Introduction to Artificial Intelligence

Local Search Algorithms and Optimization Problems

Ing. Tomas Borovicka

Department of Theoretical Computer Science (KTI), Faculty of Information Technology (FIT)
Czech Technical University in Prague (CVUT)

BIE-ZUM, LS 2014/15, 4. lecture

https://edux.fit.cvut.cz/courses/BIE-ZUM/

Summary of Previous Lecture

- We introduced cost on actions in the state space.
- We focused on algorithms searching the shortest path.
- **Dijkstra's algorithm**
  - An algorithm for finding the shortest path between two vertices in a graph.
  - Complete and optimal.
  - Exponential complexity.
- We introduce heuristics and learnt how to use additional information about the problem to search for a solution more effectively.

Informed Search Algorithms:

- Greedy Search
  - Algorithm always expands the node with the best (lowest) heuristic value.
  - Incomplete and sub-optimal.
- A^* algorithm
  - Combination of greedy search and Dijkstra, takes into account the heuristic and the path cost.
  - Complete and optimal.

Local Search

- State space search algorithms systematically explore the state space.
- For many problems path to the goal is irrelevant and only what matters is the goal itself.
- Local search algorithms operate using a single current configuration (candidate solution) and search in the neighborhood for better configuration to move in.
- Uses very little memory.
- Can find a solution in large or infinite state space.
- Typically used for **optimization problems**, where the goal is to find the best state according to some **objective function**.
- There is not a goal function or path cost as we defined for ”classical” search strategies.

Optimization Problem

Optimization problem

Let $f(s)$ be an objective function

$$f : \mathbb{R}^n \rightarrow \mathbb{R},$$

optimization problem is a problem of finding s^* such that

$$s^* = \arg\max_{s \in S}(f(s)),$$

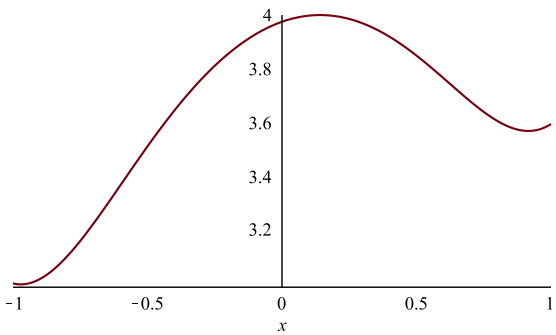
i.e. searching the configuration that maximize the objective function.

Problems of minimization and maximization of objective function are treated equally since

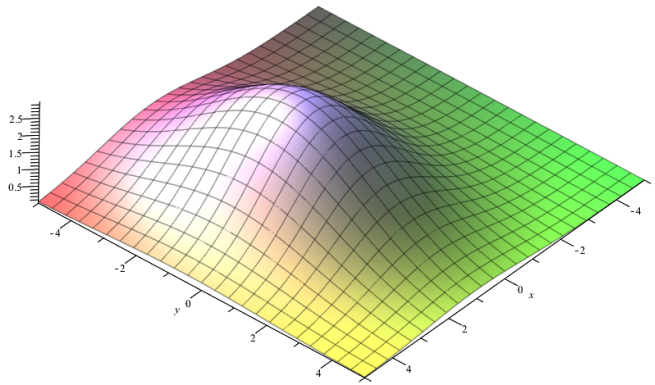
$$\arg\max_{\mathbf{x} \in X}(-f(\mathbf{x})) = \arg\min_{\mathbf{x} \in X}(f(\mathbf{x})).$$

State Space Landscape

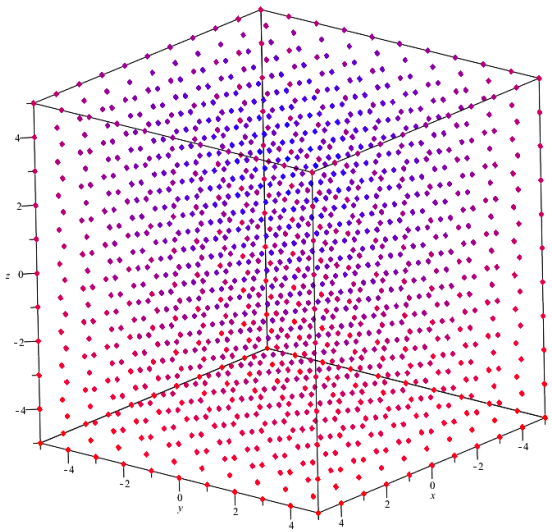
complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function



