



**Maynooth
University**
National University
of Ireland Maynooth



CS211FZ: Data Structures and Algorithms II

Basic and Brute force methods

LECTURER: CHRIS ROADKNIGHT

CHRIS.ROADKNIGHT@MU.IE

Introduction (1)



These are the algorithmic techniques that most of us are most familiar with, even if we don't realise it.

For instance, a brute force algorithm will try all possible solutions to the problem, only stopping when it finds one that is the actual solution.

A good example of a brute force algorithm in action is plugging in a USB cable. Many times, we will try one way, and if that doesn't work, flip it over and try the other.

Likewise, if we have a large number of keys but are unsure which one fits in a particular lock, we can just try each key until one works. That's the essence of the brute force approach to algorithmic design.

Basic Search Methods

Brute Force

Sampling

Hill Climbing

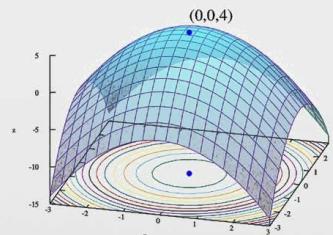
Taboo Search

Genetic Algorithms

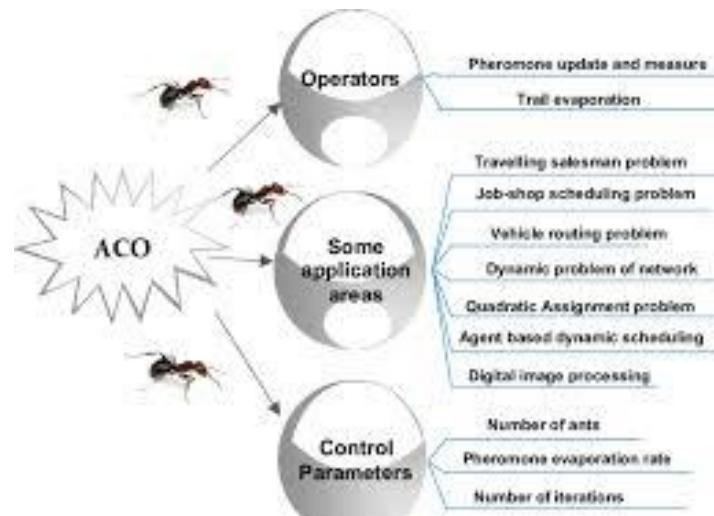
Random walk

Ant Colony Optimisation

Tabu search



https://en.wikipedia.org/wiki/File:Max_paraboloid.svg



Introduction – Brute Force

A brute force algorithm solves a problem through exhaustion: it goes through all possible choices until a solution is found.

The time complexity of a brute force algorithm is proportional to the input size.

Brute force algorithms are simple and consistent, but very slow.

$O(n)$ where n is the number of possible solutions

Pros of brute force algorithm

- ❑ The brute force approach is a guaranteed way to find the correct solution by listing all the possible candidate solutions for the problem.
- ❑ It is a generic method and not limited to any specific domain of problems.
- ❑ The brute force method is ideal for solving small and simpler problems.
- ❑ It is known for its simplicity and can serve as a comparison benchmark.

Cons of brute force algorithm

- ❑ The brute force approach is usually inefficient.
- ❑ This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- ❑ Brute force algorithms are slow.
- ❑ Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.
- ❑ But can be essential as comparators or when the data is extremely noisy or non-linear

The Traveling-Salesman Problem (TSP)



The traveling-salesman problem

In the traveling-salesman problem, a salesman must visit n cities.

Let's model the problem as a complete graph with n vertices, so that the salesperson wishes to make a tour, visiting each city exactly once and finishing at the starting city.

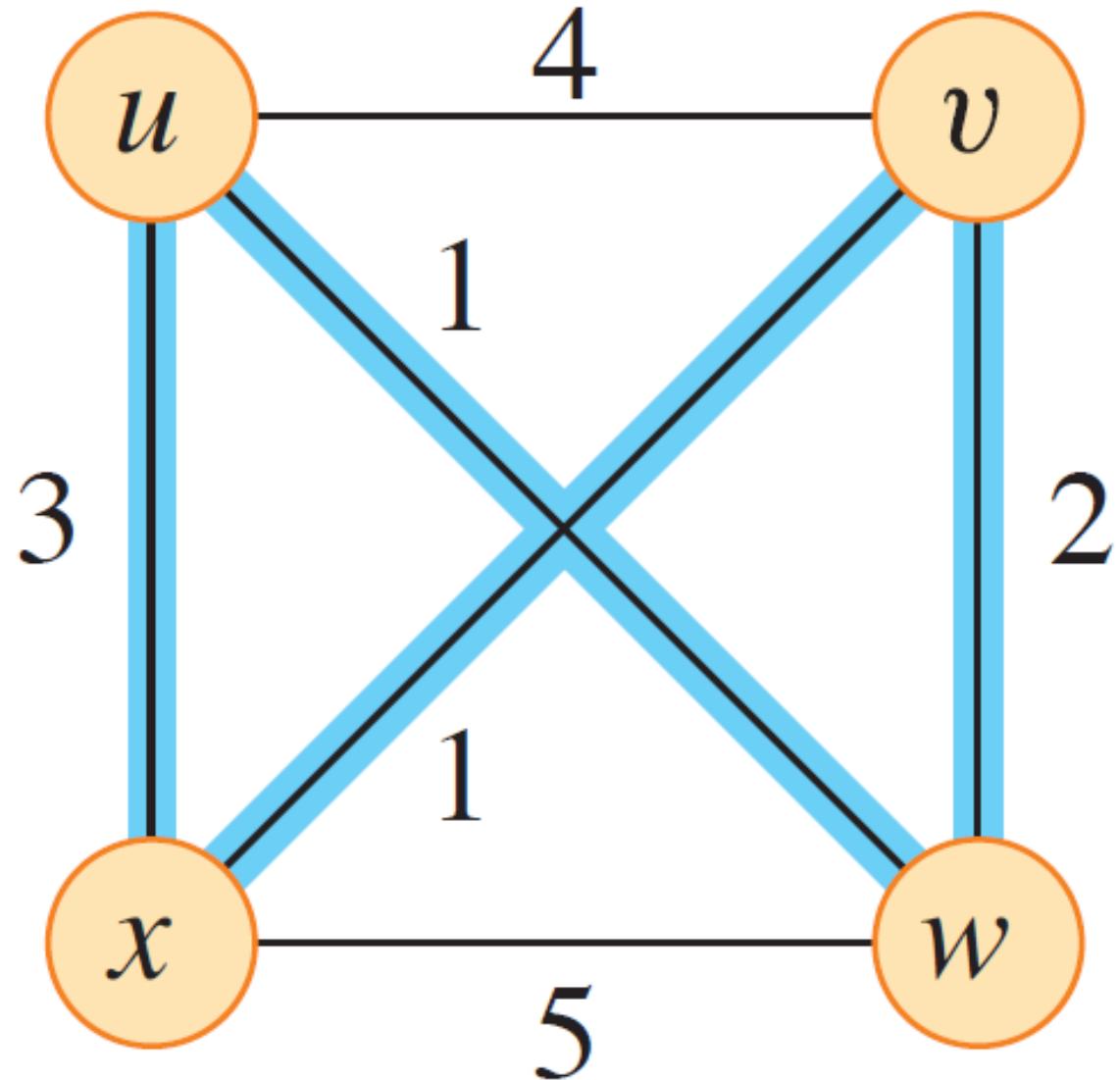
The salesperson incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j .

In the optimization version of the problem, the salesperson wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

Example

For example, in the following figure, a minimum-cost tour is $\langle u, w, v, x, u \rangle$, with cost 7.

How many tours are there?



Brute force method for TSP

When one thinks of solving TSP, the first method that might come to mind is a brute-force method.

The brute-force method is to simply generate all possible tours and compute their distances. The shortest tour is thus the optimal tour. To solve TSP using Brute-force method we can use the following steps:

Step 1. calculate the total number of tours.

Step 2. draw and list all the possible tours.

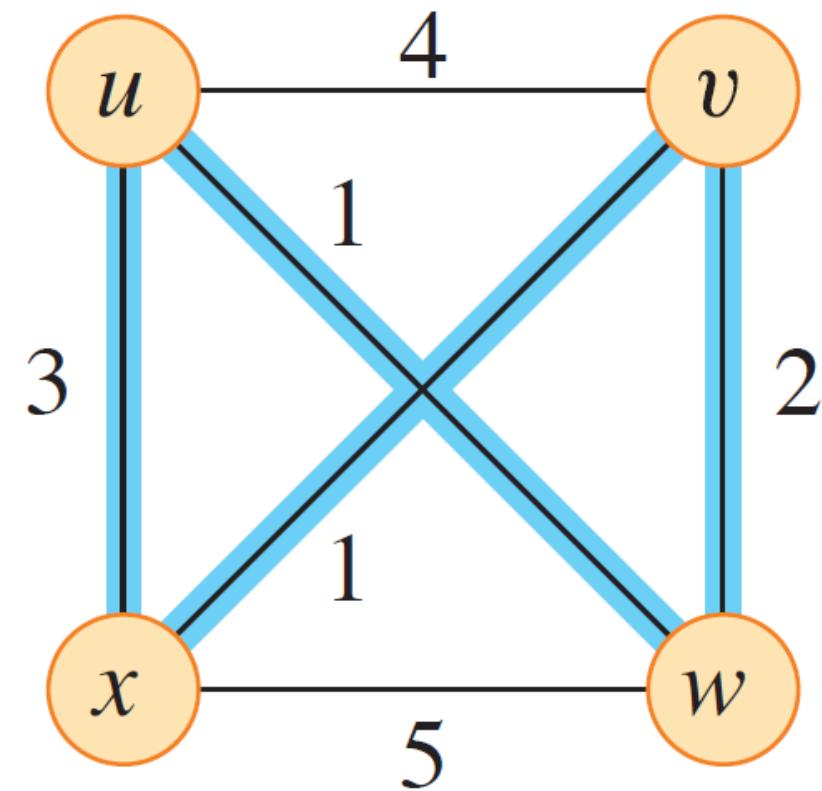
Step 3. calculate the distance of each tour.

