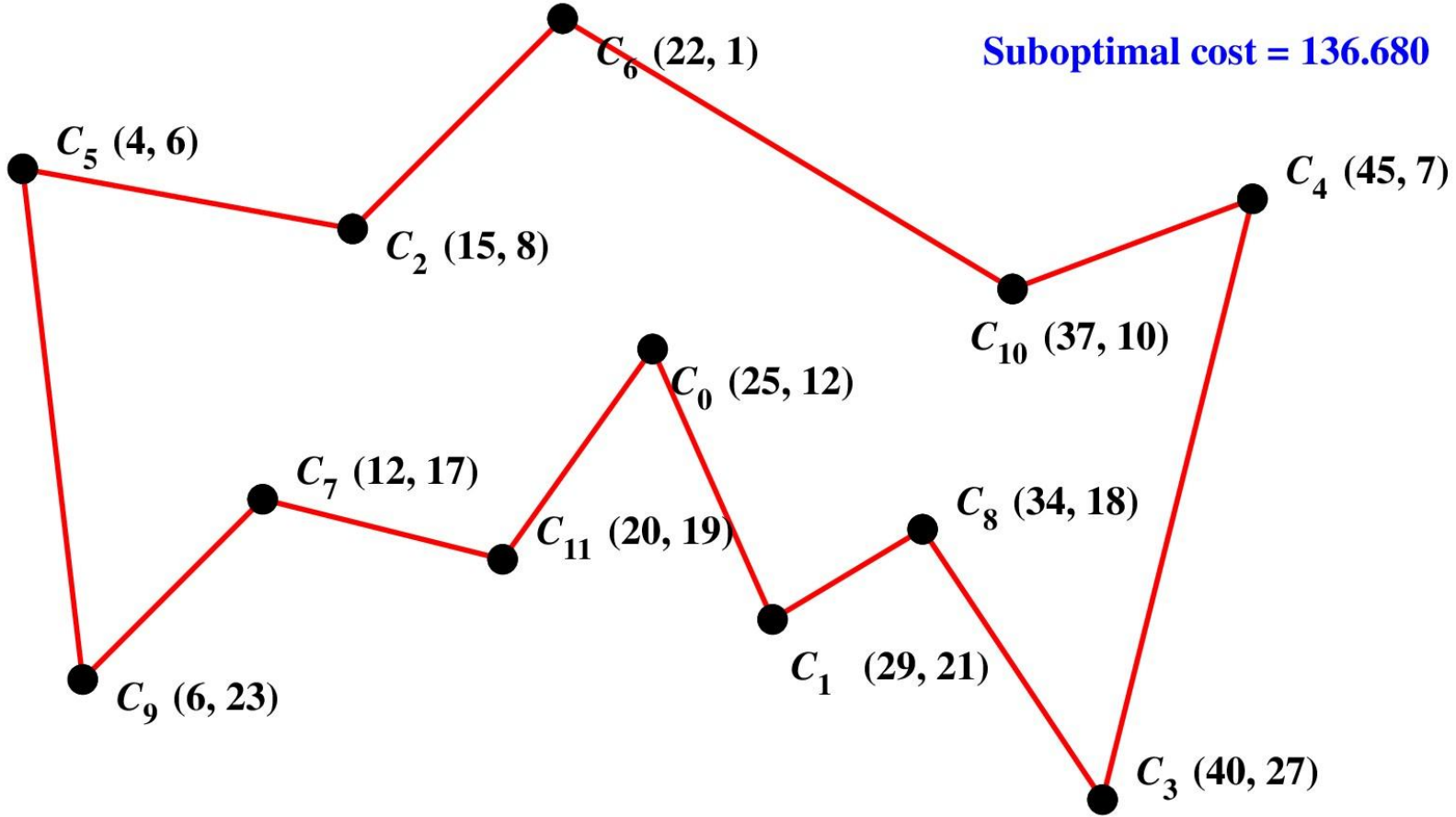
Second best solution from SA



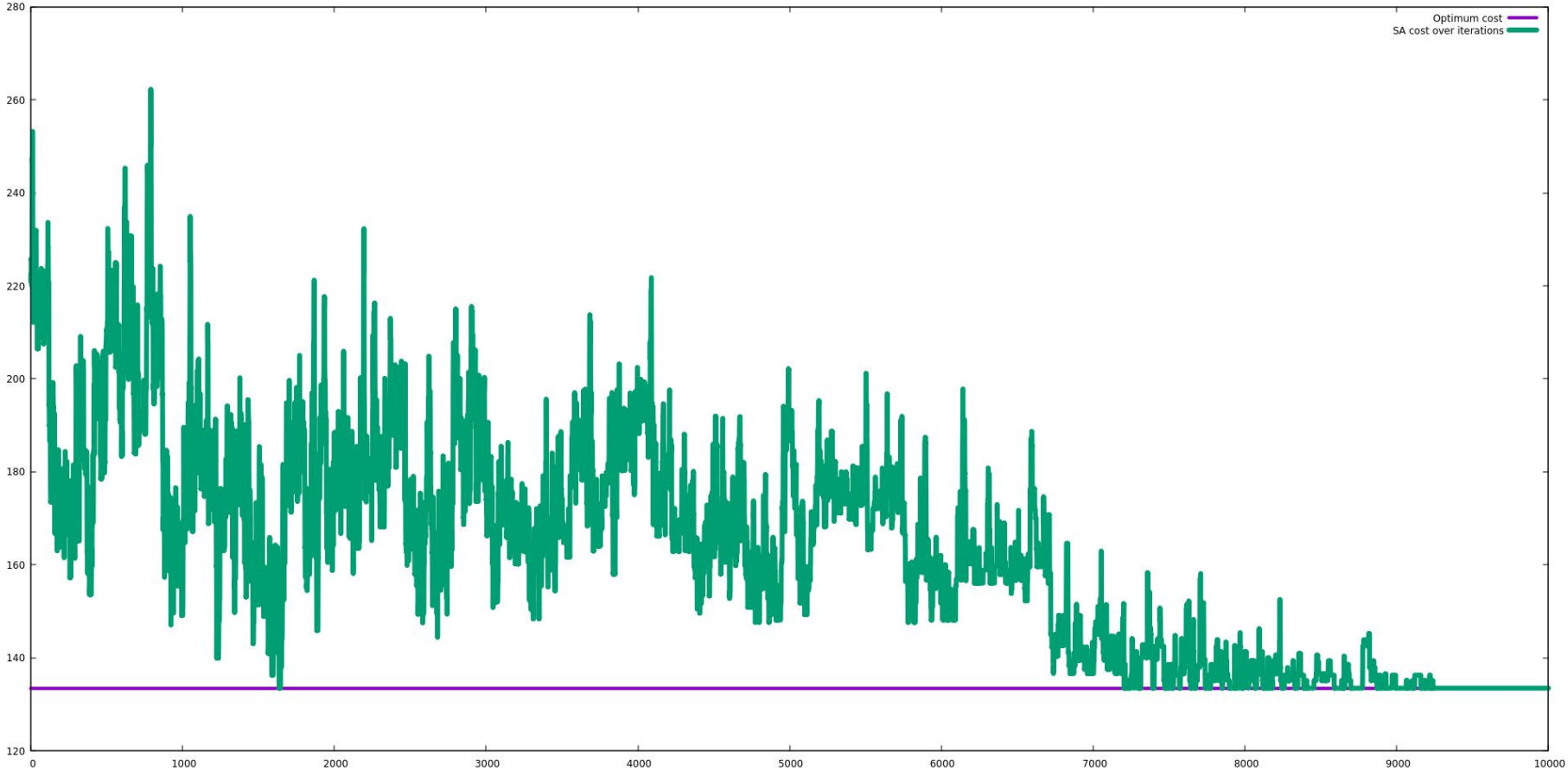
Optimal solution



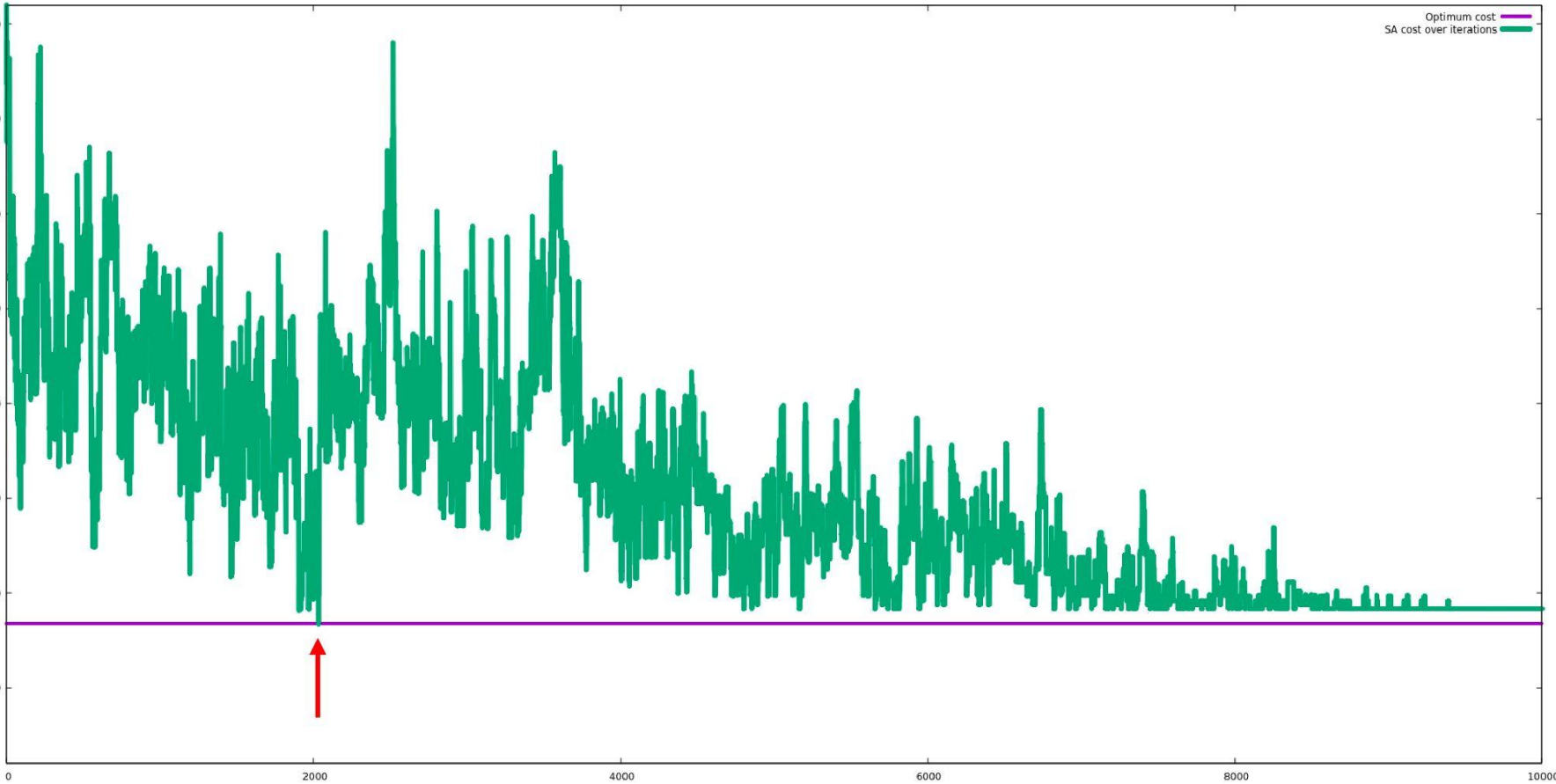
Third best solution from SA



Evolution of the optimal solution by SA



Evolution of a suboptimal solution by SA



Genetic Algorithms (GA): Inspiration

SA draws inspiration from a process of physics.

GA draws inspiration from a process of biology (**natural selection**).

The DNA is a sequence of four building blocks (chemicals) called A (Adenine), C (Cytosine), T (Thymine), and G (Guanine).

During mating, the DNA of two parents combine to form a single DNA of the offspring. Some part of the offspring's DNA comes one parent, the other part from the other parent. **[Crossover]**

Rarely, random changes occur during the combination of the two parents' DNA. **[Mutation]**

Fitness of a creature is dictated by its DNA. The better the fitness score is, the better the chance of survival and mating. **[Reproduction]**

Mutation is usually destructive but, if successful, is capable of generating new species.

Genetic Algorithms: Overview

[Formulation] Encode each state as a string (of a fixed length) over any alphabet.

[Initialization] Start with an initial population of some states (may be randomly chosen from the entire state space).

[Reproduction] Decide which pairs in the current population will mate. This decision may be random or based upon the fitness of the individuals.

[Crossover] The mating of two strings yield two offsprings, each carrying a part of the parents' DNA.

[Mutation] With a low frequency, make random disturbance in the offspring DNA (offspring strings).

[Next generation] The offsprings constitute the population for the next iteration.

Repeat until a solution is found, or a good enough solution is found, or the process takes too long.