1D space



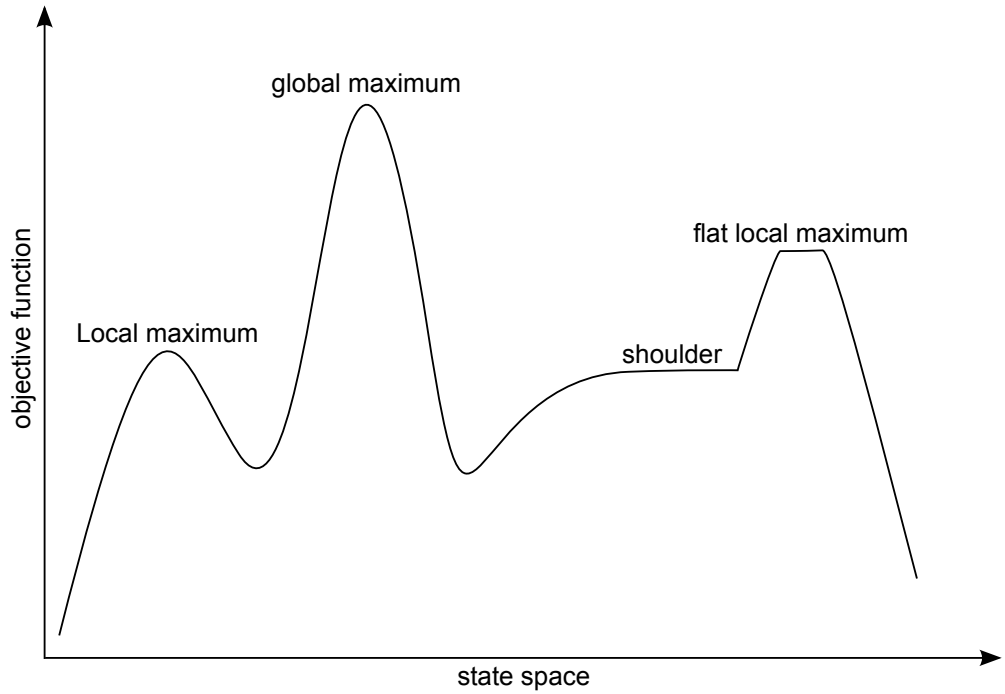
2D space



3D space

Objective Function

- state s^* is a **global maximum** if $\forall s \in S : f(s^*) \geq f(s)$,
- if $f(s^*) \geq f(s)$ only $\forall s$ in some neighborhood of s^* we call it a **local maximum**,
- **Plateau** is a "flat" area P where $\forall s \in P : f(s) = const$, if there is a neighboring state s^* where $f(s^*) > f(s)$ we call it **shoulder**.



Local Search Algorithm

- The idea is to find δs such that $f(s + \delta s) \geq f(s)$.
- Iteratively updates $s_{n+1} = s_n + \sigma s$
- Reduces the problem to a series of 1D line searches.

1. Starts at a random configuration.
2. Repeatedly considers various moves
   - accepts some,
   - rejects some,
3. Restarts when it gets stuck.

Local Search Pseudo-code

- We usually do not have any prior knowledge about the shape of the objective function, therefore we can not use standard mathematical analysis methods to find stationary points.

We use following iterative search strategy:

Algorithm 1 General iterative optimization

1: $\mathbf{x} \leftarrow$ randomly generated initial state
2: **while** $\mathbf{x}$is not good enough $\wedge$ runs too long **do**
3: $\quad \mathbf{y} \leftarrow$ new candidate state
4: $\quad$ **if** $f(\mathbf{y}) > f(\mathbf{x})$ **then**
5: $\quad\quad \mathbf{x} \leftarrow \mathbf{y}$
6: $\quad$ **end if**
7: **end while**
8: **return** $\mathbf{x}$

Brute Force Optimization

- Systematically samples all possible solutions and returns the state that maximizes the objective function.
- Feasible only for small problems, since the number of possible states increases exponentially with the number of dimensions (for continuous variables the number of possible states is infinite).

Brute Force Optimization Pseudocode

Algorithm 2 Brute-force optimization on $\langle 0, 10\rangle \times \langle 0, 10\rangle \times \ldots \times \langle 0, 10\rangle$

x $\leftarrow (0, 0, \ldots, 0)$
for all $y_1 \in \{0, 0.1, 0.2, \ldots, 9.9, 10\}$ **do**
  for all $y_2 \in \{0, 0.1, 0.2, \ldots, 9.9, 10\}$ **do**
    $\vdots$
    for all $y_n \in \{0, 0.1, 0.2, \ldots, 9.9, 10\}$ **do**
      if $f((y_1, y_2, \ldots, y_n)) > f(\mathbf{x})$ **then**
        x $\leftarrow (y_1, y_2, \ldots, y_n)$
      end if
    end for
    $\vdots$
  end for
end for

Random Optimization

- In each iteration algorithm randomly (or according to some distribution surrounding to current position) samples new candidate solution and moves if the solution is better.