Step 4. choose the shortest tour, this is the optimal solution.

Example for Brute-Force Technique for TSP

u	v	x	w	u
u	v	w	x	u
u	w	x	v	u
u	w	v	x	u
u	x	w	v	u
u	x	v	w	u

Every tour has a mirrored image



TSP procedure – pseudocode

It is assumed that if there is not a connection between vertices i and j , then $G.\text{Edge}(i, j).d$ is equal to ∞ , where d represents the cost of moving from vertex i to vertex j .

TSP (G, s)

```
1   Vertices = All vertices in G - {s}
2   MinPath =  $\infty$  //store minimum weight/distance for traveling cycle
3   for each P ∈ Permutations(Vertices)
4       CurrentPathWeight = 0 //store current Path weight/distance
5       k = s
6       for each v ∈ P:
7           CurrentPathWeight += G.Edge(k, v).d
8           k = v
9       CurrentPathWeight += G.Edge(k, s).d
10      MinPath = min(MinPath, CurrentPathWeight)
11  return MinPath
```

TSP procedure – explanation

Line 1 assigns all the vertices, except vertex s , in the graph G to vertices .

Line 2 initialize the MinPath variable with an infinite value.

The procedure $\text{Permutations}()$ in line 3 calculating and returning all the possible permutations of vertices .

The for loop in lines 3-10 is calculating the cost/distance of the each of the permutations and in line 10 assign the so far minimum tour to the MinPath variable.

The for loop in lines 6-8 is calculating the distance of the current tour/ permutation and cumulatively add the distances in CurrentPathWeight .

Permutations (review)

A permutation of a finite set S is an ordered sequence of all the elements of S , with each element appearing exactly once. For example, if $S = \{a, b, c\}$, then S has 6 permutations:

$abc, acb, bac, bca, cab, cba.$

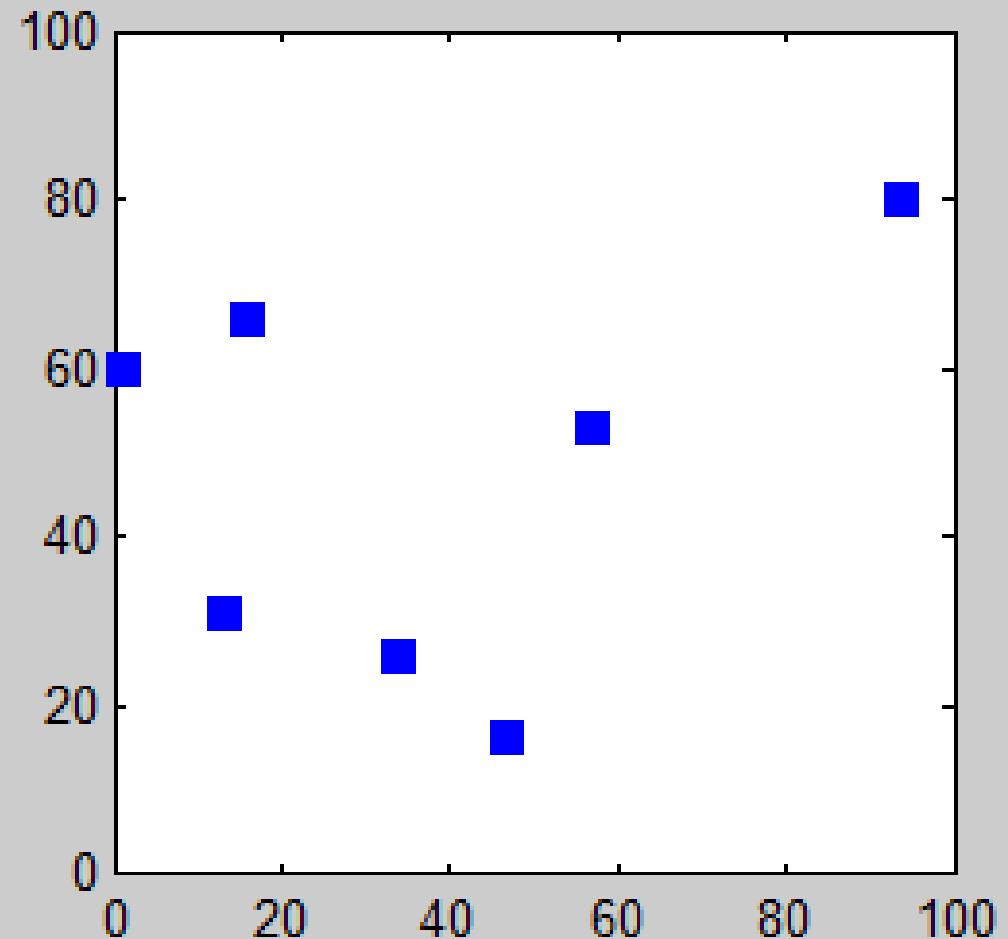
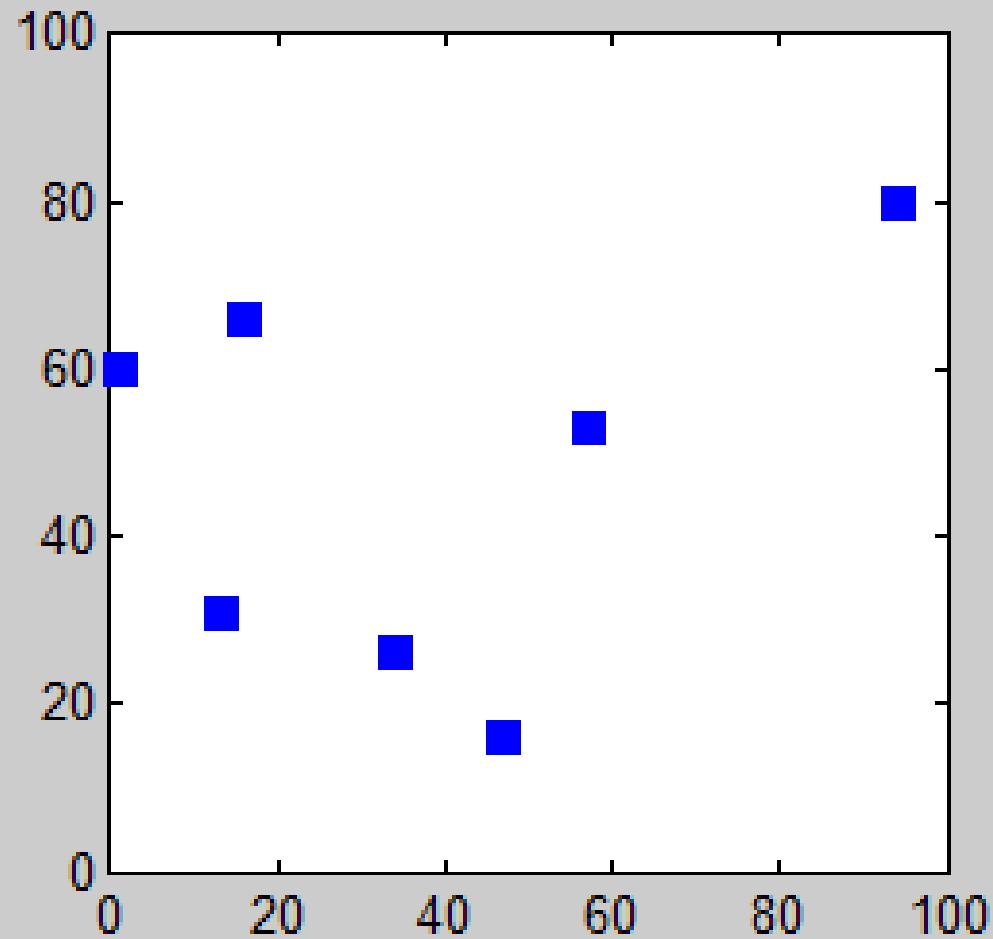
There are $n!$ permutations of a set of n elements, since there are n ways to choose the first element of the sequence, $n - 1$ ways for the second element, $n - 2$ ways for the third, and so on.

TSP procedure – running time

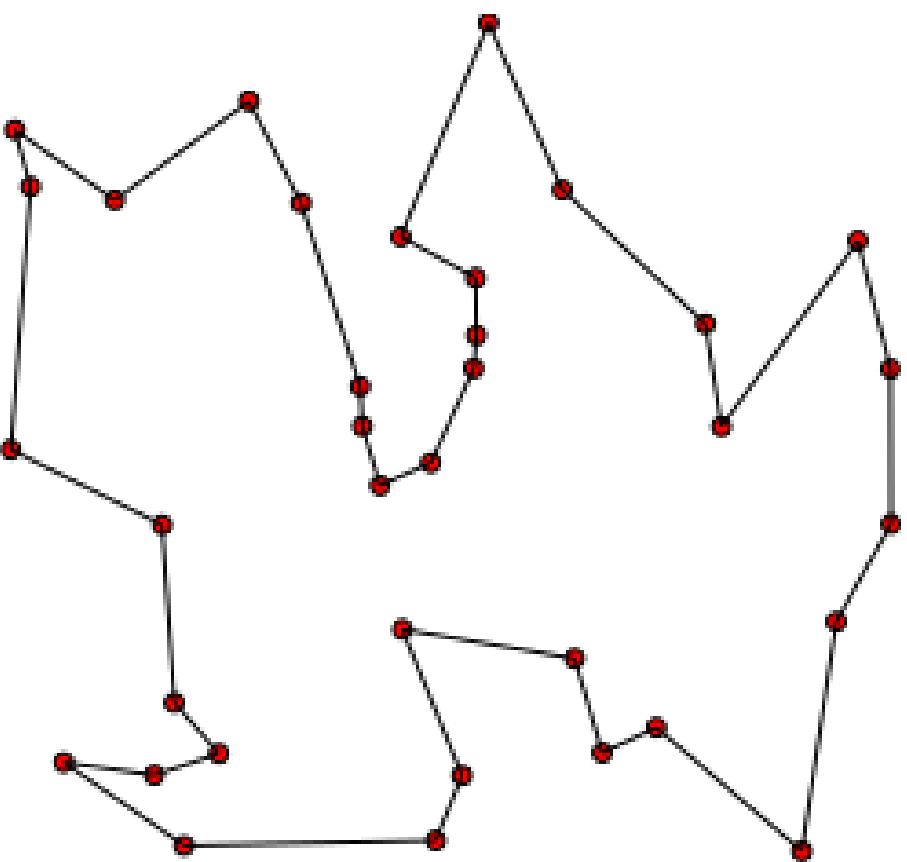
The running time of the **for** loop in lines 3 to 10 is the number of permutations of the vertices. Thus, the running time of the outer loop is $(V - 1)!$, where V is the number of vertices. (4 vertices = $3 \times 2 \times 1$)

The purpose of the **for** loop in lines 6 to 8 is visiting all the vertices; thus, in each iteration of the outer loop, the running time of the inner loop is V .