Genetic Algorithms: Generic Description

Set **population** to an initial set of strings.

Repeat until a terminating condition is reached:

    Set **nextgeneration** to an empty list.

    Repeat a certain number of times:

        Choose parent1 and parent2 (must be different) from **population**.
        Parent1 and parent2 mate to give child1 and child2.
        Apply mutation to each of child1 and child2 with a small probability.
        Add child1 and child2 to **nextgeneration**.

    Set **population** := **nextgeneration**.

Return a best member from **population**.

Formulation: Examples

n-Queens problem

A string $r_1, r_2, \ldots, r_n$ of integers in the range 1, 2, …, n (or 0, 1, 2, …, n – 1).

r_i denotes the position of the i-th queen in the i-th column.

TSP

A permutation of 0, 1, 2, …, n – 1 (or 1, 2, 3, …, n).

We may force the permutation to start with a fixed city (like 0).

Reproduction: Who will mate?

Let the current population be $P = \{ p_1, p_2, \ldots, p_t \}$.

To every p_i, we assign a **fitness** value $f(p_i) = f_i$.

Individuals with high fitness are expected to produce offsprings of high fitness.

The larger the fitness f_i, the more likely is p_i to be chosen as a parent before each crossover.

Let $f = \sum_i f_i$.

For each mating, p_i is chosen as a parent with probability f_i / f.

Some p_i may be chosen multiple times in each round (generation).

Some p_i may never be chosen in a round.

Exercise: Write a function to choose a parent according to the above distribution.

Fitness examples

n-Queens problem

We wanted to minimize $h(s)$ = the number of attacking pairs in state s. Smaller $h()$ values are better.
We can take:

- $f(s)$ = the number of non-attacking pairs in state $s = C(n,2) - h(s)$.
- $f(s) = 1 / h(s)$.

TSP

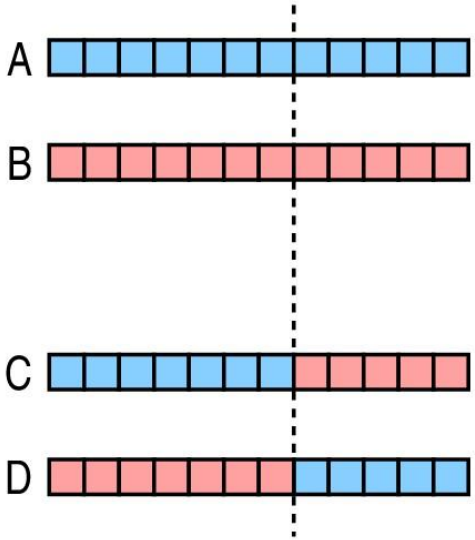
We wanted to minimize $h(T)$ = cost of the tour T. Smaller $h()$ values are better.
We can take:

- $f(T) = B - h(T)$, where B is an upper bound on the tour cost.
- $f(T) = 1 / h(T)$.

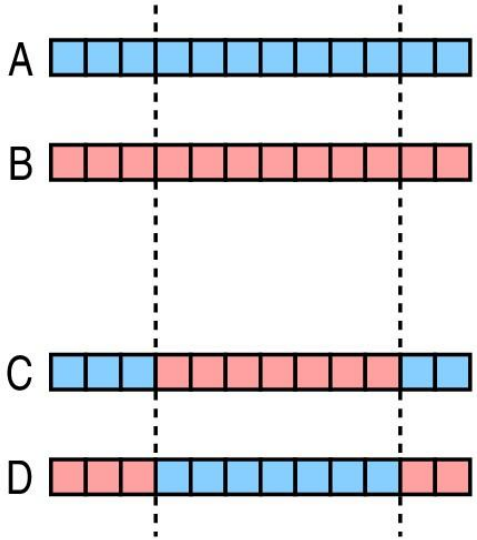
Crossover

Let A and B be the two parents.

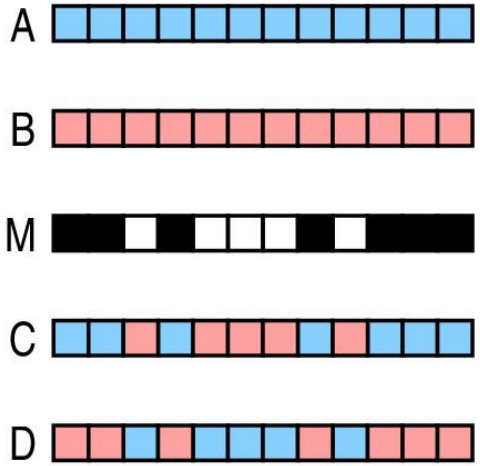
Crossover of A and B gives two offsprings C and D.