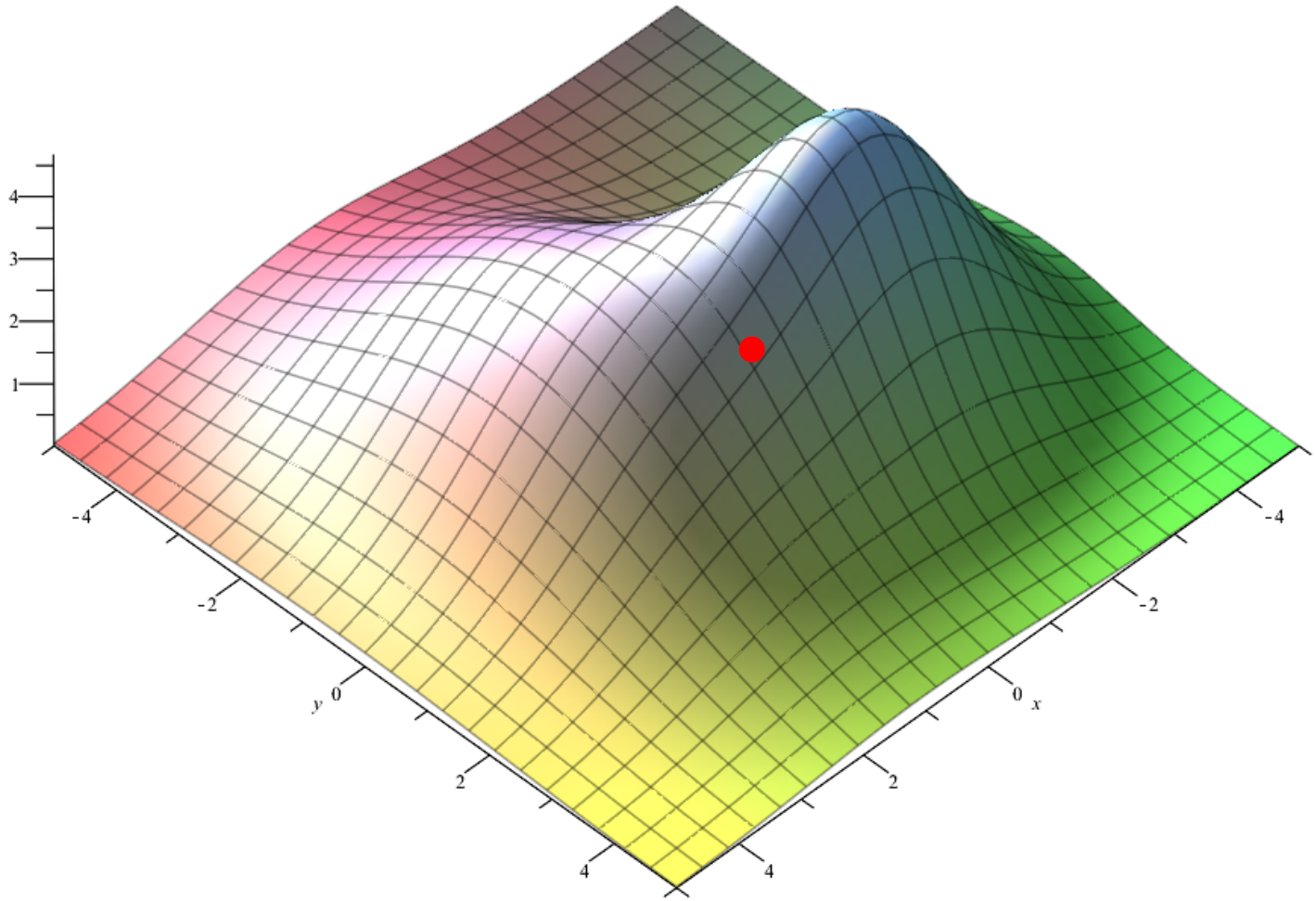
Algorithm 3 Random optimization on $\langle 0, 10\rangle \times \langle 0, 10\rangle \times \ldots \times \langle 0, 10\rangle$

$\mathbf{x} \leftarrow (0, 0, \ldots, 0)$
for $i \leftarrow 1 \ldots max\_steps$ **do**
  $\mathbf{y} \leftarrow (\text{random}(0, 10), \text{random}(0, 10), \ldots, \text{random}(0, 10))$
  if $f(\mathbf{y}) > f(\mathbf{x})$ **then**
    $\mathbf{x} \leftarrow \mathbf{y}$
  end if
end for
return $\mathbf{x}$

Hill-climbing

- Randomly samples candidate solutions in the current state's neighborhood and moves to better solution.
  - **First-choice hill climbing** - generates candidates randomly until one that is better than the current state is found.
  - **Stochastic climbing** - generates several candidates and from those that are better then the current state randomly choose one to move in.
- It continually moves in the direction of increasing objective function (i.e. uphill) and terminates in the local maximum.

