Metaheuristics: Single State Local Search Methods

Fernando Lobo

Universidade do Algarve

Stochastic Local Search

- Almost all metaheuristics make use of randomized choices
- This gives rise to a family of algorithms known as *Stochastic Local Search* (SLS) methods
- Stochastic is just a fancy name for randomized
- Note: making randomized choices doesn't mean the algorithm is blind and searches completely at random

Stochastic Local Search

- SLS methods often use of the notion of *locality*
- Locality is induced by the neighbourhood of solutions
- That is, SLS tend to search around the vicinity of solution(s) that the algorithm maintain(s) as its current state
  - Single-state SLS methods only maintain one solution as its current state
  - Multi-state SLS methods maintain several solutions as its current state (they are also called *population-based search methods*)
- We shall study both types of algorithms

Single-state SLS methods

Let's start with the most basic methods:

- Blind search
  - Uninformed Random Walk
  - Uninformed Random Picking
- Hillclimbing
  - Steepest Ascent (Descent)
  - Next Ascent (Descent)

Blind Search

- Starting from an initial solution, the algorithm wanders around the search space and simply keeps the best solution found so far
- Two variations:
  - Uninformed Random Walk
    - At each step the algorithm moves from a solution s to a solution in the neighbourhood of s
  - Uninformed Random Picking
    - At each step the algorithm moves from a solution s to any other solution in the search space
- Performs poorly

Uninformed Random Walk

$s =$ solution generated uniformly at random (u.a.r.)
$v_s = f(s)$
$best = (s, v_s)$
while *not terminate()* **do**
    $x =$ choose solution u.a.r. from $\mathcal{N}(s)$
    $v_x = f(x)$
    if $v_x > v_s$ **then**
        $best = (x, v_x)$
    end
    $s = x$
end
return *best*

Uninformed Random Picking

$s =$ solution generated uniformly at random (u.a.r.)
$v_s = f(s)$
$best = (s, v_s)$
while *not terminate()* **do**
    $x =$ choose solution u.a.r. from S
    $v_x = f(x)$
    if $v_x > v_s$ **then**
        $best = (x, v_x)$
    end
    $s = x$
end
return *best*

Some notes on the pseudocode

- u.a.r. stands for uniformly at random
- f denotes the evaluation function
- terminate() is a boolean predicate
  - could be based on an allowed maximum number of iterations
  - a certain amount of time
  - or some other criterion
- algorithm assumes we are dealing with a maximization problem
  - for minimization, change $>$ to $<$

Hillclimbing

- Maintain a current solution
- At each time step, move to a neighbour that is better than the current solution
- If no such neighbour exist, terminate
  - we have reached a local optimum

Hillclimbing

- Two common variations:
  - *Steepest Ascent*
    - visit the entire neighbourhood and move to the best neighbour (with ties broken u.a.r.)
  - *Next Ascent*
    - visit the neighbourhood in a random order, and move to the first neighbour that improves upon the current solution
- Steepest Ascent requires the evaluation of the entire neighbourhood in every search step
- Because of that, Next Ascent is often much more efficient than Steepest Ascent