Therefore, the total running time of the TSP procedure is $V \times (V - 1)!$, which is $\Theta(V!)$.



How many paths here?



2.95233E+38

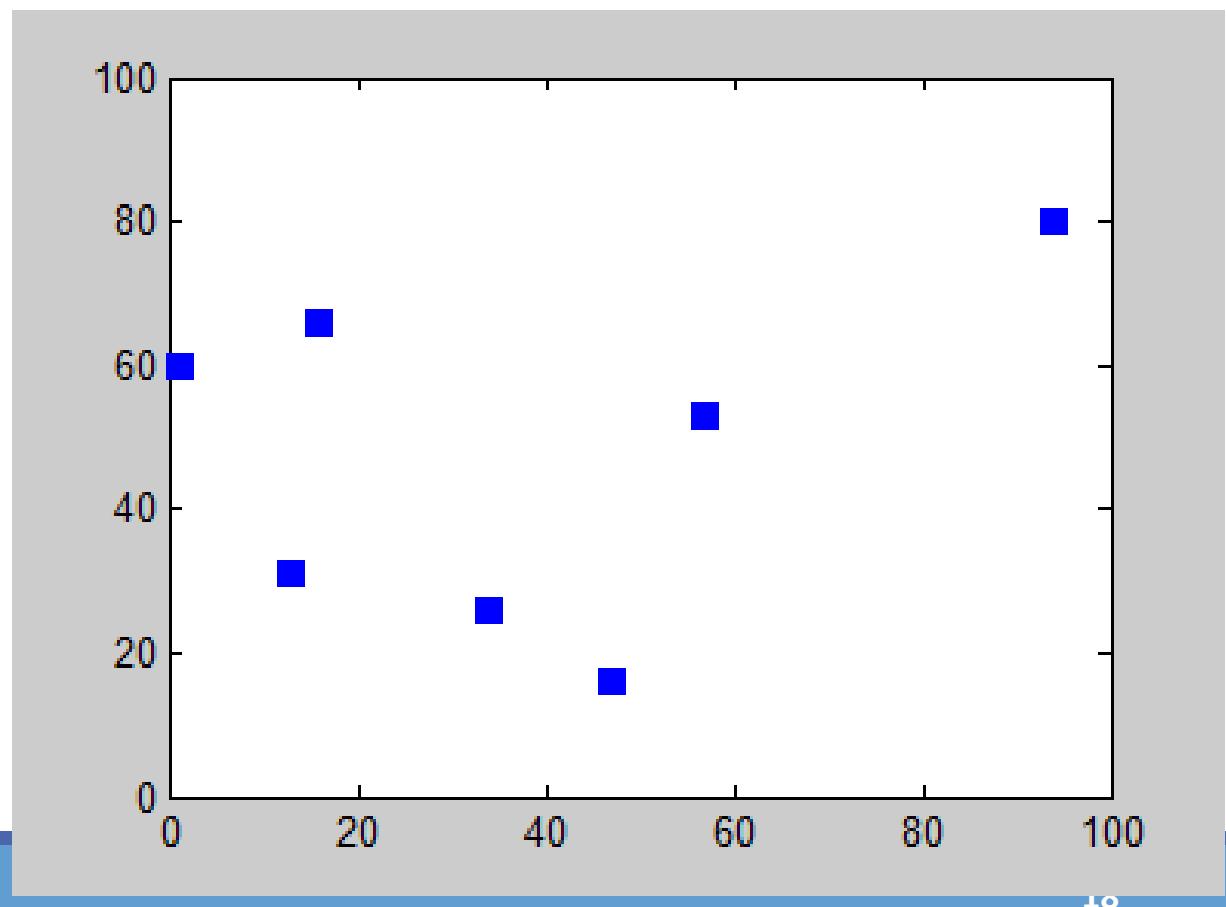
How do we better than brute force but still keeps it simple

Any method that doesn't check every solution is never sure it is correct, but can produce a good answer fast.

E.g. nearest neighbour

Gets a different distance each time
(249 was the global optimal)

Nn just check all options and
picks the nearest



Ant Colony Optimisation 1 (ACO)

Ants and Pheromones: ACO mimics how ants find the shortest path to food by leaving pheromone trails.

In the algorithm, “ants” (little programs) explore possible solutions (like routes), and better paths get more “pheromone” (a score), guiding future ants to pick those paths more often.

One of many branches of ‘biologically inspired computing’



ACO(2)

In order not to forget the route home, leave small amounts of pheromone on the ground.

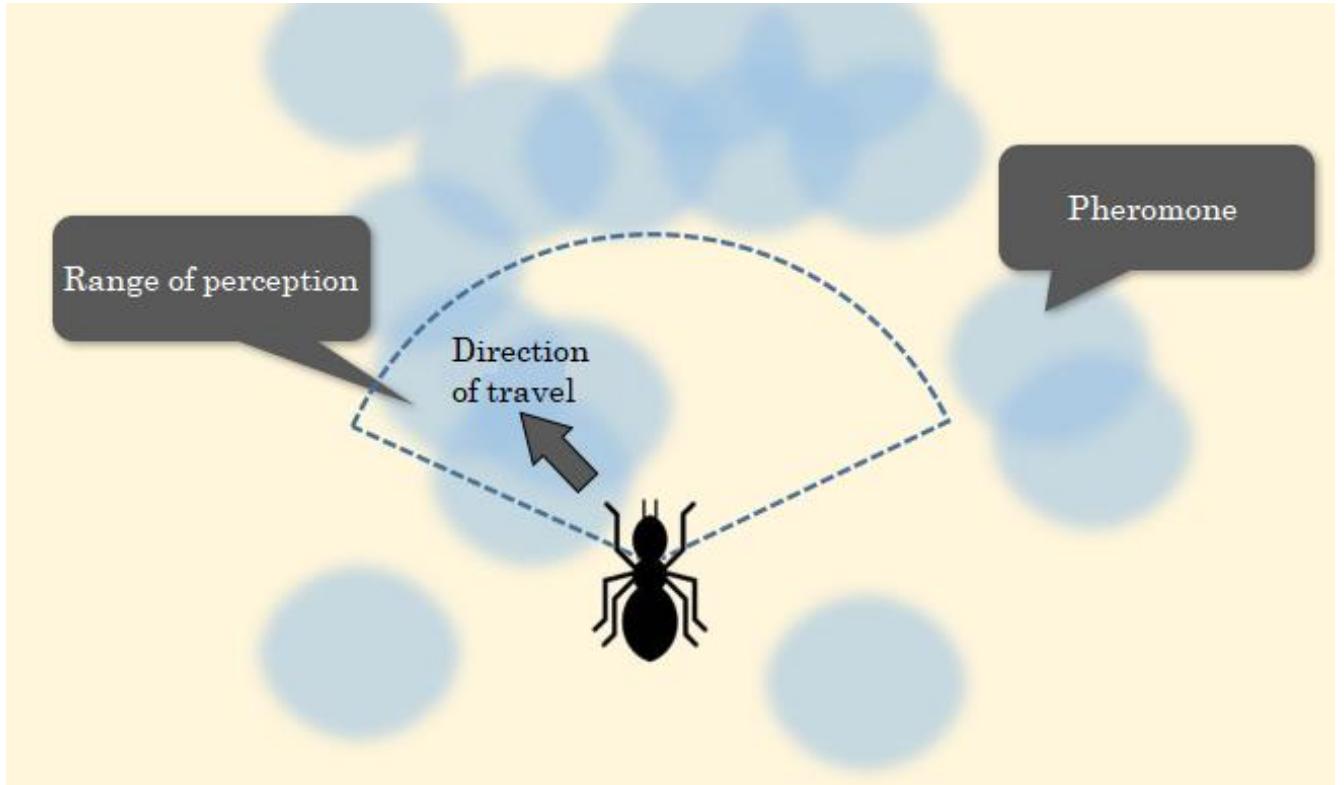
Probability and Updates: Each ant chooses its path based on pheromone levels and a heuristic (like distance)—stronger pheromones and shorter paths are more likely picks.

After all ants explore, pheromones on good paths get boosted, while weaker ones fade, refining the solution over time.



ACO(3)

Solving Hard Problems: ACO is great for NP-hard problems like the Traveling Salesman Problem (finding the shortest route through cities). It doesn't guarantee the perfect answer but often finds a really good one fast by balancing exploration (trying new paths) and exploitation (sticking to known good paths).