Hill-climbing Example



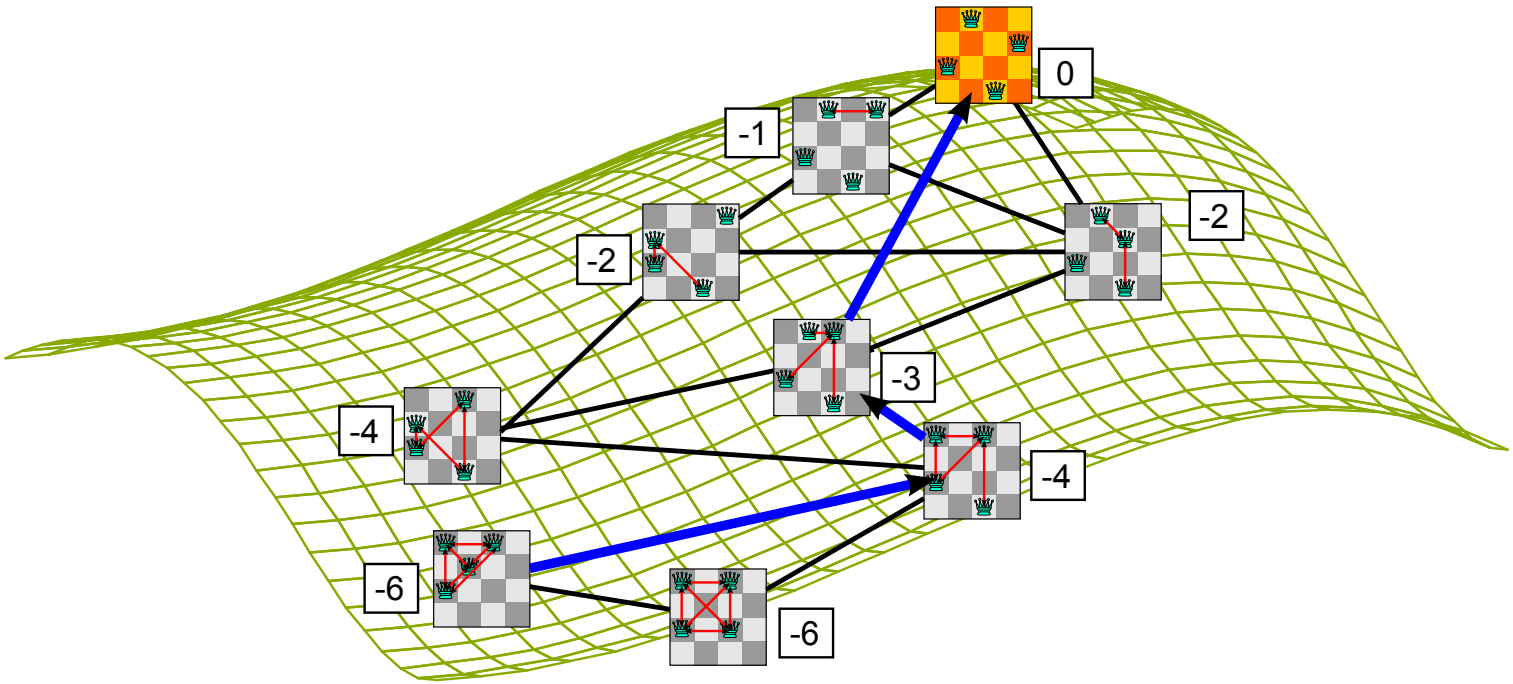
Hill-climbing: Pseudocode

Algorithm 4 Hill climbing

```
1: x ← random_state()
2: i ← 0
3: while ¬good_enough(x) ∧ i < max_iter do
4:     y ← random_neighbor(x)
5:     if f(y) > f(x) then
6:         x ← y
7:     end if
8:     i ← i + 1
9: end while
10: return x
```

Hill-climbing: Example of N Queens

- The example demonstrates that iterative local search can be used to solve even a non-numerical problem.



