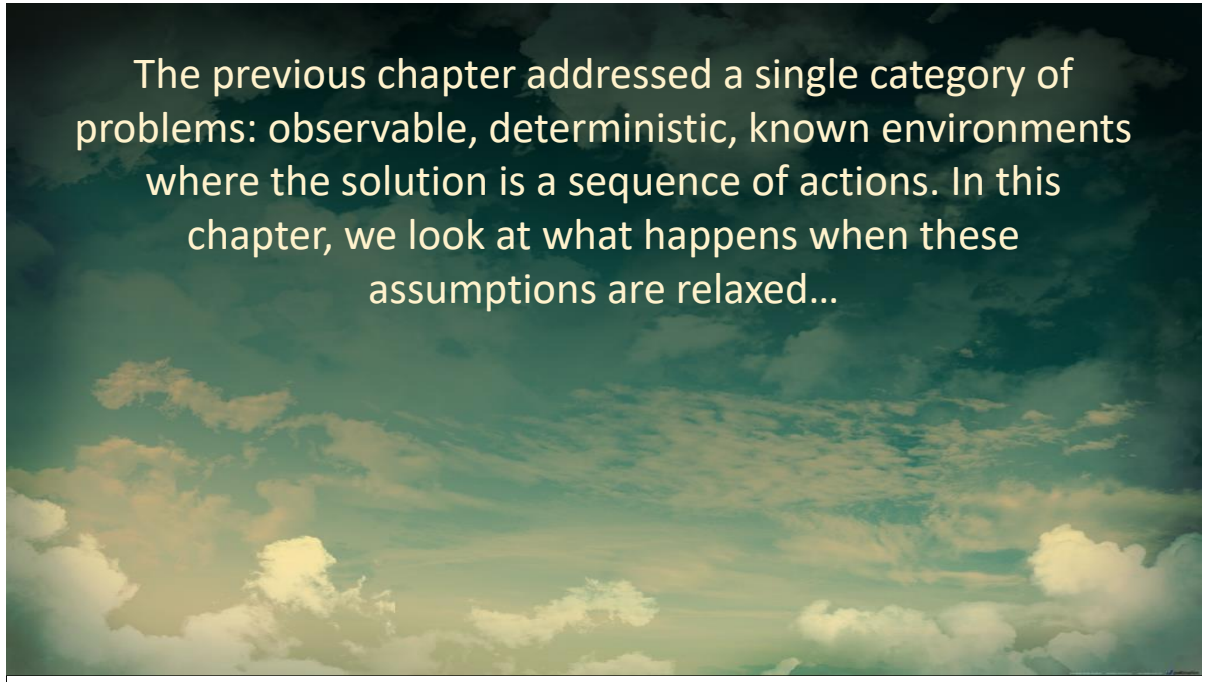
Artificial Intelligence

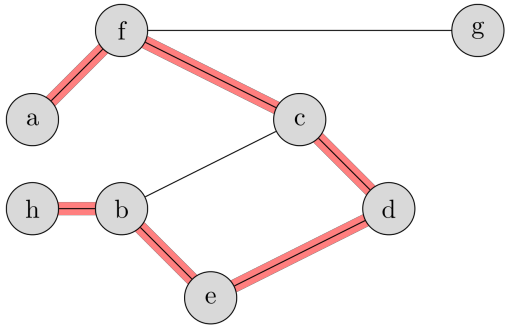
Beyond Classical Search

Ch 4



Systematicity of Classical Search

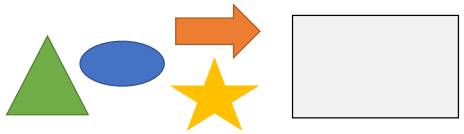
- The search algorithms that we have seen so far are designed to explore search spaces systematically.
  → If the space is finite, they will either find a solution or report that no solution exists.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives 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.