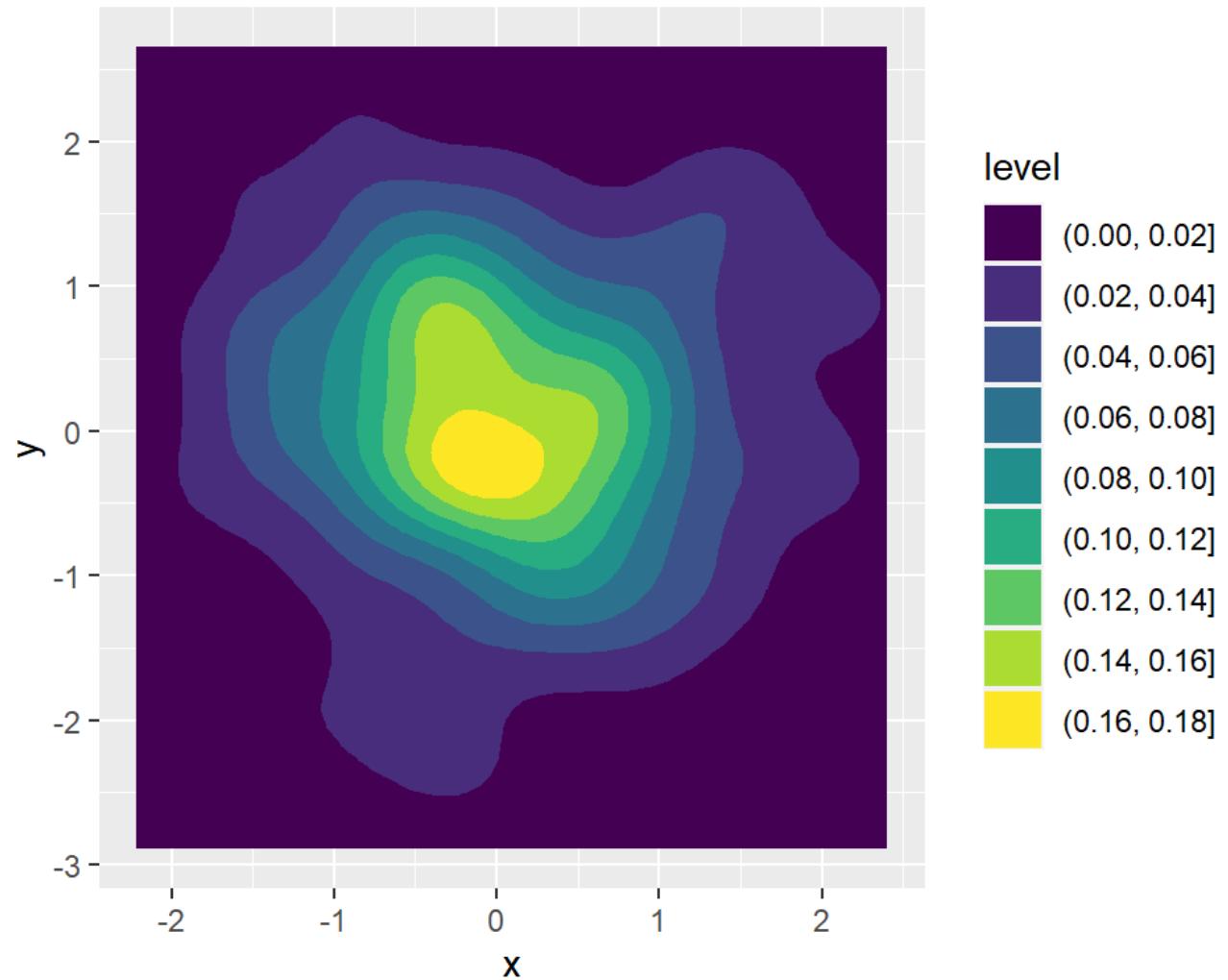
Sampling

Instead of brute force evaluation we can sample a percentage of all available options

No guarantee that we will find the best option but a statistical analysis of those samples may lead to an estimate of performance.

A calculation is made in advance based on how long brute force would take vs how long we have

Eg, It would take 100 hours for brute force but we have one hour. Which 1 % of solutions to try