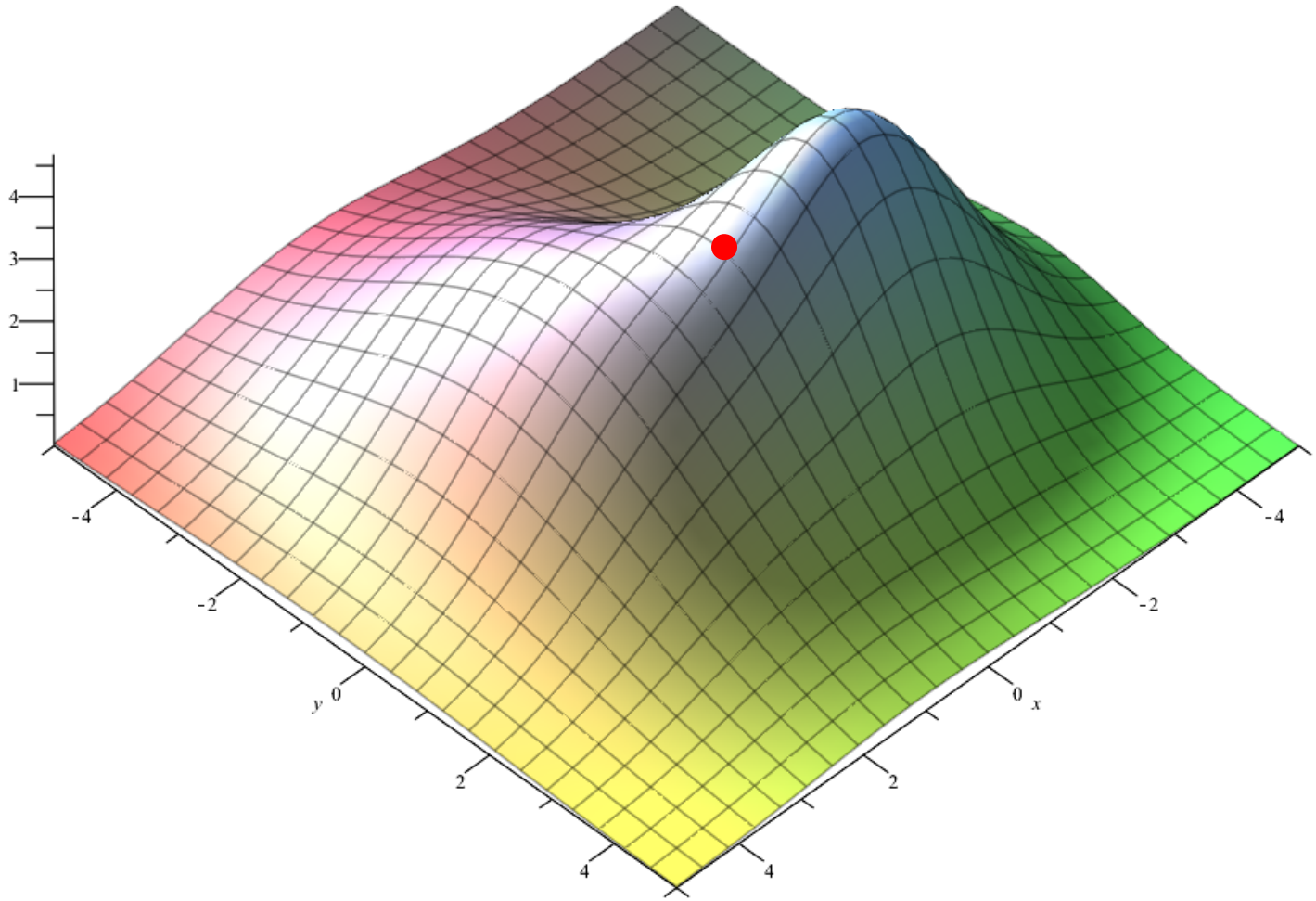
Steepest Ascent Hill-climbing

- Sometimes called greedy local search.
- It systematically samples several candidate solutions in current state's neighborhood and moves to the solution that maximizes objective function.
- Converges faster than random hill-climbing.

Algorithm 5 Steepest ascent Hill-climbing

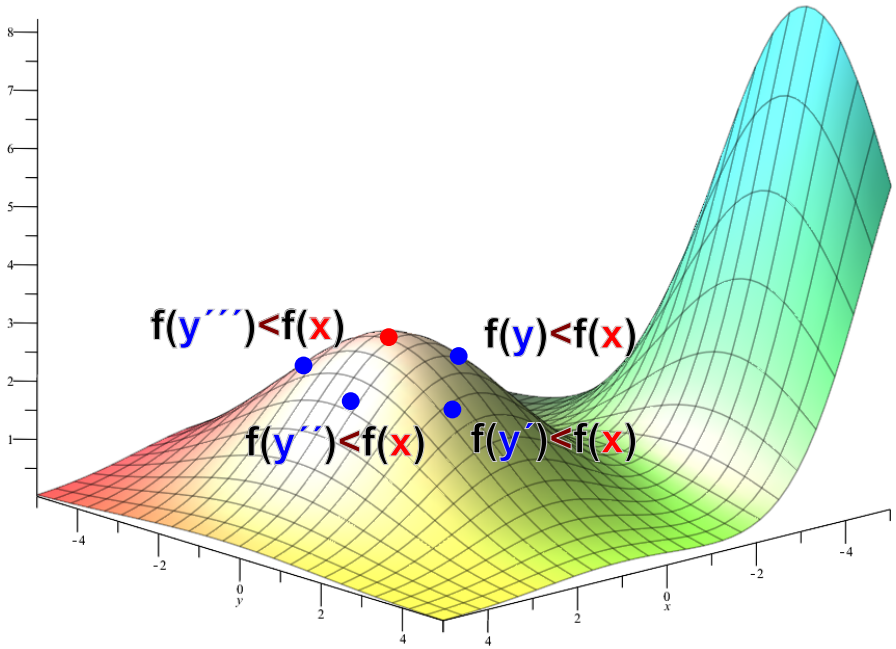
```
1: x ← random_state()
2: i ← 0
3: while ¬good_enough(x) ∧ i < max_iter do
4:     y ← random_neighbor(x)
5:     for i = 2, 3, . . . , k do
6:         z ← random_neighbor(x)
7:         if f(z) > f(y) then
8:             y ← z
9:         end if
10:    end for
11:    if f(y) > f(x) then
12:        x ← y
13:    end if
14:    i ← i + 1
15: end while
```

