Local search algorithms

Chapter 4, Sections 3–4

Outline

- ♢ Hill-climbing
- ♢ Simulated annealing
- ♢ Genetic algorithms (briefly)

Iterative improvement algorithms

In many optimization problems, **path** is irrelevant;
the goal state itself is the solution

Then state space = set of "complete" configurations;
find **optimal** configuration, e.g., TSP
or, find configuration satisfying constraints, e.g., timetable

In such cases, can use iterative improvement algorithms;
keep a single "current" state, try to improve it

Constant space, suitable for online as well as offline search

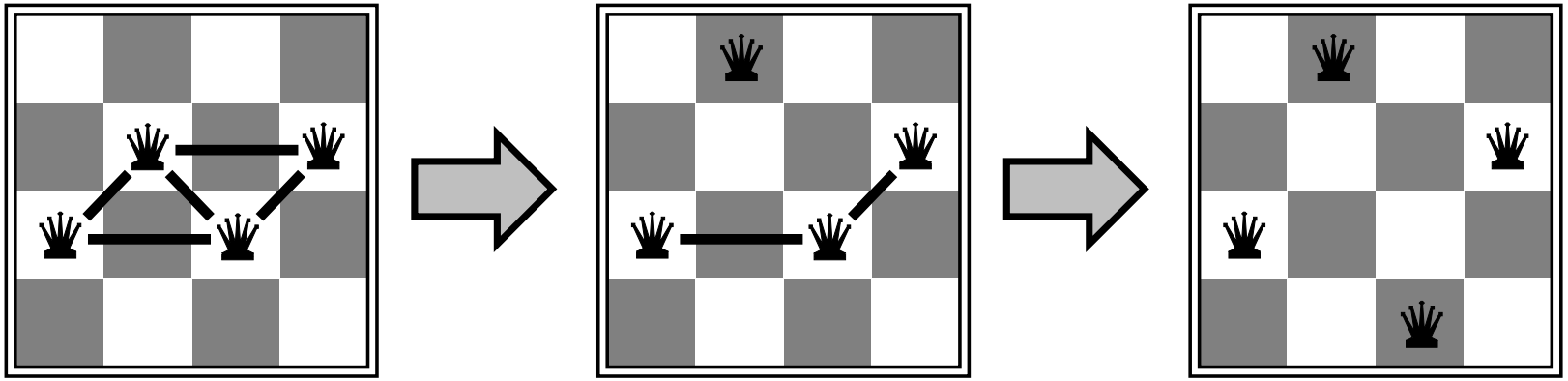
The search algorithms that we have seen so far are designed to explore search spaces systematically. This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. **When a goal is found, the path to that goal also constitutes a solution to the problem.**

In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem (see page 71), what matters is the final configuration of queens, not the order in which they are added. The same general property holds for many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts