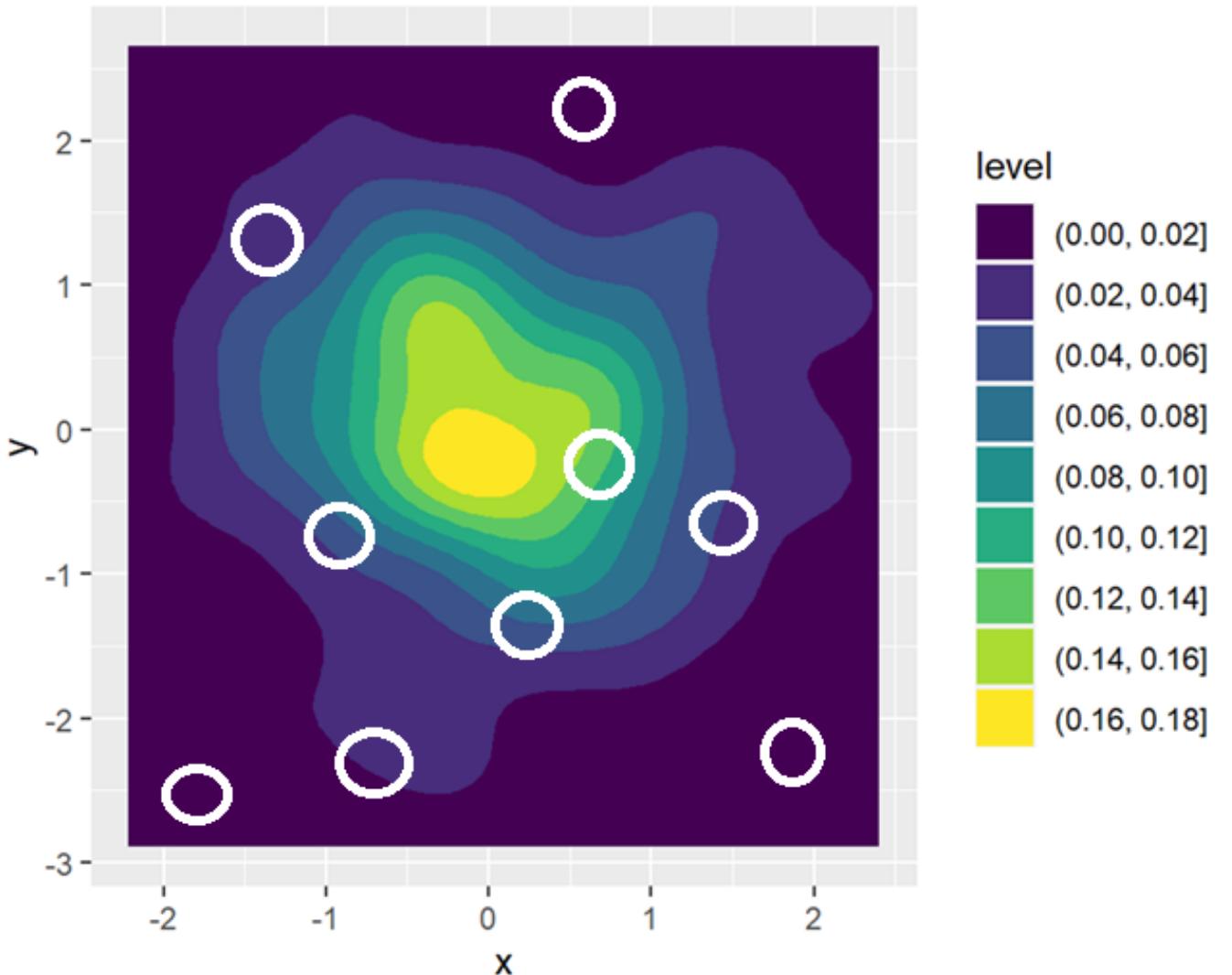


Finding the highest point on a contour plot

AS AN ANALOGY FOR
FINDING THE
GLOBAL MAXIMA IN
N DIMENSIONAL
FUNCTION SPACE

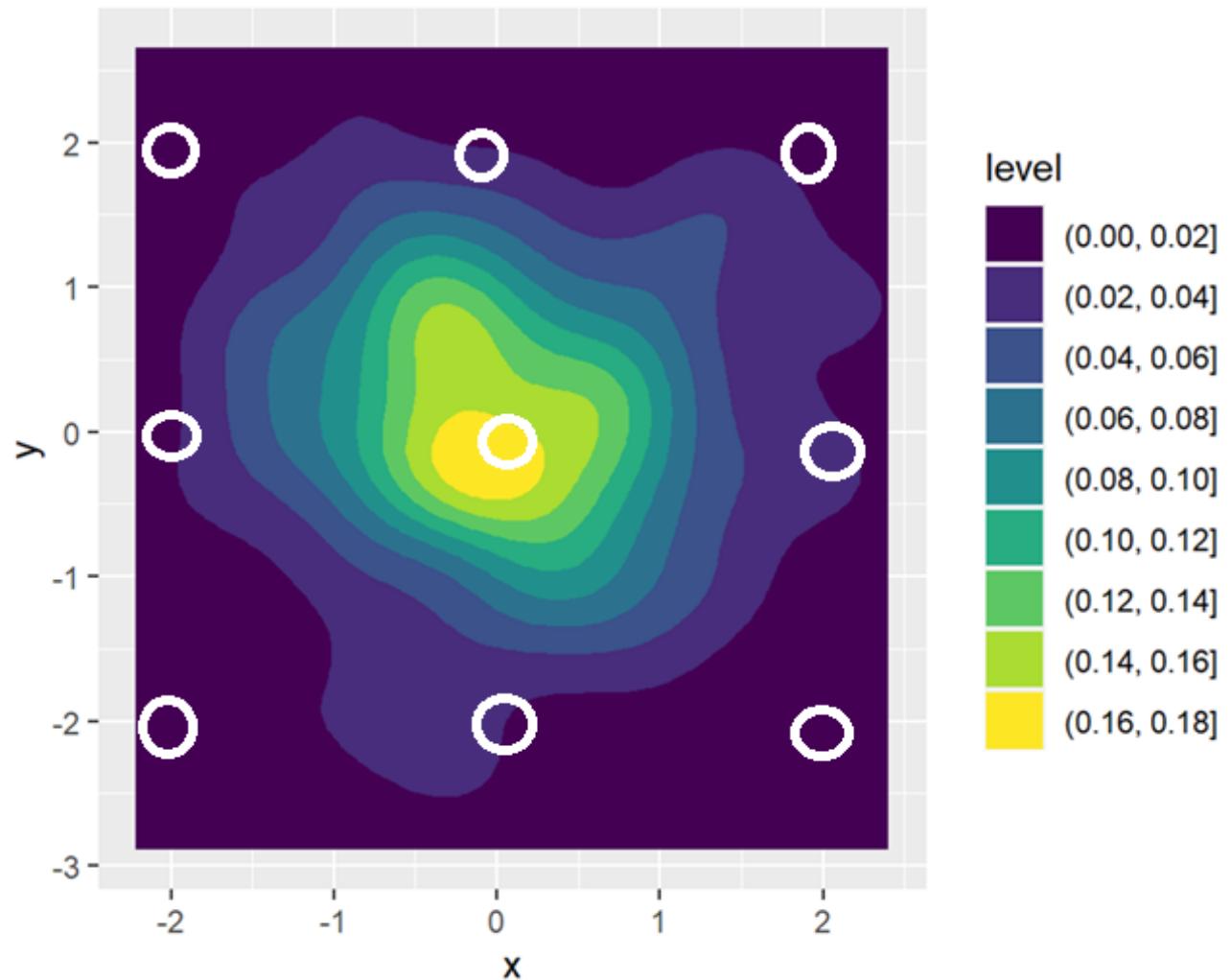
Random Sampling

Choose parameters using a random number generator



Round Robin sampling

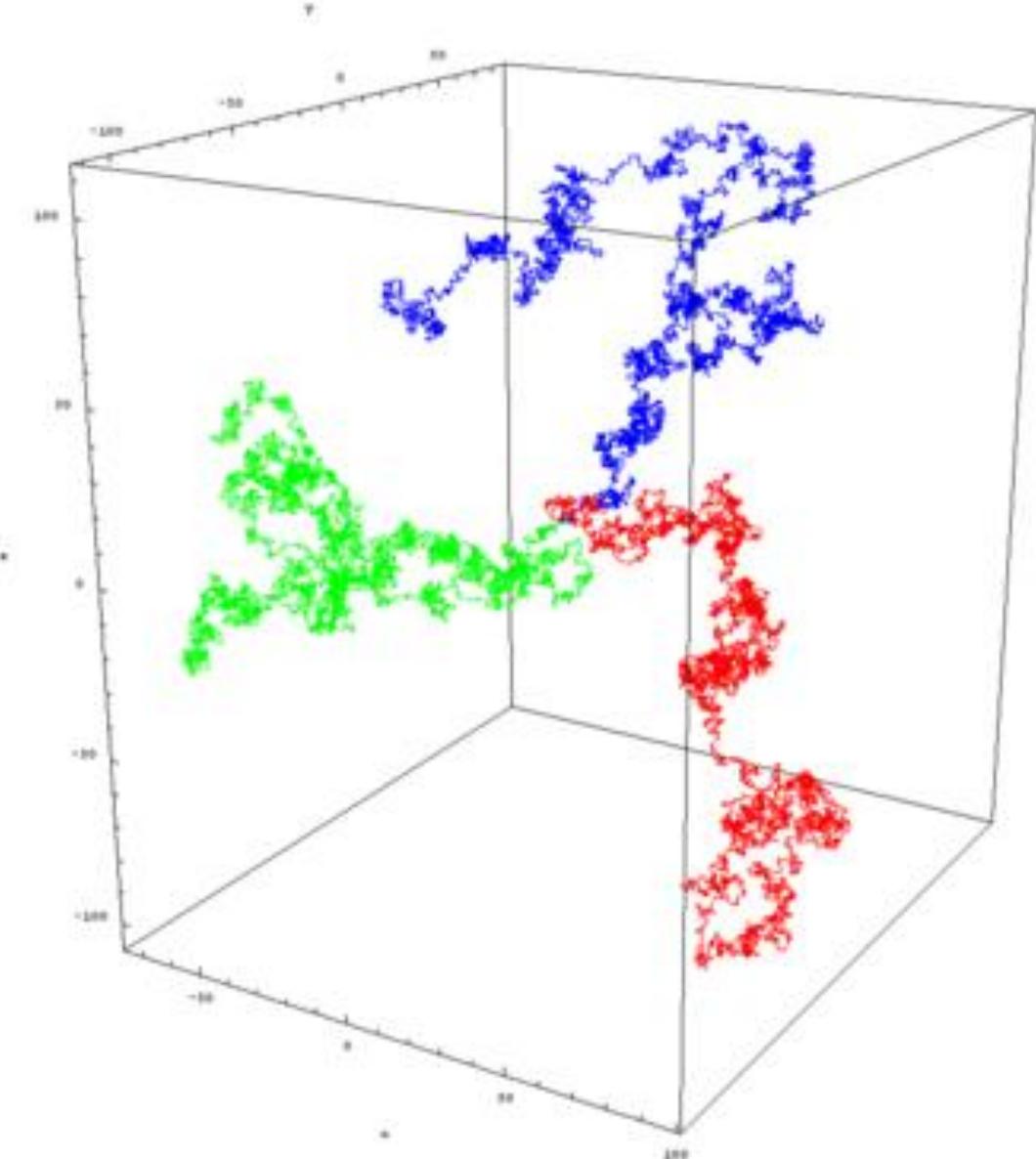
Choose evenly spaced samples.



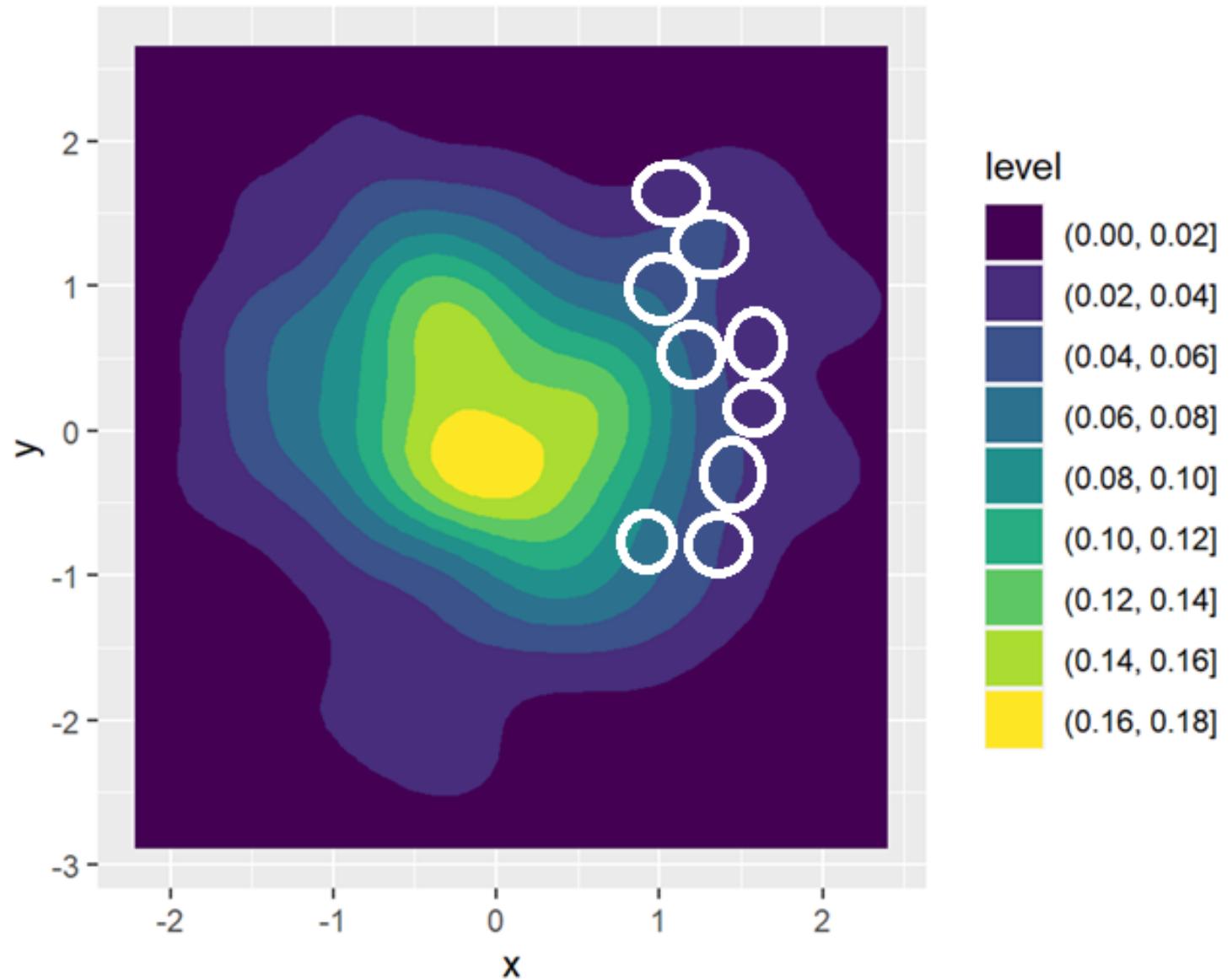
Random walk

Moves around an N dimensional problem space randomly, evaluating the N dimensional function at each stop

A version of sampling that is locally bound



Random Walk



Hill Climbing

An algorithm for finding (close to)
Global optimal approximations

Finds the optimal solution by making incremental
changes to an existing solution and then evaluating
whether the new solution is better than the current best.

