Fitness Landscape/ Random & Local Search

Shin Yoo
School of Computing, KAIST

Recap

- We need three key elements for SBSE
  - Representation: how we express candidate solutions for storage
  - Fitness Function: how we compare candidate solutions for selection
  - Operators: how we modify candidate solutions for trial-and-error

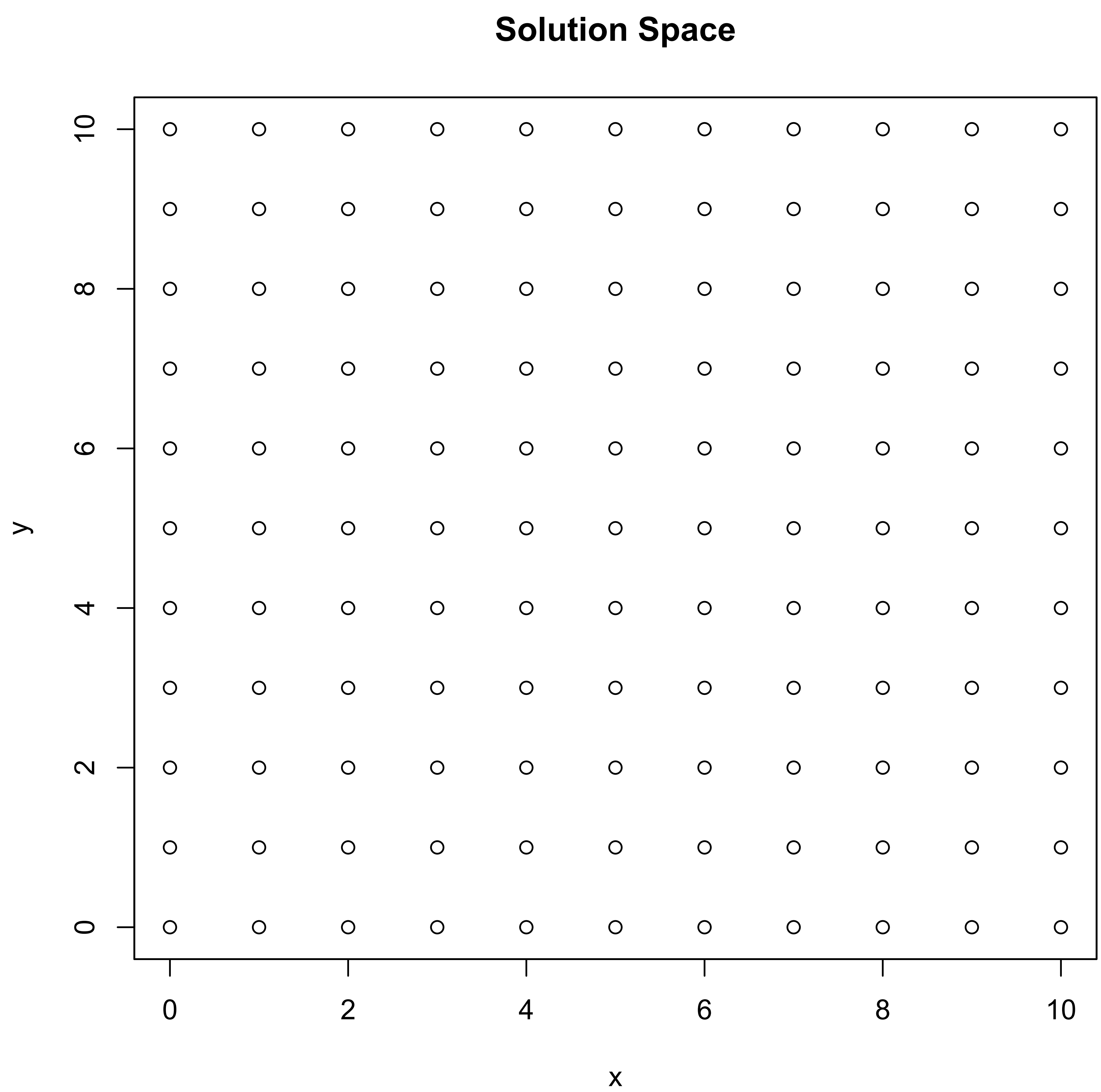
Fitness Landscape

- A spatial view of the search: there is no guarantee that the **actual** optimisation you are working on can be easily visualised spatially. However, this visual analogy is a useful tool when discussing the distribution of the fitness across possible solutions.