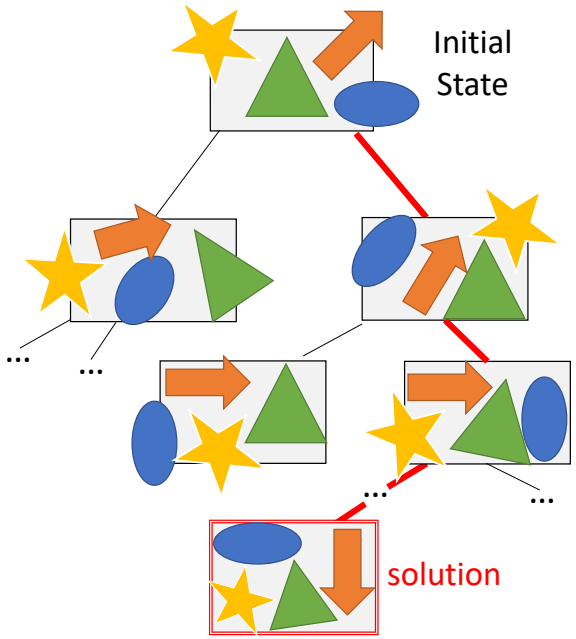
Beyond Classical Search

- In many problems, the path to the goal is irrelevant.
- For example, in the 8-queens problem 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 - Consider the problem of fitting the forms into a rectangle of metal such that we can cut copies of the forms from the metal.



- An initial state can be produced by positioning the forms at random positions and orientations.
- New states can be generated from previous ones by translating and rotating forms.
- Note that states and actions that led to the solution are irrelevant. Only the solution matters.



Local Search

- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.
- **Local search** algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Typically, the paths followed by the search are not retained. Only the current state is maintained in the memory.

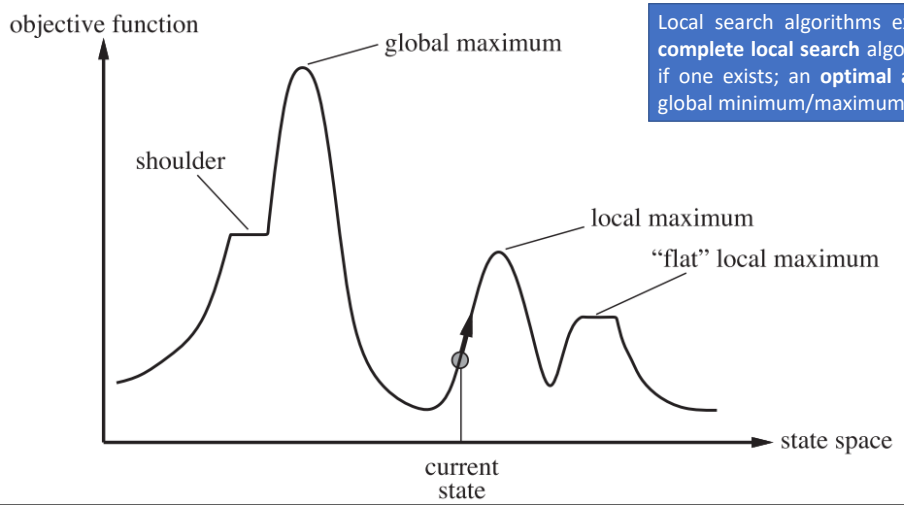
Advantages and Disadvantages of Local Search

- **Advantage**: local search uses very little memory - usually a constant amount.
- **Advantage**: local search can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- **Disadvantage**: local search algorithms are not systematic. Some states may never be reached. Therefore solutions may never be found even if they exist and they are not able to prove that no solution exists.