One–point crossover



Two–point crossover



Masked crossover

Crossover example: n-Queens problem

Let P and Q be the two regions chosen for crossover.

Consider the attack graph. Each of A and B has three types of edges: attacks within P, attacks within Q, and the cut edges between P and Q.

C keeps the within-P edges of A and the within-Q edges of B, and replaces the original cut edges of A by new cut edges.

Likewise, for D.

If A and B are fit, the within-P and within-Q edges are few.

Crossover has the potential of reducing the count(s) of the cut edges.

All the three crossover strategies work equally well (or equally badly).

Crossover example: TSP

Crossover, in general, produces strings that are not Hamiltonian tours at all.

Example: Crossover of

A = 5 10 11 4 | 9 6 2 1 8 3 | 0 7

B = 1 8 5 6 | 3 10 11 2 9 7 | 0 4

give

C = 5 10 11 4 3 10 11 2 9 7 0 7

D = 1 8 5 6 9 6 2 1 8 3 0 4

Neither offspring is a Hamiltonian tour.

We need to redefine the crossover operator for TSP in order to repair the highlighted positions.

TSP: Partially matched crossover (PMX)

In each child, keep the crossover part from the mate. Replace the repeated entries in the other part by own DNA entries. The crossover region supplies the necessary injective mapping for that. Apply it one or more times.

A = 5 10 11 4 | 9 6 2 1 8 3 | 0 7

B = 1 8 5 6 | 3 10 11 2 9 7 | 0 4

C = 5 10 11 4 3 10 11 2 9 7 0 7

D = 1 8 5 6 9 6 2 1 8 3 0 4

Repair 10: 10 → 6

C = 5 6 11 4 3 10 11 2 9 7 0 7

Repair 11: 11 → 2 2 → 1

C = 5 6 1 4 3 10 11 2 9 7 0 7

Repair 7: 7 → 3 3 → 9 9 → 8

C = 5 6 1 4 3 10 11 2 9 7 0 8

Repair 1: 1 → 2 2 → 11

D = 11 8 5 6 9 6 2 1 8 3 0 4

Repair 8: 8 → 9 9 → 3 3 → 7

D = 11 7 5 6 9 6 2 1 8 3 0 4

Repair 6: 6 → 10

D = 11 7 5 10 9 6 2 1 8 3 0 4

Properties: Positional substitution does not take care of distances.

TSP: Order crossover (OX)

In each child, keep the mate's DNA. After the crossover, follow the original path and remove all repetitions to fill the remaining part.

Initial C	5	10	11	4	3	10	11	2	9	7	0	7
Reduced A	0	7	5	10	11	4	9	6	2	1	8	3
Repaired C	4	6	1	8	3	10	11	2	9	7	0	5
Initial D	1	8	5	6	9	6	2	1	8	3	0	4
Reduced B	0	4	1	8	5	6	3	10	11	2	9	7
Repaired D	5	10	11	7	9	6	2	1	8	3	0	4

Properties: Assume that A and B are good tours (fit).

- The crossover part is good (it is a contiguous subtour).
- The remaining part is a subtour of a good tour and is usually good (like under triangle inequality).
- The jumps at the two crossover points may be bad.

TSP: Cycle crossover (CX)

C takes the first city from A. But then the first city of B cannot be taken from B, and should be taken from A. The corresponding city in B must again be taken from A. Repeat until a cycle completes. Take the remaining cities from B. D is obtained by complementing C's choices.

A	5	10	11	4	9	6	2	1	8	3	0	7
B	1	8	5	6	3	10	11	2	9	7	0	4
C	5	8	11	6	3	10	2	1	9	7	0	4
D	1	10	5	4	9	6	11	2	8	3	0	7

Properties

- Each city comes from one of the parents.
- Long jumps may be introduced.
- This strategy may fail to produce new offsprings.
  Exercise: Give an example with $C = A$ and $D = B$, and another with $C = B$ and $D = A$.

Mutation

In biology, mutation happens rather infrequently. More often than not, the effects of mutation on offsprings are disastrous. At the same time, mutation combined with natural selection can create new species. For that to happen, you need to run the process for billions of years.