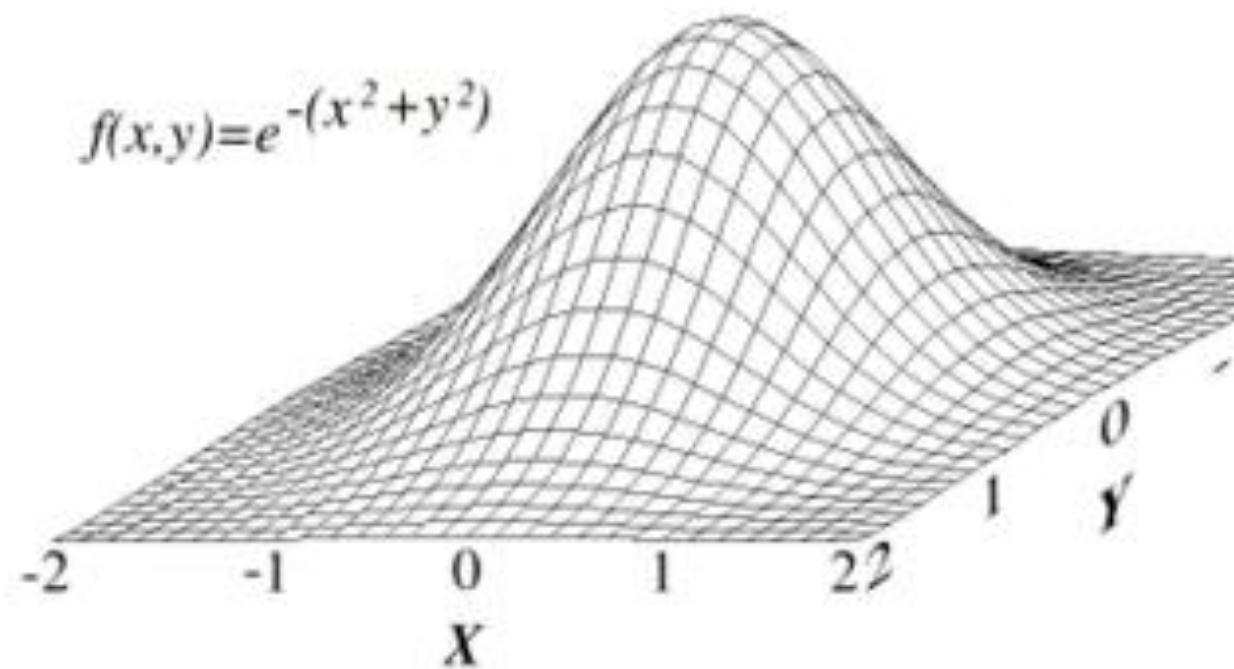
The process is analogous to climbing a hill where you
continually seek to improve your position until you reach
the top

Random walk

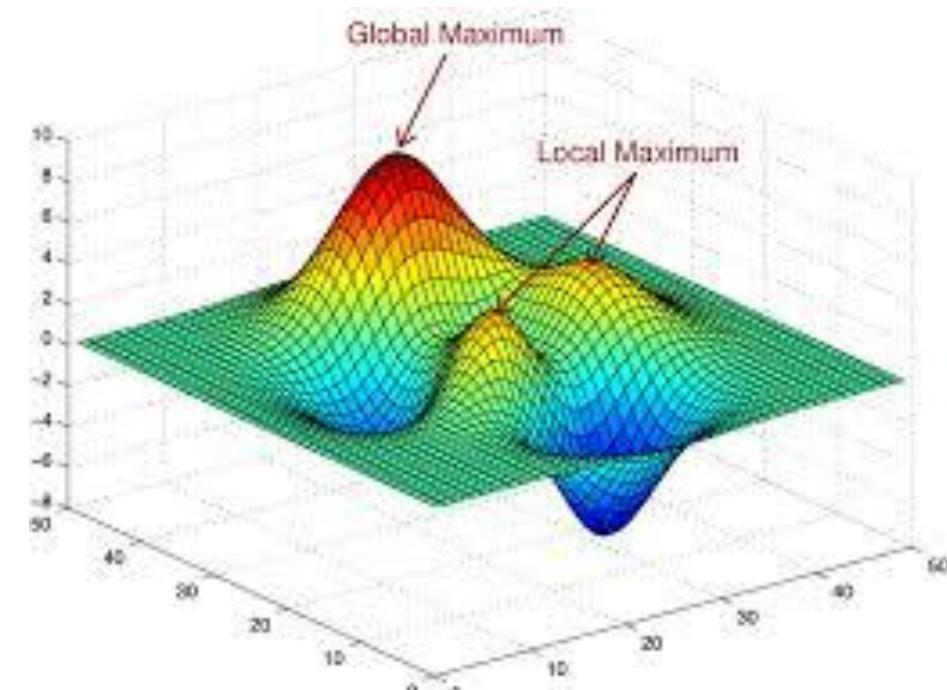
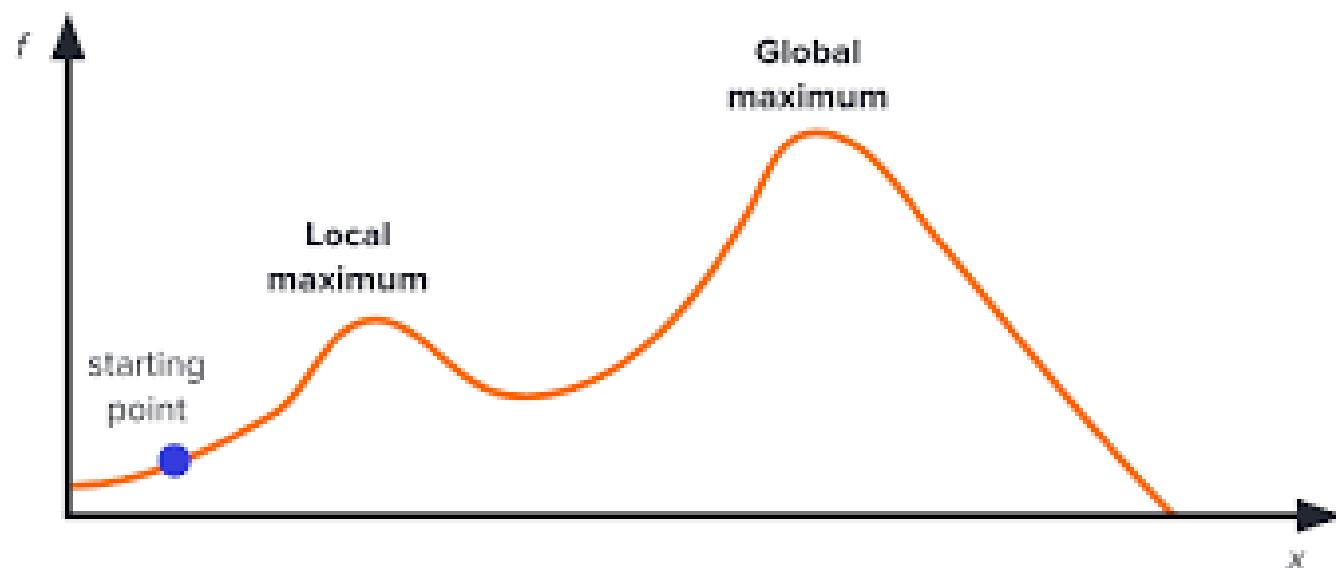
Have I improved

if yes, carry on in that 'direction'

if no, backtrack and change 'direction'