Local Search for Optimization

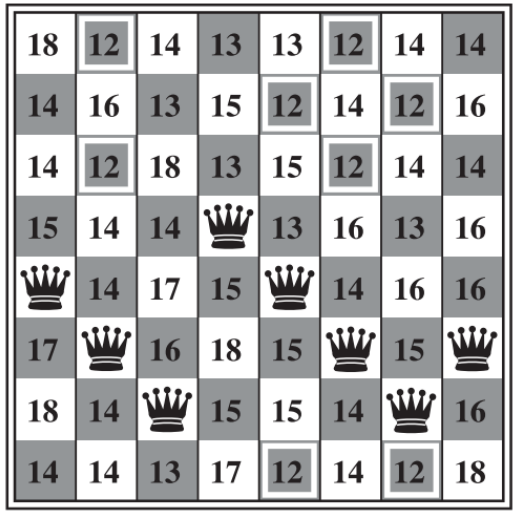
- In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.
  - The **objective function** in a mathematical optimization problem is a function that maps states to real values. Optimization consists of searching for states that minimize or maximize the objective function.
  - The **decision variables** in an optimization problem are those variables whose values can vary over the feasible set of alternatives in order to either increase or decrease the value of the objective function. For example, the position and orientation of the forms in the previous example or the position of the queens in the 8-queens problem.
- Many optimization problems do not fit the "standard" search model introduced in the previous chapter. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no "goal test" and no "path cost" for this problem.

The figure below shows the state-space landscape of an optimization problem. A landscape has both "location" (defined by the state) and "elevation" (defined by the objective function). The aim is to find the lowest valley - a global minimum or the highest peak - a global maximum (note that you can convert from one to the other just by inserting a minus sign).



Local search algorithms explore this landscape. A **complete local search** algorithm always finds a goal if one exists; an **optimal algorithm** always finds a global minimum/maximum.

- Consider the 8-queens problem with the complete-state formulation (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 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.
- The figure 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**. The best moves are marked.



Hill-Climbing Search

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

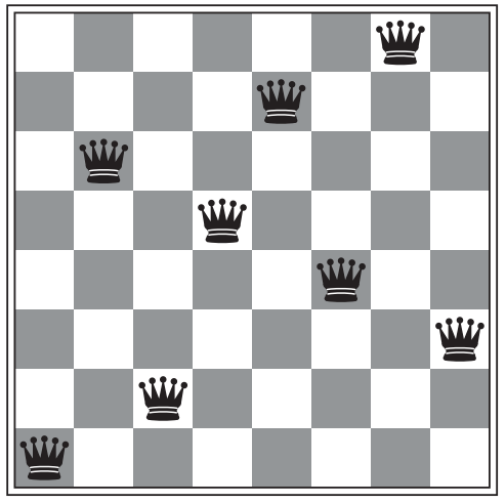
current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of *current*
if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
current ← *neighbor*

The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value (assuming a maximization problem) - that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

Hill-Climbing Search

- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- Hill climbing does not look ahead beyond the immediate neighbors of the current state.
- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- Hill climbing often makes rapid progress toward a solution, but, unfortunately, it often gets stuck due to local maxima, ridges, and plateaux.

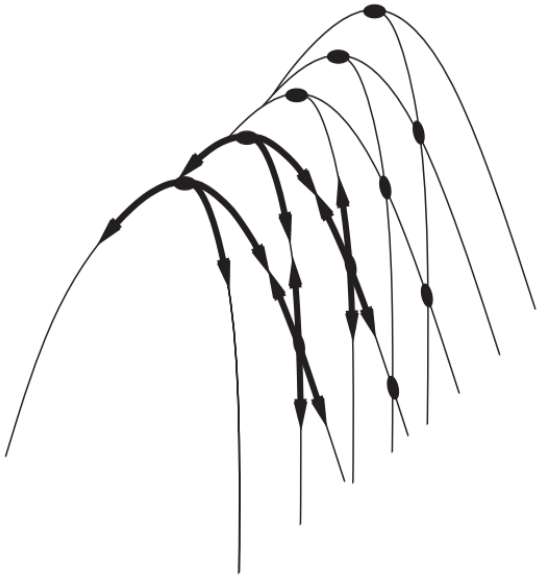
Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.