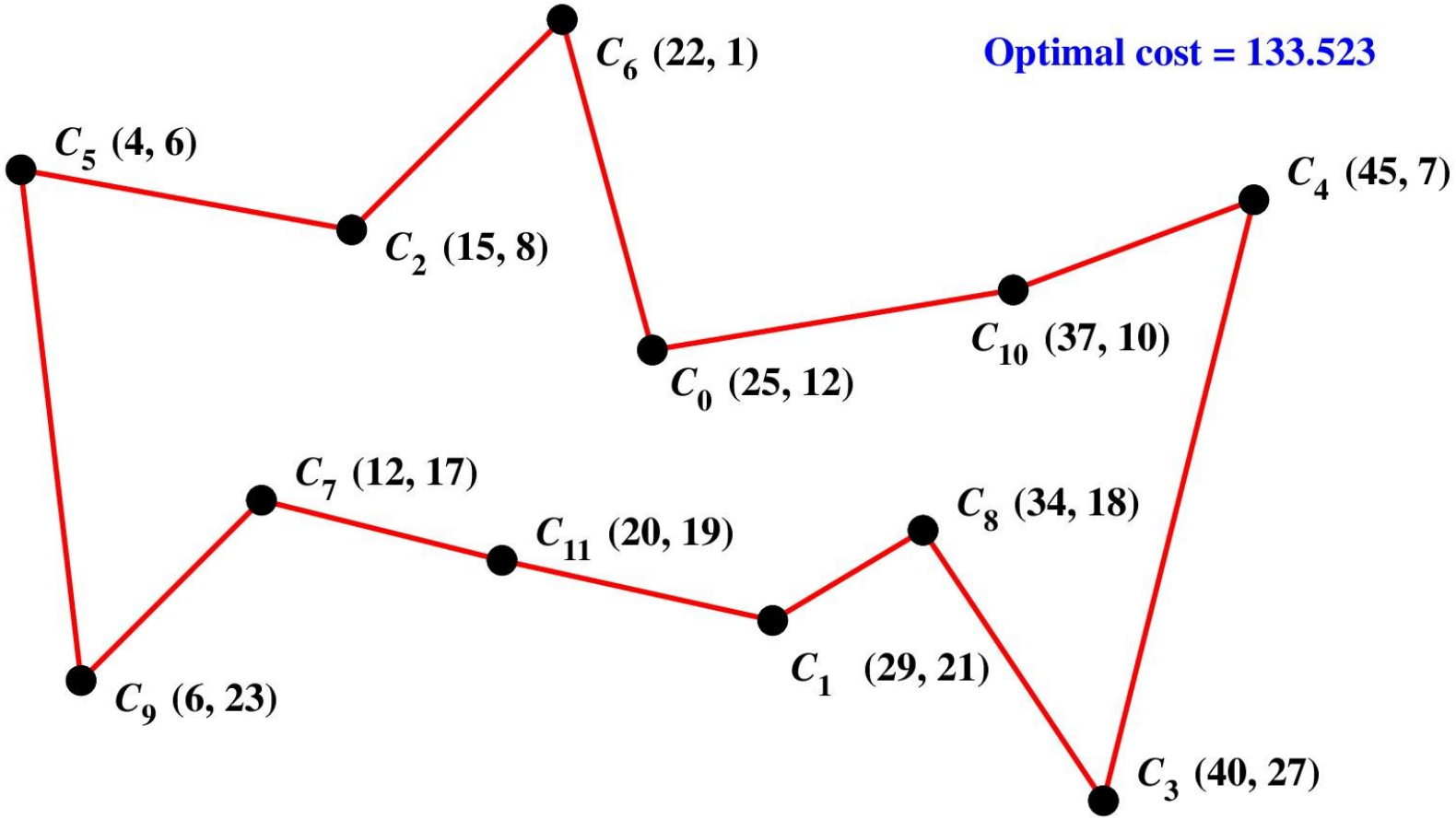
In GA, we may expect similar outcomes.

Examples

n-Queens problem: Randomly choose a column, and change the position of the queen in that column to another randomly chosen column.

TSP: Pick two random positions in the tour, and swap the two cities in those positions.

Example of solving TSP by GA



TSP by GA: With artificially small parameters

P = current population = $(p_0, p_1, p_2, \ldots, p_{N-1})$, where N = population size = 10.

$f(p_i)$ = fitness of the i-th member of the population = $1 / (\text{cost}(p_i))^3$.