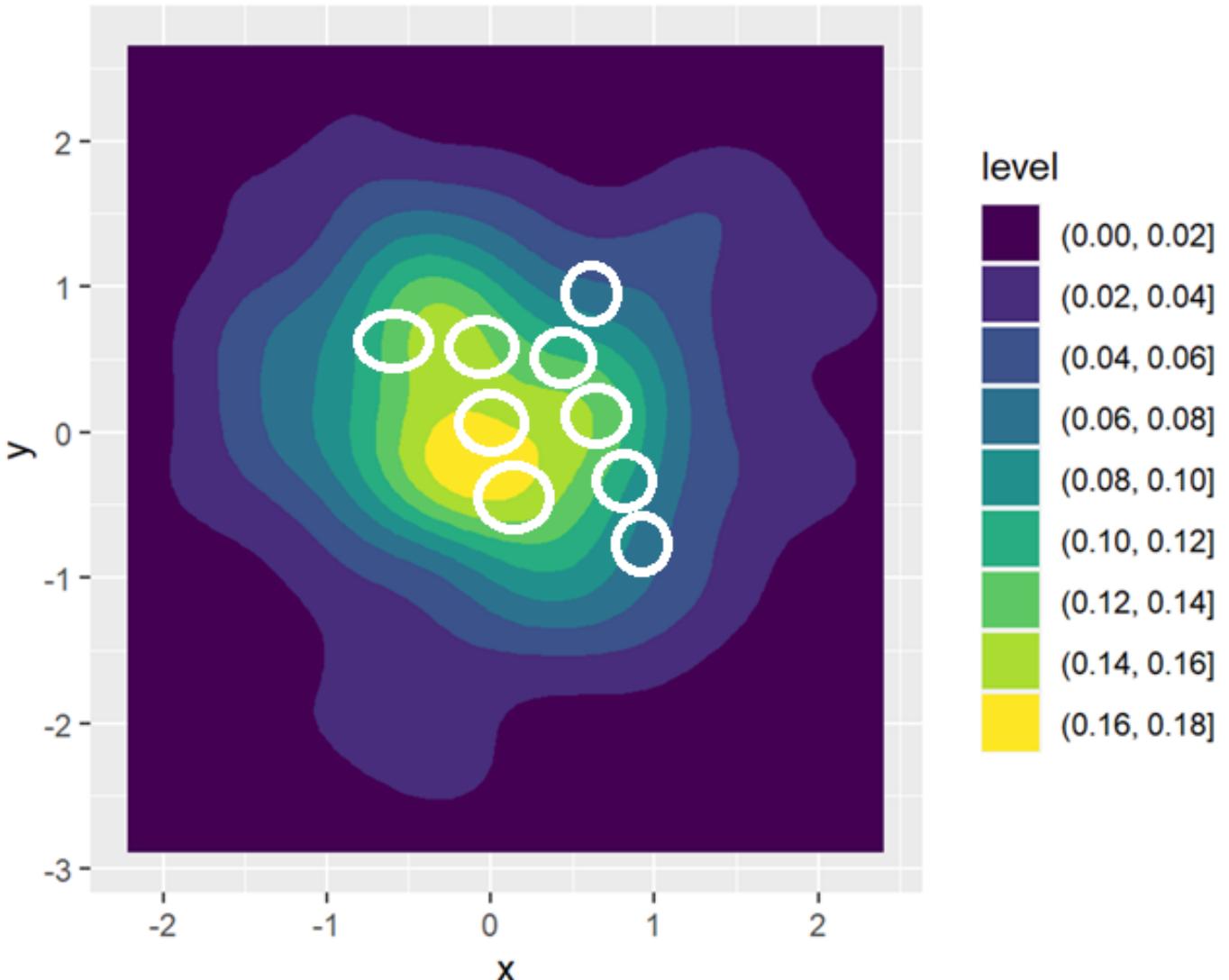


Global Optimal vs Local Optimal

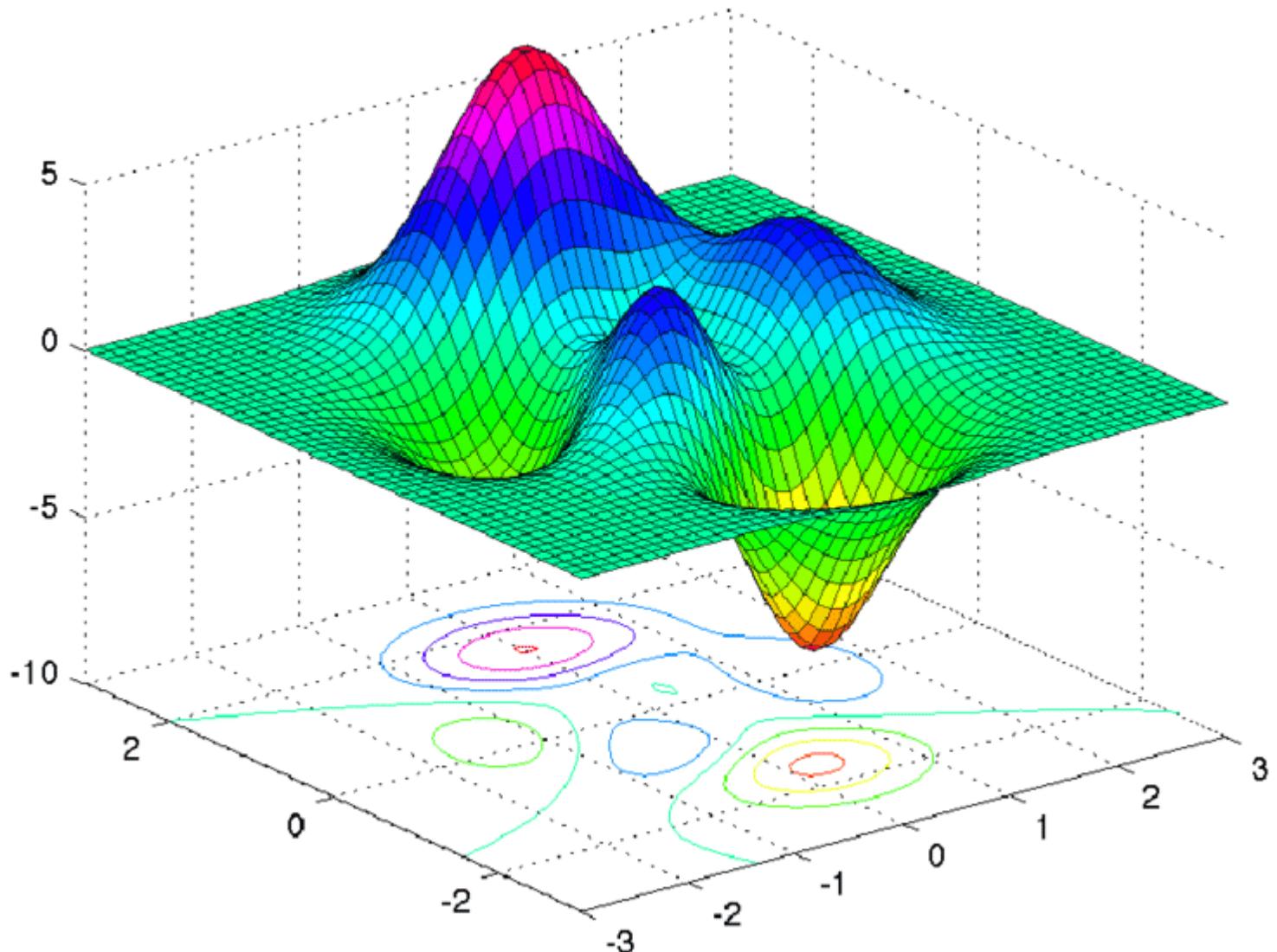


Hill Climbing

For a problem with simple linear properties, hill climbing is a good use of limited resources...
...but we don't know the properties in advance



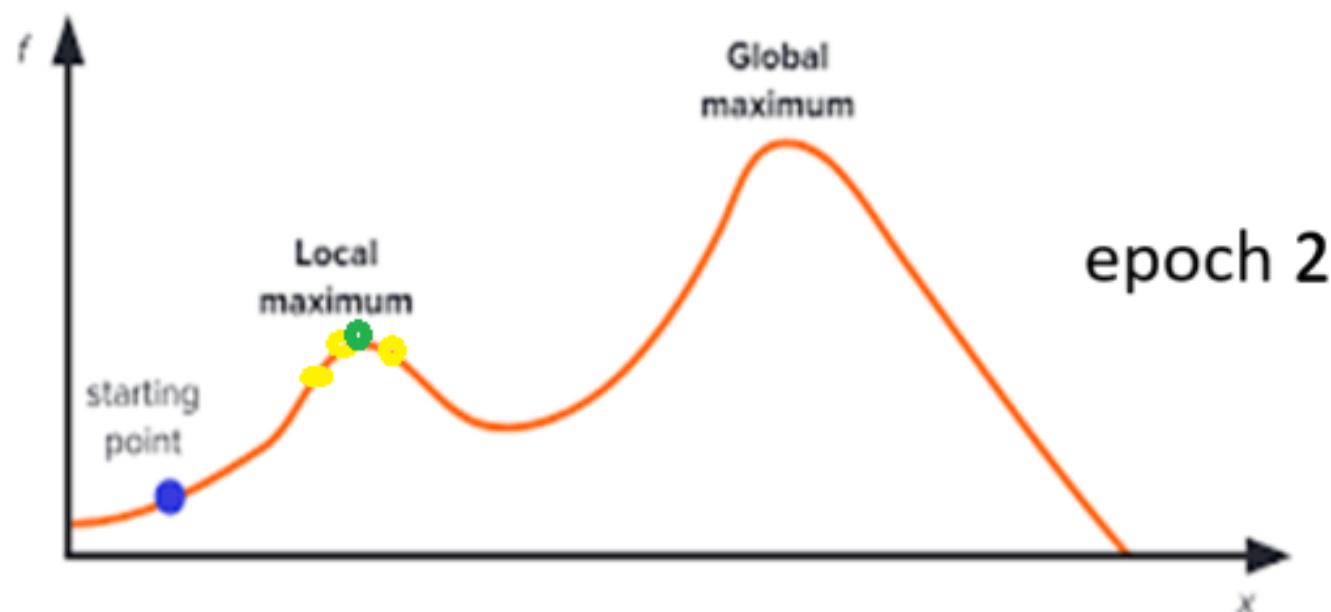
Problem spaces
with multiple
peaks. How do
we avoid 'low'
peaks when hill
climbing?