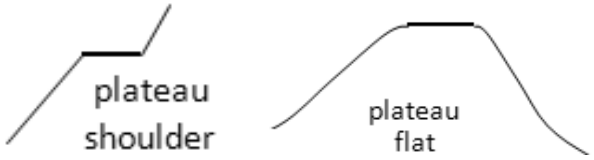
A local minimum in the 8-queens state space; the state has h=1 but every successor has a higher cost.

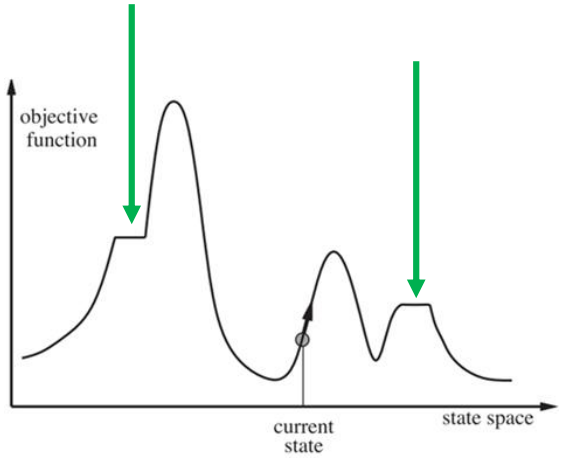
Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

The figure illustrates ridges. 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 maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau.





Variants of Hill-Climbing

- **Stochastic hill-climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- **First-choice hill-climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
- **Random-restart hill-climbing** is a complete version of hill-climbing. It adopts the well-known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

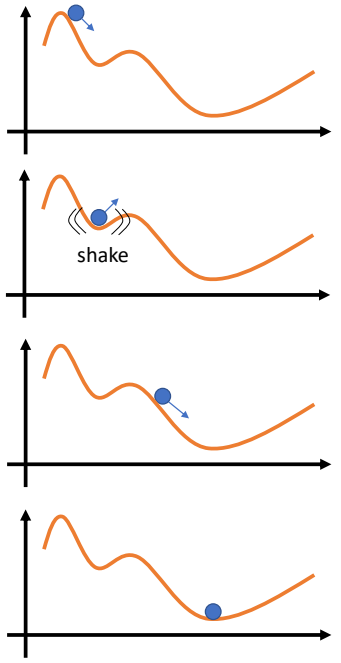
Limitations of Random-Restart Hill-Climbing

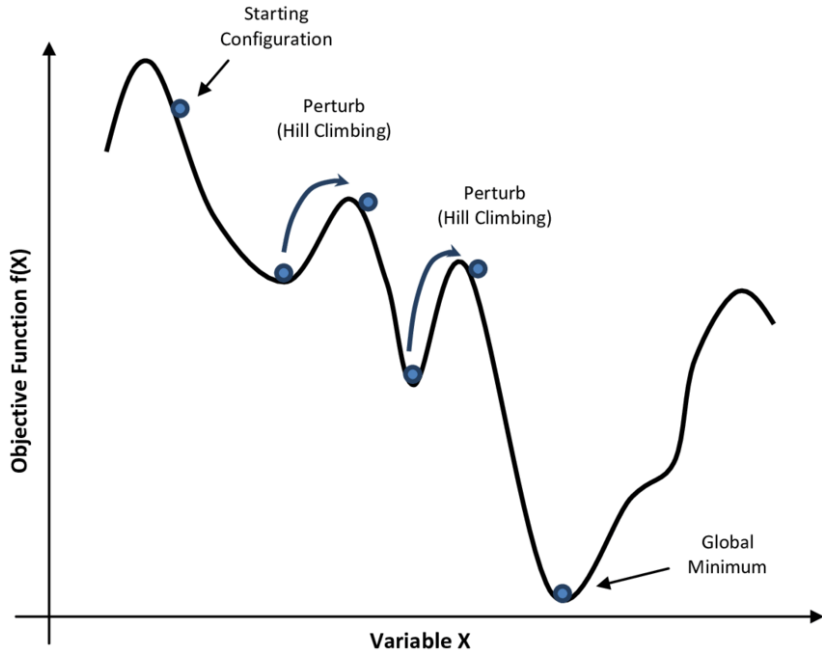
- The success of hill climbing depends very much on the shape of the state-space landscape: if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly.
- NP-hard problems typically have an exponential number of local maxima to get stuck on.
- Despite this, a reasonably good local maximum can often be found after a small number of restarts.