Steepest Ascent Hillclimbing

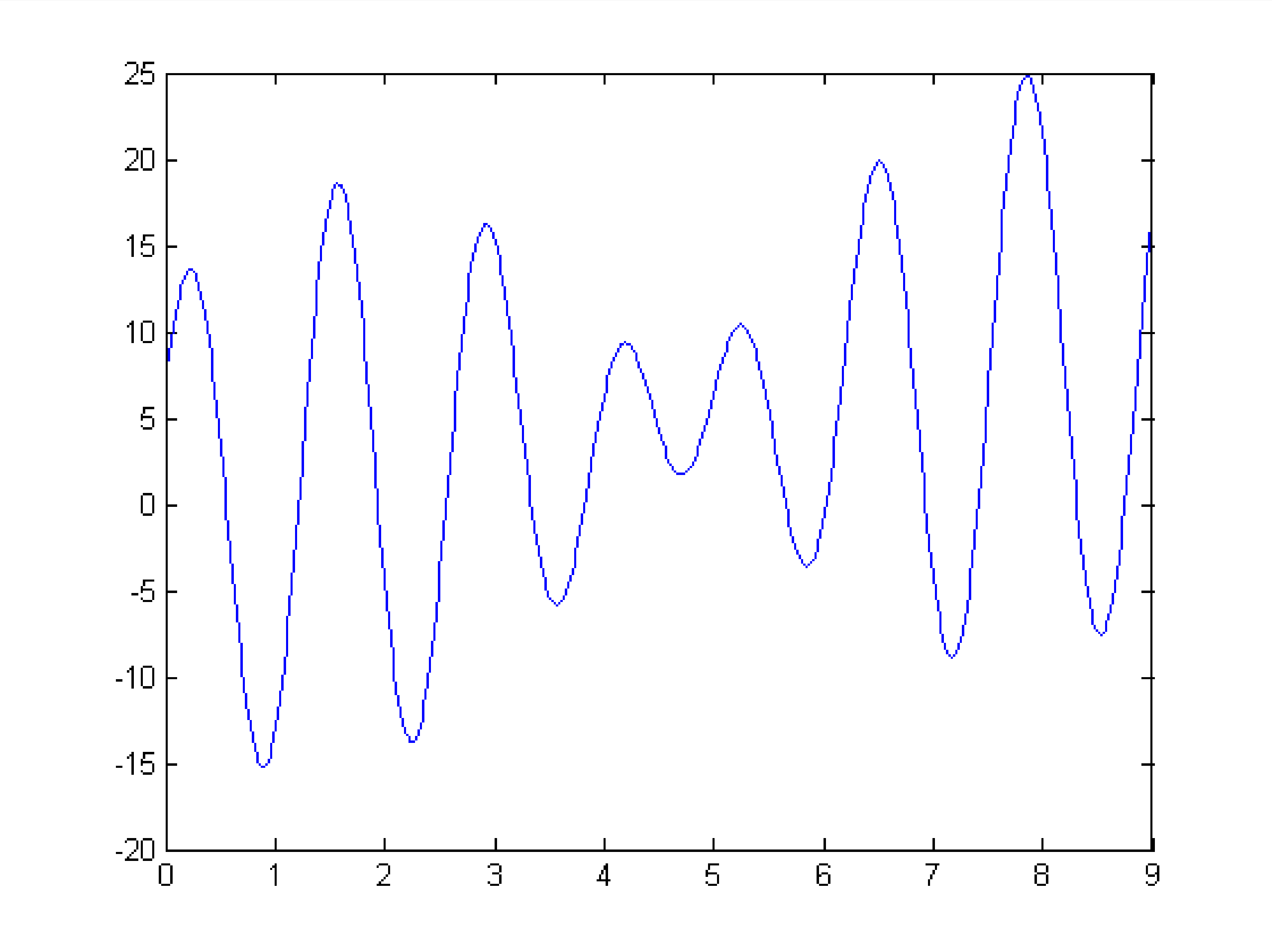
```
s = solution generated u.a.r.
v_s = f(s)
repeat
    best_improvement = 0
    foreach x ∈ N(s) do
        v_x = f(x)
        if v_x − v_s > best_improvement then
            best_improvement = v_x − v_s
            best_neighbour = (x, v_x)
        end
    end
    if best_improvement > 0 then
        (s, v_s) = best_neighbour
    end
until best_improvement = 0
return (s, v_s)
```

Next Ascent Hillclimbing

---

s = solution generated u.a.r.
$v_s = f(s)$
repeat
  improve = false
  foreach $x \in \mathcal{N}(s)$ **do**
    $v_x = f(x)$
    if $v_x > v_s$ **then**
      improve = true
      $(s, v_s) = (x, v_x)$
      break
    end
  end
until *not improve*
return (s, v_s)