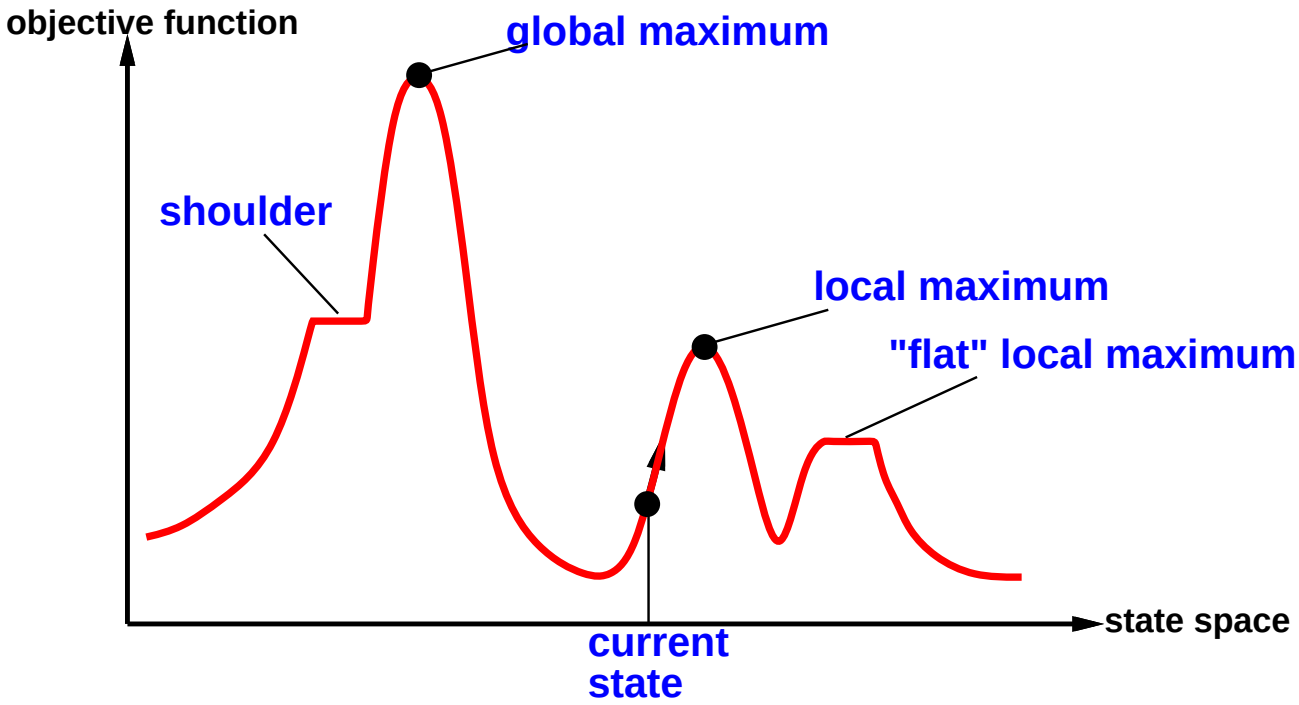
Hill-climbing (or gradient ascent/descent)

"Like climbing Everest in thick fog with amnesia"

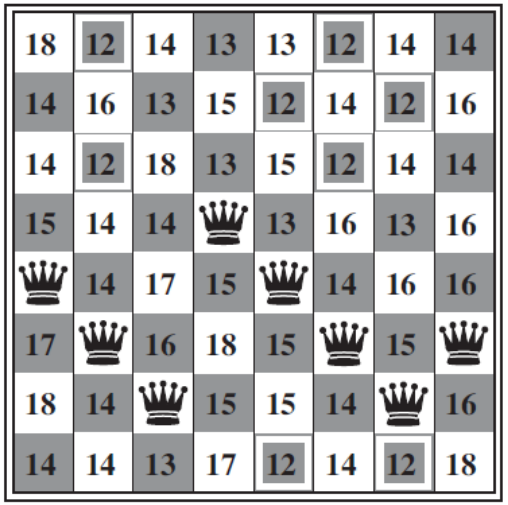
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Hill-climbing contd.

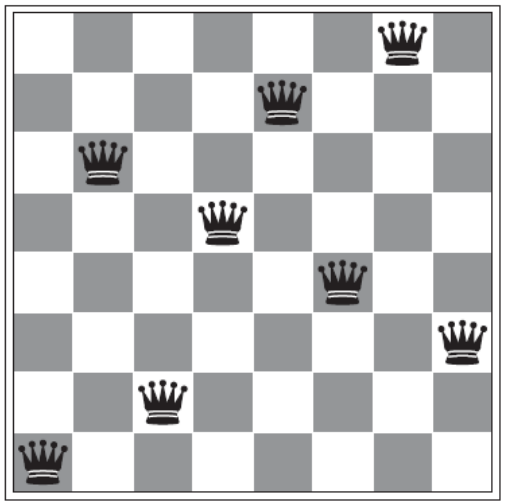
Useful to consider state space landscape



Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure 4.1 illustrates the problem schematically. More concretely, the state in Figure 4.3(b) is a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.



(a)



(b)

Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

To illustrate hill climbing, we will use the 8-queens problem. Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8×7=56 successors). The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions. Figure 4.3(a) shows a state with h=17. The figure also shows the values of all its successors, with the best successors having h=12. Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.