Simulated Annealing

- A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.
- In contrast, a purely random walk - that is, moving to a successor chosen uniformly at random from the set of successors - is complete but extremely inefficient.
- Simulated annealing is an algorithm that try to combine hill climbing with a random walk for yielding both efficiency and completeness.

- 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).





- Simulated-annealing is quite similar to hill climbing. However, instead of picking the best move, it picks a random move.
- If the move improves the situation, it is always accepted.
- Otherwise, the algorithm accepts the move with some probability.
- The probability decreases exponentially with the "badness" of the move - the amount δ 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.

pseudocode of simulated-annealing algorithm

```
initialize (temperature T, random starting point)
while cool_iteration <= max_iterations
    cool_iteration = cool_iteration + 1
    temp_iteration = 0
    while temp_iteration <= nrep
        temp_iteration = temp_iteration + 1
        select a new point from the neighborhood
        compute current_cost (of this new point)
        δ = current_cost - previous_cost
        if δ<0, accept neighbor
        else, accept with probability exp(-δ/T)
    end while
    T = α * T      (0<α<1)
end while
```

Sample a random number between 0 and 1 uniformly and if the number is smaller than the probability, accept the neighbor.

α : temperature reduction factor

Genetic Algorithm (GA)

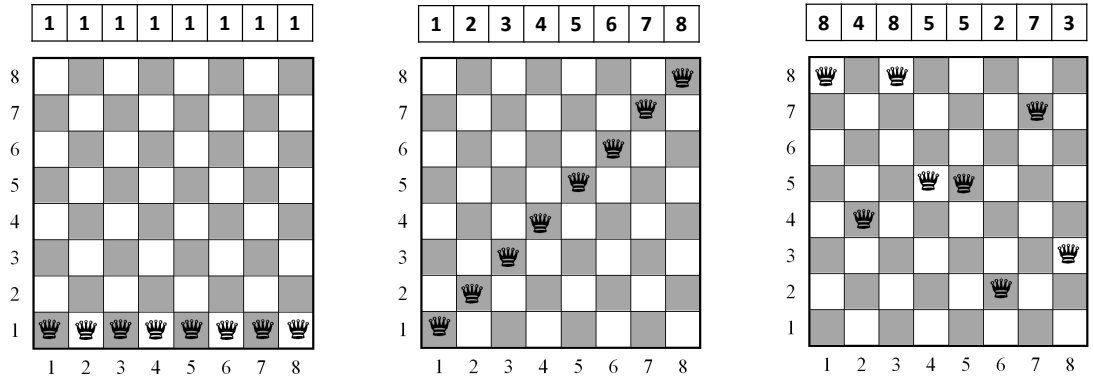
- There are several variants of GA. Here we present the traditional and most commonly used version.
- GA is a sample from a class of methods called evolutionary algorithms.
- In the context of evolutionary algorithms, objective functions are usually called fitness functions.
- The algorithm maintain a population of *k* states called individuals. Each individual is a potential solution. The algorithm iteratively create a new population from the previous one trying to increase the fitness of the individuals.

```
function GA (n, fitness_fn, k)
    Inputs: n, number of generations
            fitess_fn, fitness function
            k, population size

    population <- sample k random states
    pop_fitness <- fitness_fn(population)

    repeat for n generations
        Initialize a new_population with an empty set
        repeat k times
            Select two parents p1 and p2 by analysing the
                population the fitness of the individuals.
            Generate two childs c1 and c2 by performing
                cross-over between the parents
            Mutate the child c1 with some probability
            Mutate the child c2 with some probability
            Add c1 and c2 to the new population
        population <- new_population
        pop_fitness <- fitness_fn(population)
```