A 1-dimensional multimodal function



A 2-dimensional multimodal function

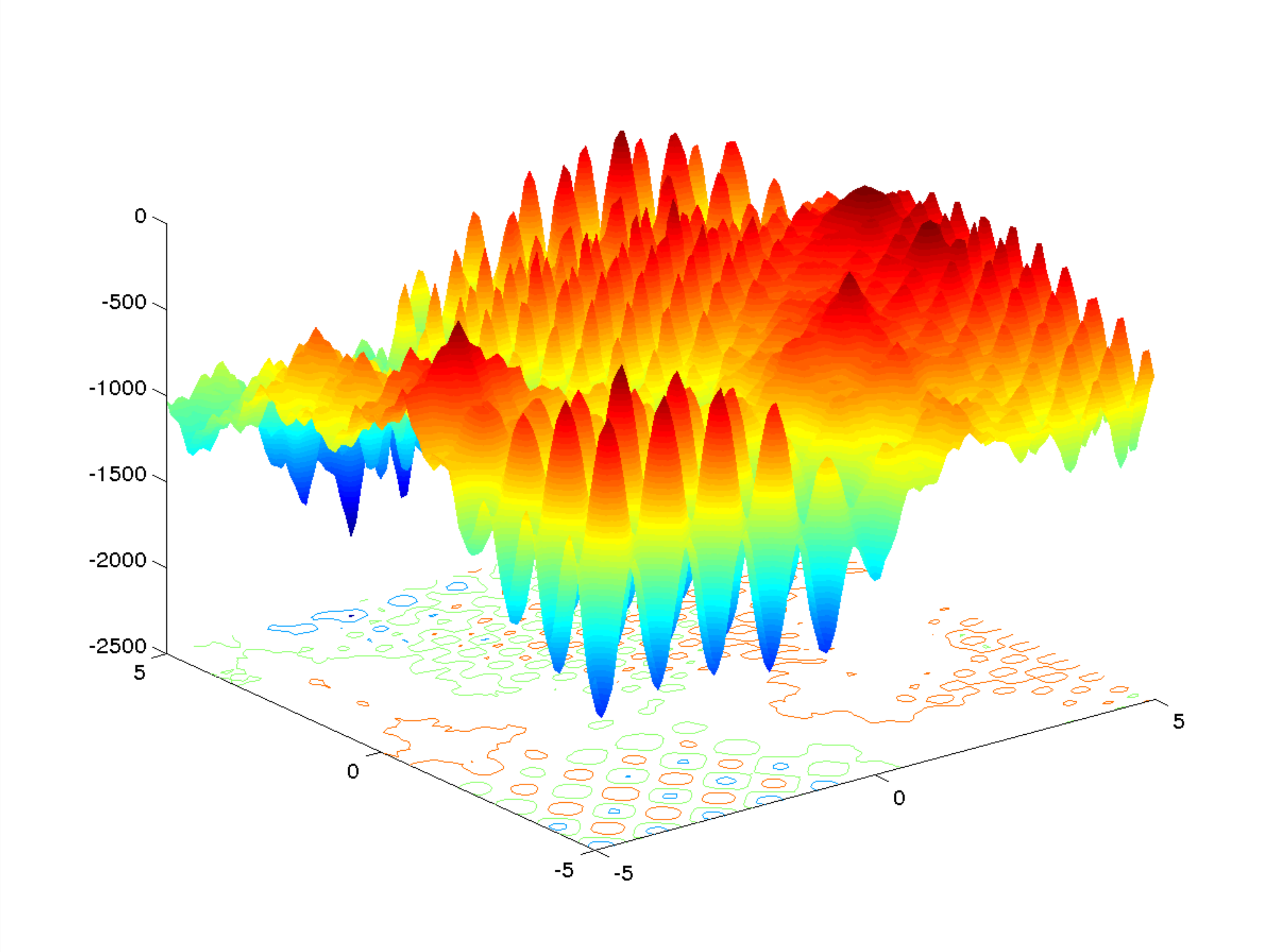


Figure taken from the GECCO 2020 Competition on Niching Methods for Multimodal Optimization website located at
`https://www.epitropakis.co.uk/gecco2020/`

Multi-start Hillclimbing

- Drawback of hillclimbing: getting stuck at a local optimum
- Quick fix: once a local optimum is reached, restart hillclimbing again
- Hopefully we will reach another local optimum
- We can iterate this over and over, obtaining a collection of local optima
- The resulting algorithm is called *Multi-start Hillclimbing*

Another strategy to escape a local optimum

- Use larger neighbourhoods
- SAT example: 2-bit Hamming distance neighbourhood, instead of 1-bit
- The resulting local optimum would be better than any of its 2-bit neighbours
- But this has a cost:
  - time to explore a larger neighbourhood increases
  - in theory, we could make the neighbourhood so large that its size becomes of the order of the size of the search space!
  - need to find a balance between time complexity and ability to escape local optima

Neighbourhood pruning

- A strategy to improve efficiency
- Reduce size of neighbourhood by excluding neighbours that cannot (or are unlikely to) result in an improvement

Neighbourhood pruning: TSP example under 2-exchange

- 2 edges are removed, and 2 edges are added
- To be an improving move, at least one of the added edges has to have a smaller distance (weight) than a removed edge
  - Why? Because the contribution of the other edges are held the same in both solutions
- We can use this knowledge and avoid visiting (guaranteed) non-improving neighbours
- We can even build an appropriate data structure to support this

Neighbourhood pruning: TSP example under 2-exchange (candidate lists)

Candidate Lists:

- For each vertex store a list of neighbouring vertices sorted according to edge weight
- These are called *candidate lists*
- Building complete candidate lists requires $O(n^2)$ memory and $O(n^2 \log n)$ time
- To avoid this, the size of the candidate lists is often bounded by a constant value: 10 to 40

Variable Neighbourhood Ascent

- Idea: when stuck at a local optimum, switch to a larger neighbourhood
- Use k neighbourhoods $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_k$ in increasing order of size
- When an improvement is made using neighbourhood $\mathcal{N}_j$, we go back to using $\mathcal{N}_1$
- At the end, the resulting solution will be a local optimum with respect to all these neighbourhoods