Ridges: a ridge is shown in Figure RIDGE 4.4. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

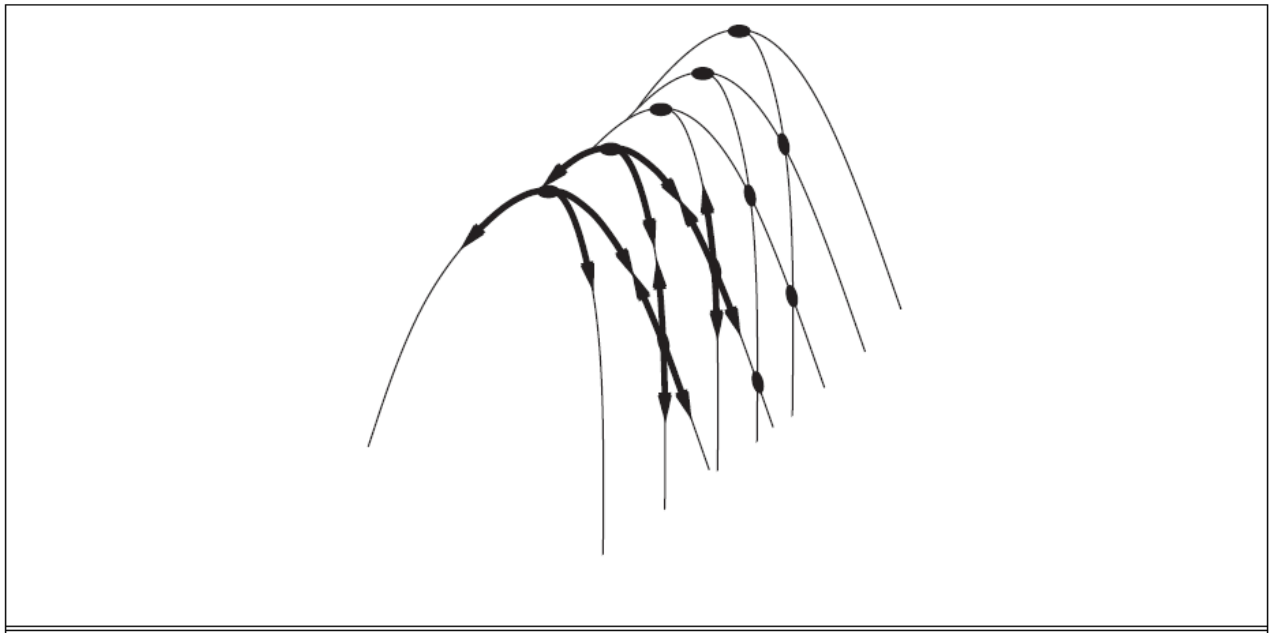


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Plateaux: A plateau is a flat area of the state-space landscape. It can be a flat local SHOULDER maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. (See Figure 4.1.) A hill-climbing search might get lost on the plateau.

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

In metallurgy, ANNEALING annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a lowenergy crystalline state.

To explain simulated annealing, we switch our point of view from GRADIENT DESCENT hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

The innermost loop of the simulated-annealing algorithm (previous slide) is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

Local beam search

```
function BEAM-SEARCH( problem, k) returns a solution state
    start with k randomly generated states
    loop
        generate all successors of all k states
        if any of them is a solution then return it
        else select the k best successors
```

Not the same as k parallel searches
Searches that find good states will recruit other searches to join them