Steepest Ascent Hill-climbing Example



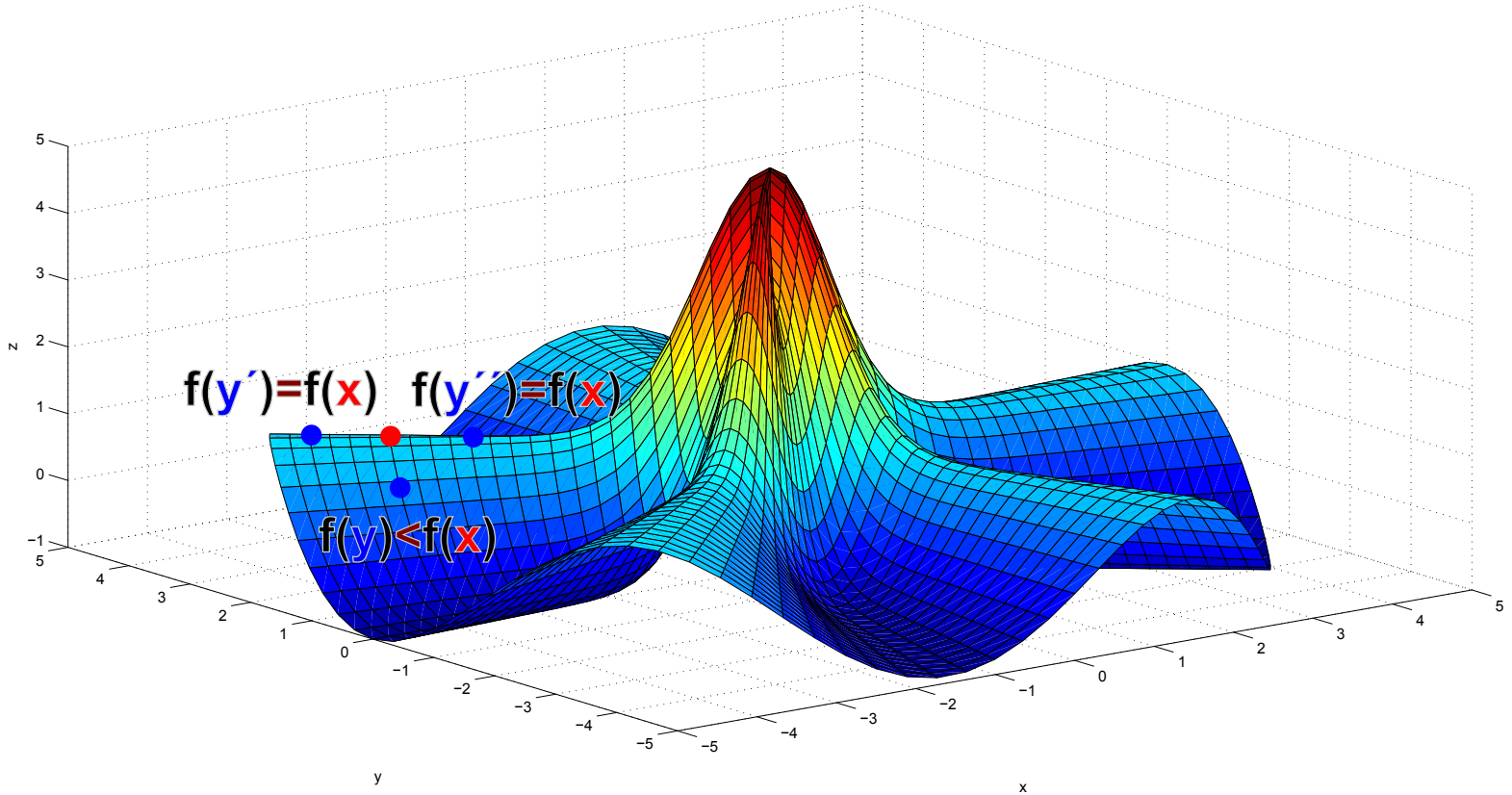
Hill-climbing: Local Extrems

- Any move in a neighborhood of local maximum makes current situation worse.
- Algorithm is stuck with nowhere to go.
- Necessary to temporarily accept worse solutions...