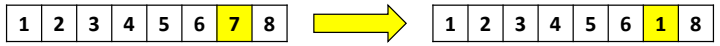
- **fitness_fn** is a function that compute the value of the objective function for all individuals in the population.
- **Tournament selection** is a common technique for selecting parents from the population. To generate each parent from the pair, two individuals are chosen at random from the population and the individual with highest fitness is selected (wins the tournament).
- The **cross-over** operation produces new individuals by "mixing" the parents. Different "mixing" strategies can be employed depending on the application.
- The **mutation** operation transforms the individual by changing the state value. The mutation strategies are also application-dependent.

Solving the 8-Queens Problem with GA

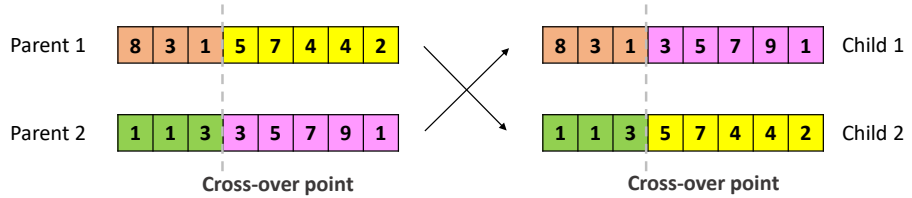
Let a state (or an individual) be represented as sequence of 8 numbers ranging from 1 to 8. Each of the 8 numbers represents the position (row) of a queen in a column of the board.



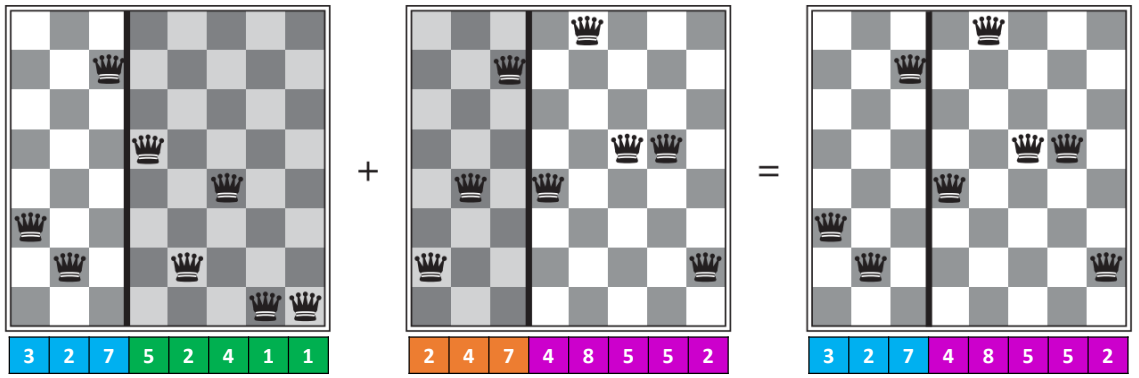
- **Fitness Function**: number of *nonattacking* pairs of queens, which has a value of 28 for a solution. GA will try to solve the problem by maximizing this fitness function.
- **Mutation**: Choose a queen at random and move it to a random square in its column.



- **Single-Point Crossover**: Choose a queen **q** at random. Create a child by merging the positions from **0** to **q** from the first parent and the positions **q+1** to **8** from the second parent. Do the opposite for the second child.



Visualization of the Single-Point Crossover Operation



IMPORTANT: Note that although the crossover was performed in the 3rd position in the presented examples, this is not by any means the default case. **The crossover point is chosen at random time the crossover operation is performed, i.e., for every pair of selected parents**.