Problem: often all k states end up on same local hill

Stochastic beam search:
choose k successors randomly, biased towards good ones

Close analogy to natural selection

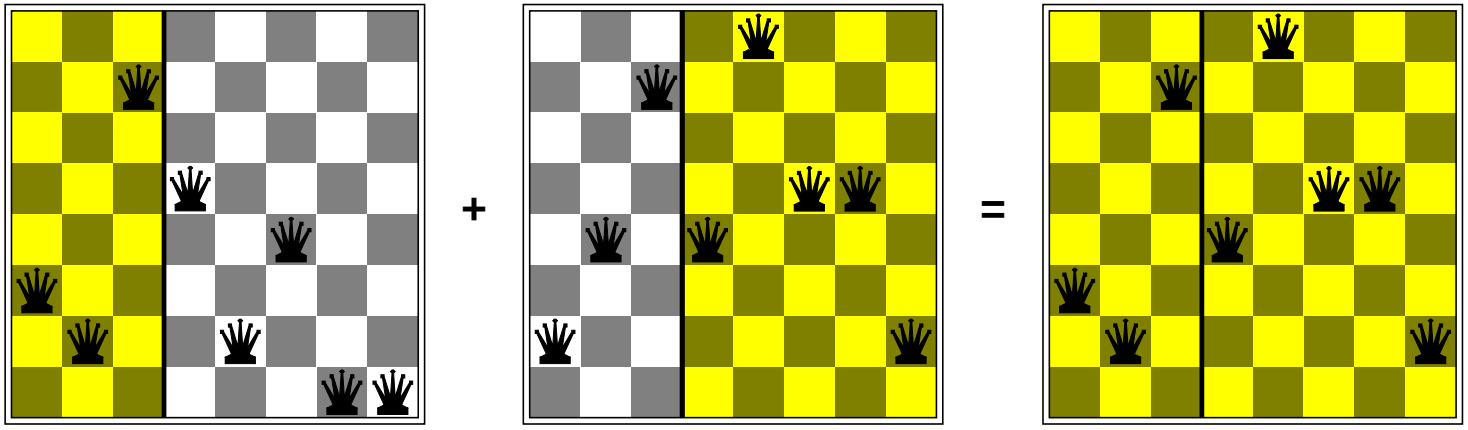
At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the parallel search threads. In effect, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

Genetic algorithms

Genetic algorithms
= stochastic local beam search + generate successors from **pairs** of states

Each state should be a string of characters;
Substrings should be meaningful components

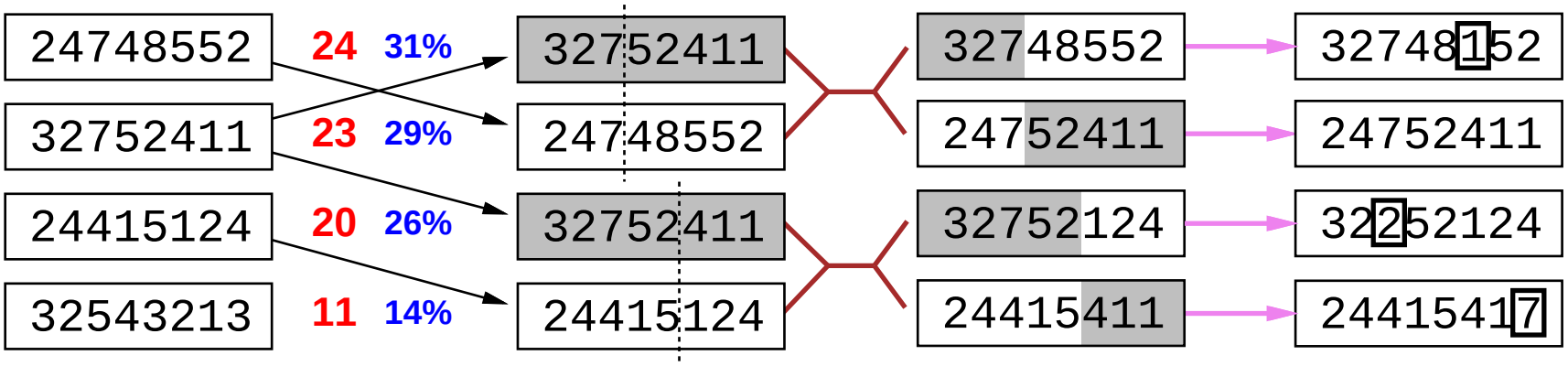
Example: n-queens problem
i'th character = row where i'th queen is located



672 47588 752 51447 672 51447

Genetic algorithms

Genetic algorithms
= stochastic local beam search + generate successors from **pairs** of states



Genetic algorithms $\neq$ biological evolution
for example, real genes encode replication machinery