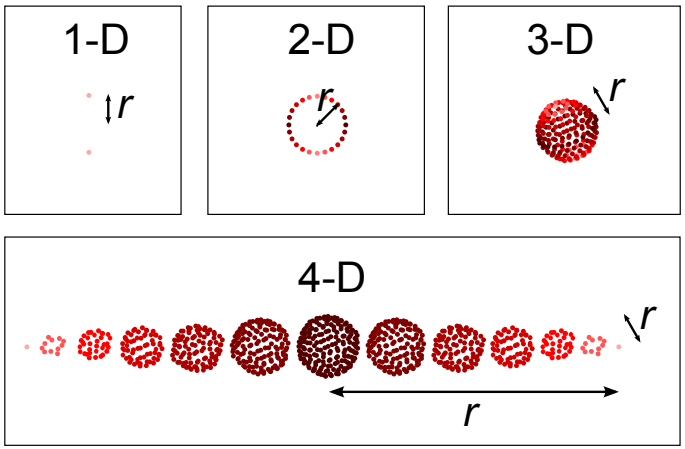
Hill-climbing: Plateau

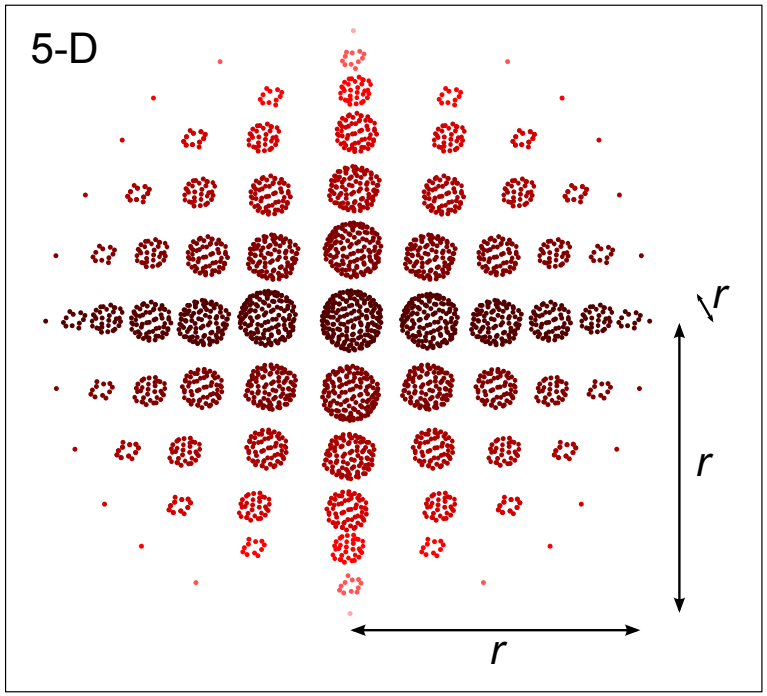
- Any move in a neighborhood does not make current situation better.
- I ta can be local maximum or shoulder, but where and how far to go?
- We can allow sideways, but we should limit the number of moves to not get stuck.



Hill climbing: Curse of Dimensionality

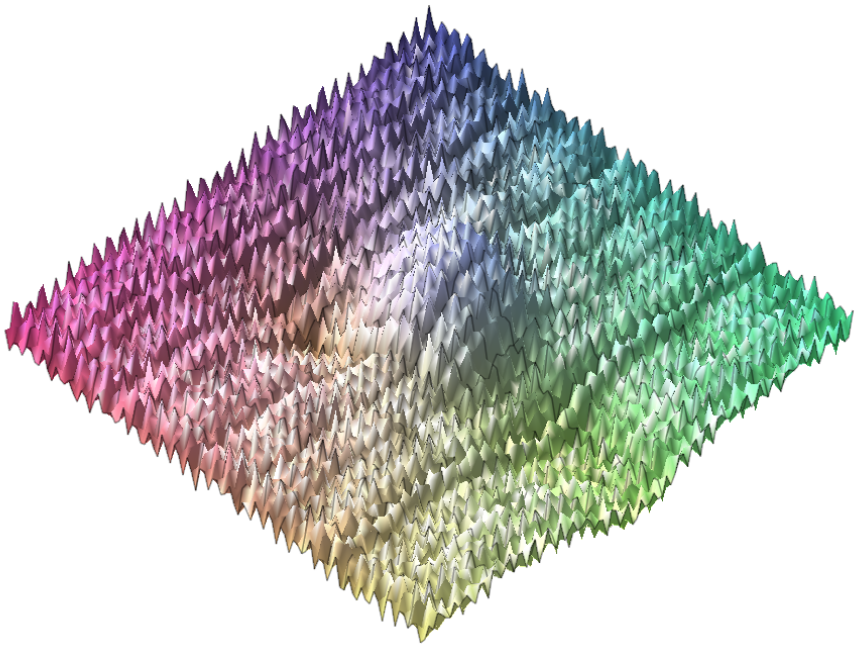
- Adding extra dimensions to Euclidean space increases the volume of the space exponentially.
- I.e. number of states grows exponentially with number of features.
- Sampling the neighborhood of some state becomes too difficult.





Size of Neighborhood

- In practice, most of the objective functions are noisy and with many local optimums.



What size of neighborhood?

- Too small - we may get stuck in local optimum.
- Too big - we may miss the optimum.

Simulated Annealing - Motivation

- In metallurgy, 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 low energy crystalline state.
- If you let metal cool rapidly, its atoms aren't given a chance to settle into a tight lattice and are frozen in a random configuration, resulting in brittle metal. If we decrease the temperature very slowly, the atoms are given enough time to settle into a strong crystal.

Simulated Annealing

- Combination of random hill-climbing search and Metropolis algorithm.
  - The idea derived from Metropolis algorithm allows to move with certain probability to worse than current solution.

1. Sample new candidate from neighborhood of current state (random hill-climbing).
2.
   - If $f(s') > f(s)$ move to the state.
   - If $f(s') \leq f(s)$ move to the state with probability proportional to:
     - ★ "badness" of the move i.e. $\delta f = (f(s') - f(s))$,
     - ★ time, i.e. actual temperature T.
3. When enough iterations have passed without improvement, terminate.

- Typical probability function decreases exponentially with δf:

$$P(s, s', T) = e^{\frac{f(s') - f(s)}{T}}.$$

Temperature Schedule

- Temperature is the only feature that varies during the calculation, therefore temperature schedule is one of the most important features in Simulated Annealing.
- Actual temperature influence behavior of the search
  - high T: probability of ”locally bad” move is higher (Random Walk),
  - low T: probability of ”locally bad” move is lower (Stochastic Hill-Climbing)
- Annealing procedure repeatedly lowering the temperature until the system converges.