The Exploration vs Exploitation Dilemma

- Note that cross-over and mutation are **exploration components** that may prevent getting stuck in local optima, while selection is an **exploitation operation** that aims at reaching the optima.
- Finding a good **trade-off between exploration and exploitation** is challenging, but also necessary. This trade-off is known as the **exploration vs exploitation dilemma**.
- **Too much exploration = jumping over the state space and never converging into optima**.
- **Too much exploitation = getting stuck in local optima and never reaching global optima**.

Searching in Continuous Spaces with Gradients

- Suppose we want to place three new airports anywhere in Romania, such that the sum of squared distances from each city on the map to its nearest airport is minimized.
- The state space is then defined by the coordinates of the airports: (x1,y1), (x2,y2),and (x3,y3).
- This is a six-dimensional space; we also say that states are defined by six variables.
- Moving around in this space corresponds to moving one or more of the airports on the map.

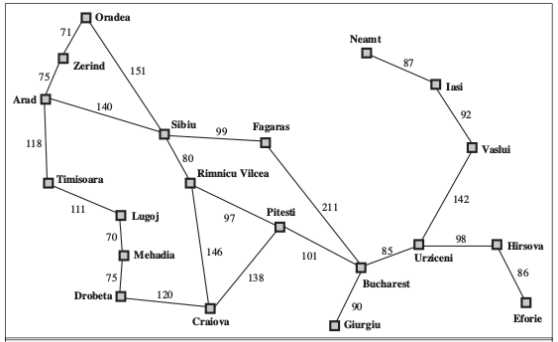


Figure 3.2 A simplified road map of part of Romania.

The objective function f(x1,y1,x2,y2,x3,y3) is relatively easy to compute for any particular state once we compute the closest cities. Let C_i be the set of cities whose closest airport (in the current state) is airport i. Then, in the neighborhood of the current state, where the C_is remain constant, we have:

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2 \ .$$

This expression is correct locally, but not globally because the sets C_i are (discontinuous) functions of the state. Many methods attempt to use the gradient of the landscape to find a maximum. **The gradient of the objective function is a vector ∇f that gives the magnitude and direction of the steepest slope**. For our problem, we have:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right) \ .$$

- In some cases, we can find a maximum by solving the equation ∇f =0. This could be done, for example, if we were placing just one airport; the solution is the arithmetic mean of all the cities' coordinates.
- In many cases, however, this equation cannot be solved in closed form.
- For example, with three airports, the expression for the gradient depends on what cities are closest to each airport in the current state. This means we can compute the gradient locally (but not globally):

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c) \ .$$

Given a locally correct expression for the gradient, we can perform steepest-ascent hill-climbing (also called **gradient ascent** or **gradient descent** in case of minimization) by updating the current state according to the formula:

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

where α is a small constant often called the **step size (or learning rate in machine learning)**. Choosing α is an important problem. **If α is too small, too many steps are needed for reaching optima; if α is too large, the search could overshoot them**.

When the objective function is not differentiable or is not available in a differentiable form, we can calculate a so-called **empirical gradient by evaluating the response to small increments and decrements in each coordinate**.

Constrained Optimization and Linear Programming

- An optimization problem is constrained if solutions must **satisfy some hard constraints on the values of the variables**.
- For example, in our airport-siting problem, we might constrain sites to be inside Romania and on dry land (rather than in the middle of lakes).
- The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function.
- The best-known category is that of **linear programming** problems, in which **constraints must be linear inequalities** forming a convex set and **the objective function is also linear**.
- **The time complexity of linear programming is polynomial in the number of variables**.

Convex Optimization

- Linear programming is a special case of the more general problem of convex optimization, which allows the constraint region to be any convex region and the objective to be any function that is convex within the constraint region.
- **A set of points *S* is convex if the line joining any two points in *S* is also contained in *S*. A convex function is one for which the space "above" it forms a convex set; by definition, convex functions have no local (as opposed to global) optima (all optima are global)**.
- **Under certain conditions, convex optimization problems are also polynomially solvable and may be feasible in practice with thousands of variables**.
- Several important problems in machine learning and control theory can be formulated as convex optimization problems.