Probably of the first one --> 24/(24+23+2-+11) * 100 --> 31%

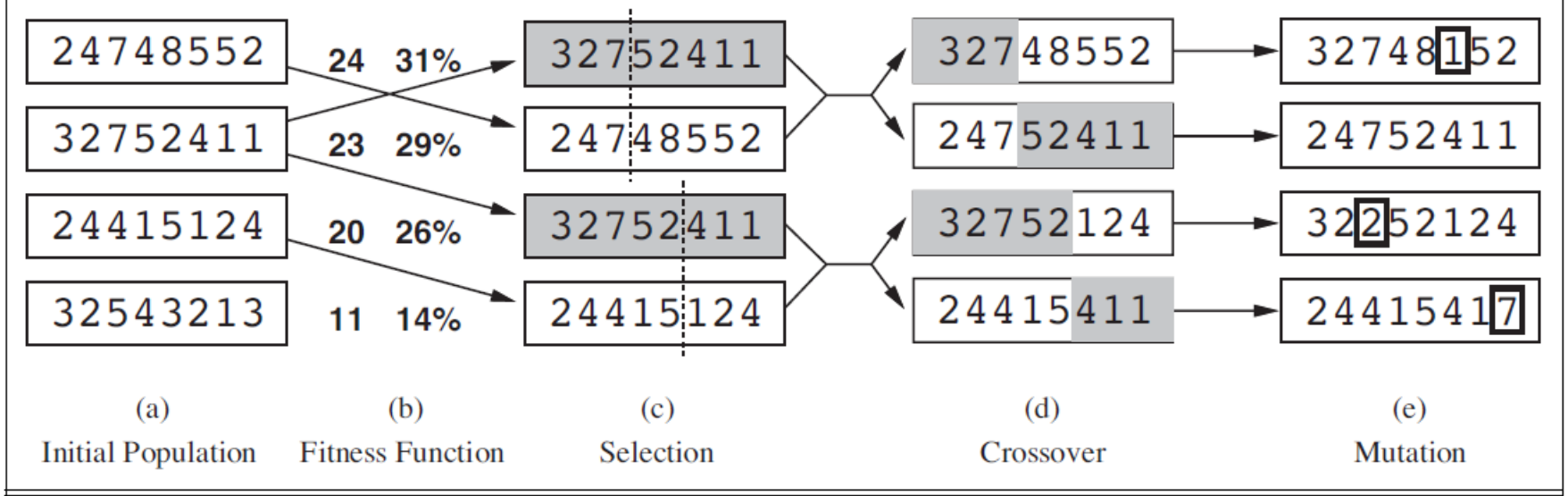


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8× log2 8=24 bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. Figure 4.6(a) shows a population of four 8-digit strings representing 8-queens states.

The production of the next generation of states is shown in Figure 4.6(b)–(e). In (b), each state is rated by the objective function, or (in GA terminology) the fitness function. A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of nonattacking pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. **In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.**

TSP and GA

- Heuristic optimisation methods for solving TSP
- GA representations and operators:
  - Adjacency representation
  - Ordinal representation
  - Path representation
  - Matrix representation
- Incorporating local search methods
- A fast GA for TSP ?

The TSP arises in numerous applications, and the size of the problem can be very large [239]:

> Circuit board drilling applications with up to 17,000 cities are mentioned in [287], X-ray crystallography instances with up to 14,000 cities are mentioned in [49], and instances arising in VLSI fabrication have been reported with as many as 1.2 million cities [266]. Moreover, 5 hours on a multi-million dollar computer for an optimal solution may not be cost-effective if one can get within a few percent in seconds on a PC. Thus there remains a need for heuristics.

Within the last few decades, many people have offered algorithms that generate approximate solutions to the TSP. These include nearest neighbor, greedy, nearest insertion, farthest insertion, double minimum spanning trees, strip, space-filling curves, and others offered by Karp, Litke, Christofides, and so forth [239].

GA for TSP

Approaches differ in the used representation and variation operators

Main problem: the GA is too slow to solve really big problems (10,000 cities)

TSP with 100 cities can be solved by global optimisation within hours

Comparisons are done on publicly available test problems with documented optimal or best-known solution (TSPLIB)

Ordinal representation

The ith element of the list is city j from the remaining cities, unvisited so far

position(C)	1	2	3	4	5	6	7	8	9
representation(l)	1	1	2	1	4	1	3	1	1
tour	1	2	4	3	8	5	9	6	7

$$j \in \{1, 2, \ldots, n - i + 1\}$$

The first number on the list l is 1, so take the first city from the list C(1,2,3,4,5,6,7,8) as the first city of the tour(city number 1). and remove it from C. The resulting partial tour is (1). The next number on the list l is also 1, so take the first city from the current C(2,3,4,5,6,7,8) list as the next city of the tour (city number 2), and remove it from C. The resulting partial toru is then (1-2). The next number on the list l is 2, so take second city from the current list C(3,4,5,6,7,8) as the next city of the tour(city number 4) and remove it from C. The new partial tour is (1-2-4)....