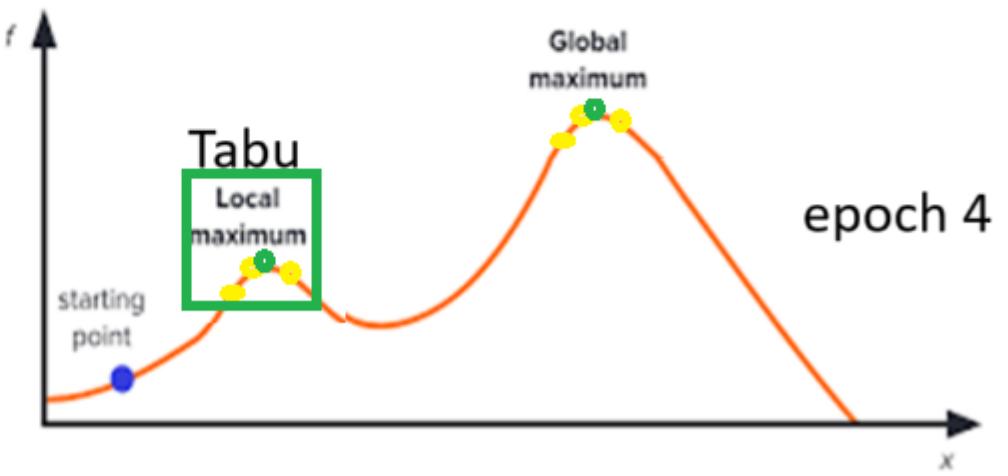
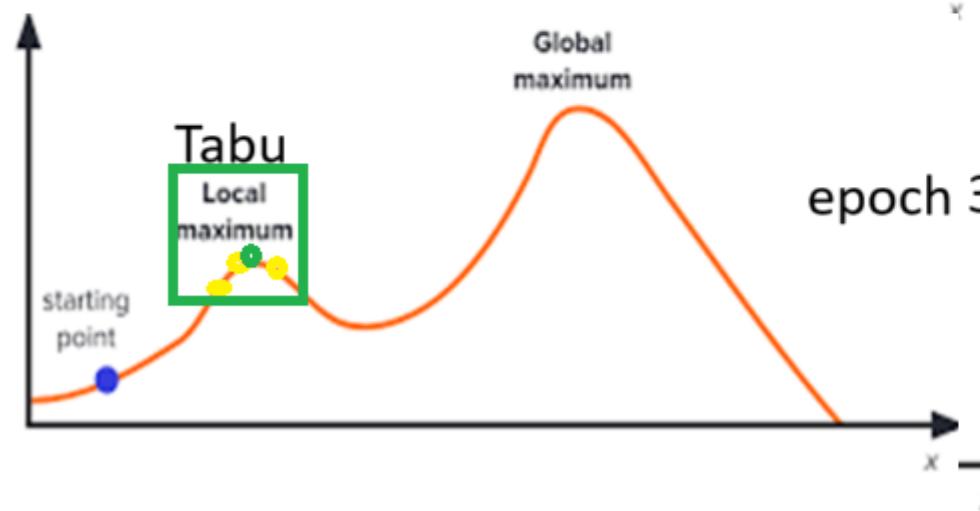
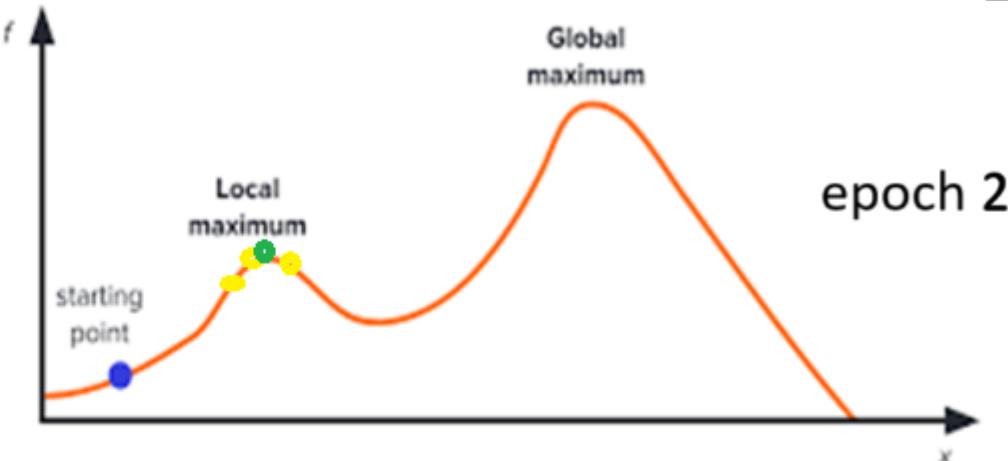
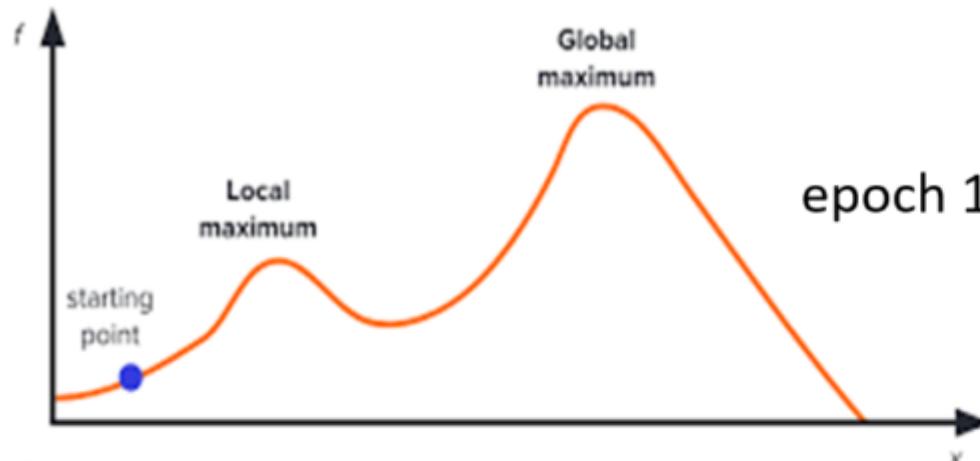
Tabu search

Once a certain area in and space has been explored, don't return to that area for a specified time.

Help escape over exploring a local maxima



Tabu search in 4 steps



Back Tracking

Backtracking

In many cases, a backtracking algorithm amounts to a clever implementation of exhaustive search, with generally unfavourable performance.

This is not always the case, however, and even so, in some cases, the savings over a brute-force exhaustive search can be significant.

Performance is, of course, relative: an $O(n^2)$ algorithm for sorting is pretty bad, but an $O(n^5)$ algorithm for the traveling salesman (or any NP-complete*) problem would be a landmark result.

*no efficient way to solve them above brute force, but solutions can be checked

Backtracking

A practical example of a backtracking algorithm is the problem of arranging furniture in a new house.



Furniture example

There are many possibilities to try, but typically only a few are actually considered. Starting with no arrangement, each piece of furniture is placed in some part of the room.

If all the furniture is placed and the owner is happy, then the algorithm terminates.

If we reach a point where all subsequent placement of furniture is undesirable, we have to undo the last step and try an alternative.

Backtracking

Of course, this might force another undo, and so forth.

If we find that we undo all possible first steps, then there is no placement of furniture that is satisfactory. Otherwise, we eventually terminate with a satisfactory arrangement.

Notice that although this algorithm is essentially brute force, it does not try all possibilities directly. For instance, arrangements that consider placing the sofa in the kitchen are never tried.

Many other bad arrangements are discarded early, because an undesirable subset of the arrangement is detected. The elimination of a large group of possibilities in one step is known as ***pruning***.

Another abstracted Backtracking example - Sudoku

- Finds an empty cell
- Tries numbers 1-9 in that cell
- Checks if each number is valid (row, column, and 3x3 box)
- If valid, places number and recursively solves rest of puzzle
- If solution fails, backtracks by setting cell back to 0
- Continues until puzzle is solved or all possibilities are exhausted

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

Sudoku

A program would solve a puzzle by placing the digit "1" in the first cell and checking if it is allowed to be there.

If there are no violations (checking row, column, and box constraints) then the algorithm advances to the next cell and places a "1" in that cell.

When checking for violations, if it is discovered that the "1" is not allowed, the value is advanced to "2".

If a cell is discovered where none of the 9 digits is allowed, then the algorithm leaves that cell blank and moves back to the previous cell. The value in that cell is then incremented by one.

This is repeated until the allowed value in the last (81st) cell is discovered.

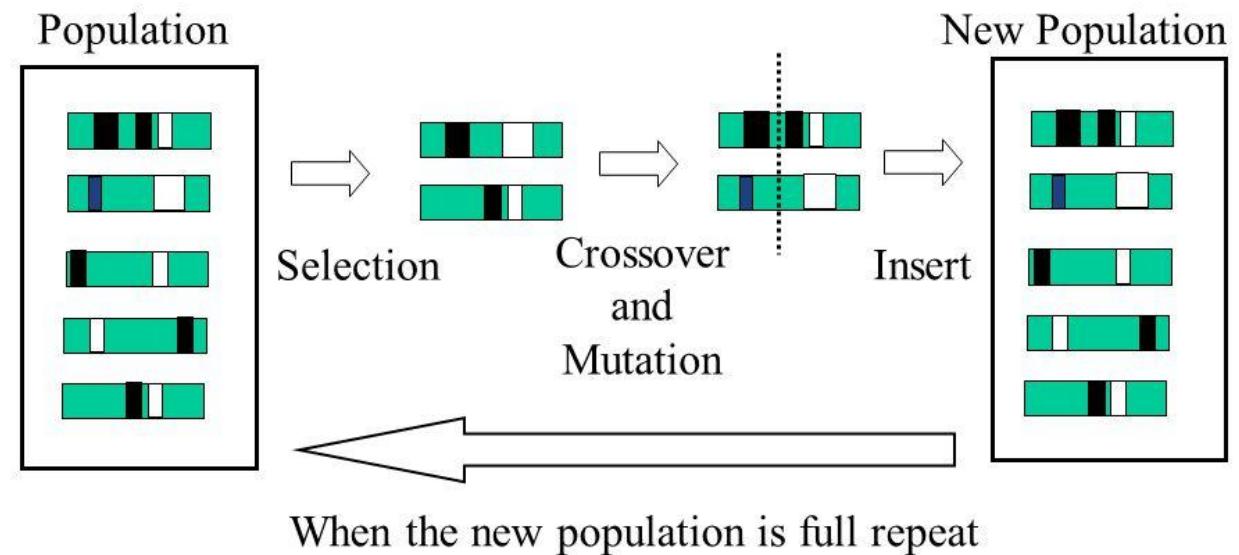
Genetic Algorithms

Genetic Algorithms are optimization techniques inspired by natural evolution, used to find solutions to complex problems

By 'complex' we mean not solvable, in acceptable time, using brute force

Mimic processes like selection, crossover, and mutation.

Generational Algorithm



Solving (simple) Sudoku with GAs

Rules, numbers 1-6, No repeats in rows, columns or boxes

Start with your base case

Generate a population of possible solutions

Select the 'best' 2

Change some values (mutation)

Crossover some values

Repeat until a new population

Try solving this simpler version GA...then
Brute force then backtracking then ?????

Solution

References

<https://textbooks.cs.ksu.edu/cc310/4-data-structures-and-algorithms/12-brute-force/>

[https://en.wikipedia.org/wiki/Sudoku solving algorithms](https://en.wikipedia.org/wiki/Sudoku_solving_algorithms)

<https://www.codecademy.com/learn/learn-data-structures-and-algorithms-with-python/modules/brute-force-algorithms/cheatsheet>

<https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/>

Amanur Rahman Saiyed, “The Traveling Salesman problem”, Indiana State University, 2012. <http://cs.indstate.edu/~zeeshan/aman.pdf>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. 4th edition. MIT press.