Exponential schedule

$$T(t) = T_0 \alpha^t$$

Linear schedule

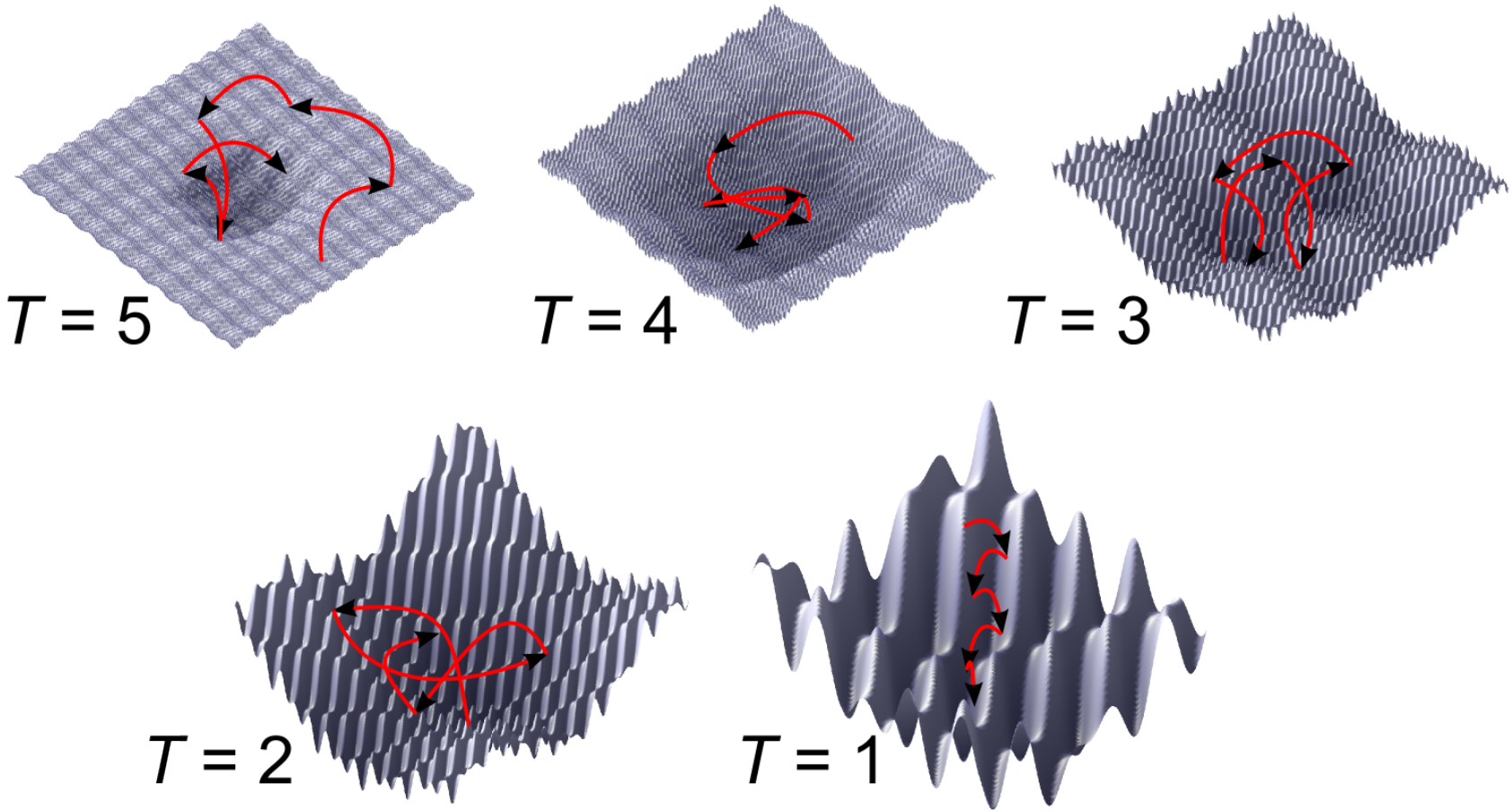
$$T(t) = T_0 - \eta t$$

Logarithmic schedule

$$T(t) = \frac{c}{log(t + d)}$$

Simulated Annealing Example

In this example the objective function is minimized (it is more appropriate for understanding of Simulated Annealing).



Simulated Annealing Pseudocode

Algorithm 6 Simulated annealing

```
1:  x ← random_state()
2:  t ← high_number
3:  while t ≥ 0 do
4:      y ← random_neighbor(x)
5:      if f(y) > f(x) then
6:          x ← y
7:      else if P(f(x), f(y), t) ≥ random(⟨0, 1⟩) then
8:          x ← y
9:      end if
10:     t ← decrease()
11: end while
12: return x
```

Tabu Search

- Prevent returning quickly to the same state.
- Uses memory structures to maintain a history and prevent recent states being revisited.
- A neighborhood is constructed to identify adjacent solutions. Typically we use some similarity measures such as
  - euclidean or cosine distance for vectors in $\mathbb{R}^n$,
  - hamming distance for vectors in $\{0, 1\}^n$,
  - structural similarity for graphs, trees, ...
- The simplest approach of tabu search is to keep "tabu list", fixed length queue of recent candidate solutions. In each iteration adds recent candidate to queue and drops the oldest one if the list is full. Move to currently tabu'ed state and it's neighborhood is not allowed.
- However, the effective tabu strategy depends on the particular problem domain and variety of approaches exists.

Tabu Search Example