Q = population in the next generation = $(q_0, q_1, q_2, \ldots, q_{N-1})$.

$N / 2 = 5$ matings are made in each iteration.

The m-th mating gives the offsprings q_{2m} and q_{2m+1}.

Each mating is repeated if (at least) one of the offsprings are already generated.

The crossover positions are given by a pair (s, t). Cities in the indices s, $s+1$, $s+2$, …, $t-1$ are exchanged.

The remaining places are filled by the OX strategy.

A and B are indices (zero-based) in the P array.

Mutation is not used.

TSP by GA: One iteration

```
+++ Current population
    Member   0:  0  2 10  9  3  1 11  4  7  6  8  5 :: cost = 278.501180
    Member   1:  5  4 10  9  2 11  7  8  6  0  3  1 :: cost = 238.124831
    Member   2:  3  2 11  7  9  1 10  0  6  8  4  5 :: cost = 239.525245
    Member   3: 10  5  7 11  1  9  4  3  2  8  6  0 :: cost = 247.413044
    Member   4:  9  1 10  0  4  3  6 11  5  2  7  8 :: cost = 231.566733
    Member   5:  2  8  6  1  7  9  3 11  5  0 10  4 :: cost = 238.375907
    Member   6:  0  4  9  7  3 11  2  5  1  6 10  8 :: cost = 232.989363
    Member   7: 11  3  2  7 10  4  8  0  1  6  5  9 :: cost = 204.704128 [BEST]
    Member   8:  7  5  2  4  6 10 11  9  8  3  1  0 :: cost = 205.423450
    Member   9: 11  4  7  1 10  5  3  2  8  6  9  0 :: cost = 299.637285
```

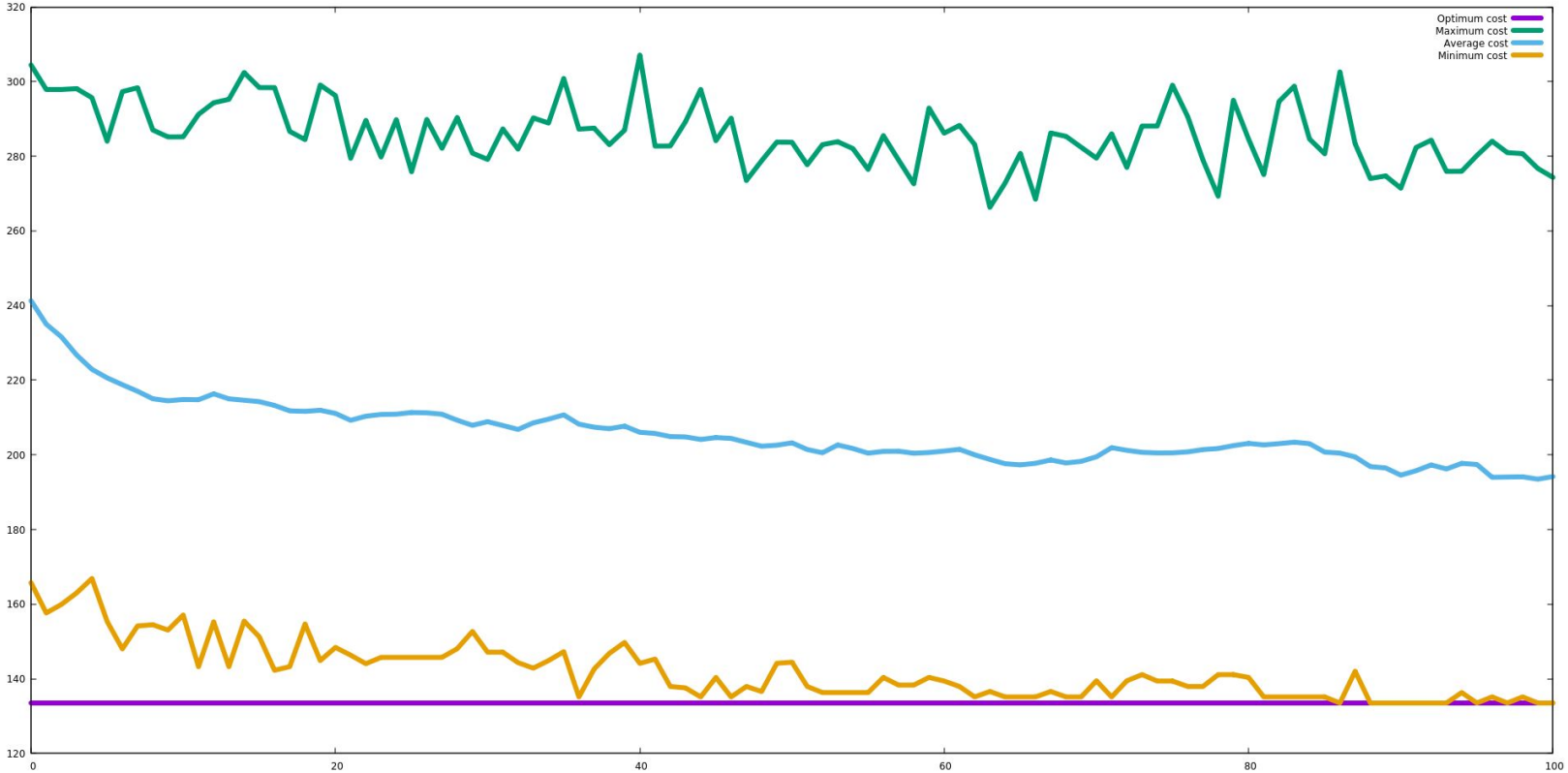
TSP by GA: One iteration

```
+++ Mating schedule
   A = 3, B = 7, s = 4, t = 8
   A = 1, B = 8, s = 1, t = 7
   A = 8, B = 7, s = 5, t = 10
   A = 7, B = 2, s = 3, t = 11
   A = 3, B = 6, s = 2, t = 7

+++ Next generation
    Member   0: 11  1  9  3 10  4  8  0  2  6  5  7 :: cost = 179.919554 [BEST]
    Member   1:  7 10  8  0  1  9  4  3  6  5 11  2 :: cost = 233.517549
    Member   2:  7  5  2  4  6 10 11  8  0  3  1  9 :: cost = 205.464005
    Member   3:  6  4 10  9  2 11  7  8  3  1  0  5 :: cost = 199.491381
    Member   4:  2 10 11  9  3  4  8  0  1  6  7  5 :: cost = 211.797593
    Member   5:  7  4  0  1  6 10 11  9  8  3  5  2 :: cost = 239.028429
    Member   6:  3  2  5  7  9  1 10  0  6  8  4 11 :: cost = 210.559781
    Member   7:  2 11  9  7 10  4  8  0  1  6  5  3 :: cost = 218.805052
    Member   8:  1  4  9  7  3 11  2  8  6  0 10  5 :: cost = 263.499035
    Member   9:  3  2  7 11  1  9  4  5  6 10  8  0 :: cost = 241.354948
```