Classical one-point crossover generates legal tours!

The partial tour to the left of the crossover point remains unchanged

Crossover for ordinal representation

position	1	2	3	4	5	6	7	8	9
p_1 representation	1	1	2	1	4	1	3	1	1
p_2 representation	5	1	5	5	5	3	3	2	1
o_1 representation	1	1	2	1	5	3	3	2	1
o_2 representation	5	1	5	5	4	1	3	1	1
p_1 tour	1	2	4	3	8	5	9	6	7
p_2 tour	5	1	7	8	9	4	6	3	2
o_1 tour	1	2	4	3	9	7	8	6	5
o_2 tour	5	1	7	8	6	2	9	3	4

Path representation

The most natural representation:

position	1	2	3	4	5	6	7	8	9
representation	5	1	7	8	6	2	9	3	4
tour	5	1	7	8	6	2	9	3	4

Partially-mapped crossover (PMX)

$$
\begin{aligned}
p_1 &= (123|4567|89) \\
p_2 &= (452|1876|93) \\
o_1 &= (xxx|1876|xx) \\
o_2 &= (xxx|4567|xx)
\end{aligned}
$$

Mappings: $1 \leftrightarrow 4, 8 \leftrightarrow 5, 7 \leftrightarrow 6, 6 \leftrightarrow 7$ **Based on the middle part**

Fill in the no-conflict xs:

$$
\begin{aligned}
o_1 &= (x23|1876|x9) \\
o_2 &= (xx2|4567|93)
\end{aligned}
$$

Use the mappings for the remaining xs:

$$
\begin{aligned}
o_1 &= (423|1876|59) \\
o_2 &= (182|4567|93)
\end{aligned}
$$

PMX. This crossover builds an offspring by choosing a subsequence of a tour from one parent and preserving the order and position of as many cities as possible from the other parent [192]. A subsequence of a tour is selected by choosing two random cut points, which serve as boundaries for the swapping operations. For example, the two parents (with two cut points marked by "|")

$p_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$ and
$p_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$

would produce offspring as follows. First, the segments between cut points are swapped (the symbol "x" can be interpreted as "at present unknown"):

$o_1 = (\mathrm{x}\ \mathrm{x}\ \mathrm{x} \mid 1\ 8\ 7\ 6 \mid \mathrm{x}\ \mathrm{x})$ and
$o_2 = (\mathrm{x}\ \mathrm{x}\ \mathrm{x} \mid 4\ 5\ 6\ 7 \mid \mathrm{x}\ \mathrm{x})$.

This swap also defines a series of mappings:

$1 \leftrightarrow 4$, $8 \leftrightarrow 5$, $7 \leftrightarrow 6$, and $6 \leftrightarrow 7$.

Then, we can fill in additional cities from the original parents for which there's no conflict:

$o_1 = (\mathrm{x}\ 2\ 3 \mid 1\ 8\ 7\ 6 \mid \mathrm{x}\ 9)$ and
$o_2 = (\mathrm{x}\ \mathrm{x}\ 2 \mid 4\ 5\ 6\ 7 \mid 9\ 3)$.

Finally, the first x in the offspring o_1 (which should be 1, but there was a conflict) is replaced by 4 because of the mapping $1 \leftrightarrow 4$. Similarly, the second x in offspring o_1 is replaced by 5, and the respective x and x in offspring o_2 are 1 and 8. The offspring are

$o_1 = (4\ 2\ 3 \mid 1\ 8\ 7\ 6 \mid 5\ 9)$ and
$o_2 = (1\ 8\ 2 \mid 4\ 5\ 6\ 7 \mid 9\ 3)$.

The PMX operator exploits similarities in the value and ordering simultaneously when used with an appropriate reproductive plan [192]. Nevertheless, an evolutionary algorithm in [142] offered better results in fewer function evaluations simply using a one-parent "remove-and-replace" variation operator on an ordinal representation for 100-city TSPs.