TSP by GA with realistic parameters

Population size is 1000, and we make 100 iterations



Relative experimental performance of local search algorithms

Exhaustive search requires looking at 11! (about 40 million) tours.

Greedy Local Search was very fast. It terminated usually after 5 – 15 iterations.
GLS never gave the optimal solution.
The quality of the suboptimal solutions was poor.
The TSP landscape is filled with potholes.

Simulated Annealing generated 10,000 tours.
Its performance was good (optimal or near optimal) most of the times.
The cooling schedule may make SA prohibitively slow.

Genetic Algorithm generated a total of 1000 × 100 = 100,000 tours.
For small population sizes, GA leads to too much specialization.
Optimal and near optimal solutions are generated most of the times (with large population).
GA was the slowest in our (small) experiment.
GA has less solid theoretical foundations than SA.

What if there are 100 cities? Or 1000? Or more?

GA: Some remarks

- Is it good to return a best individual from the last population?

  You may keep track of the all-time-best.

- Match with reality?
  - Not all offsprings do survive to reach adulthood (and mate).
  - This is true not only in the wild (like lion/impala cubs) but also for human babies.
  - We can allow more than $N / 2$ (like N, $2N$, $3N$, …) mates in each generation.
  - Keep the best N among the offsprings, for the next iteration.
  - **Exercise:** Propose a way to do it with only an N-sized table for the next generation.

- Is it easy to formulate and mimic the natural process?
  - It ran for about 4 billion years.
  - Is that time the only issue?

GA: Concluding remarks

More on GA's:

> David E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

GA's have been proved to be effective in

- Circuit layout designing
- Job-shop scheduling
- Evolving the architecture of deep neural networks

"*It is not clear how much of the appeal of genetic algorithms arises from their superiority on specific tasks, and how much from the appealing metaphor of evolution.*"

— AIMA