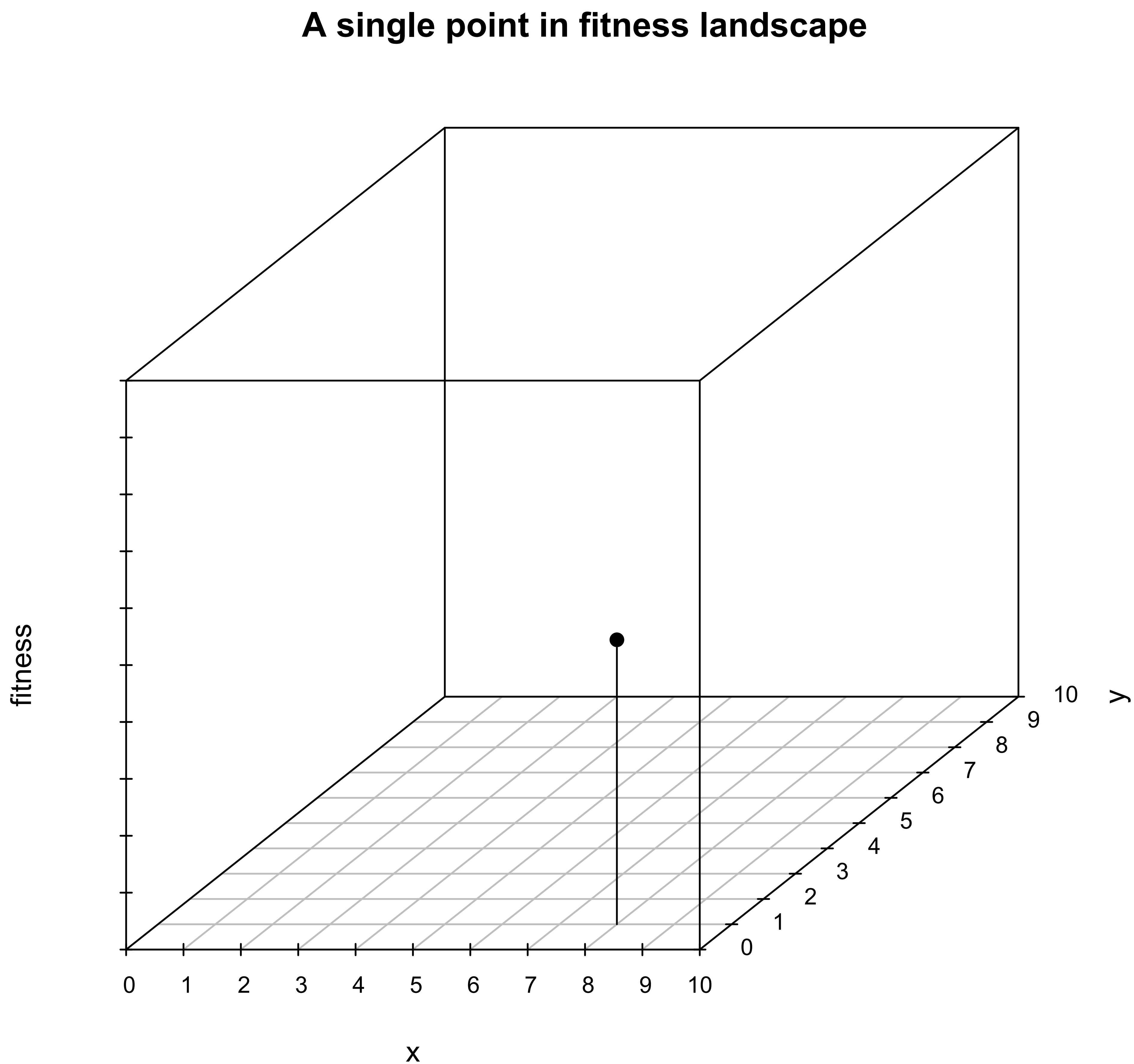
- Given a solution space S (a hyperplane), and a fitness function F, a fitness landscape is a hyper dimensional surface that represents $F: S \rightarrow \mathbb{R}$

Fitness Landscape

- Let's use a fake problem:
  - Given $0 \leq x \leq 10$, $0 \leq y \leq 10$, find (x, y) such that $x + y = 10$.



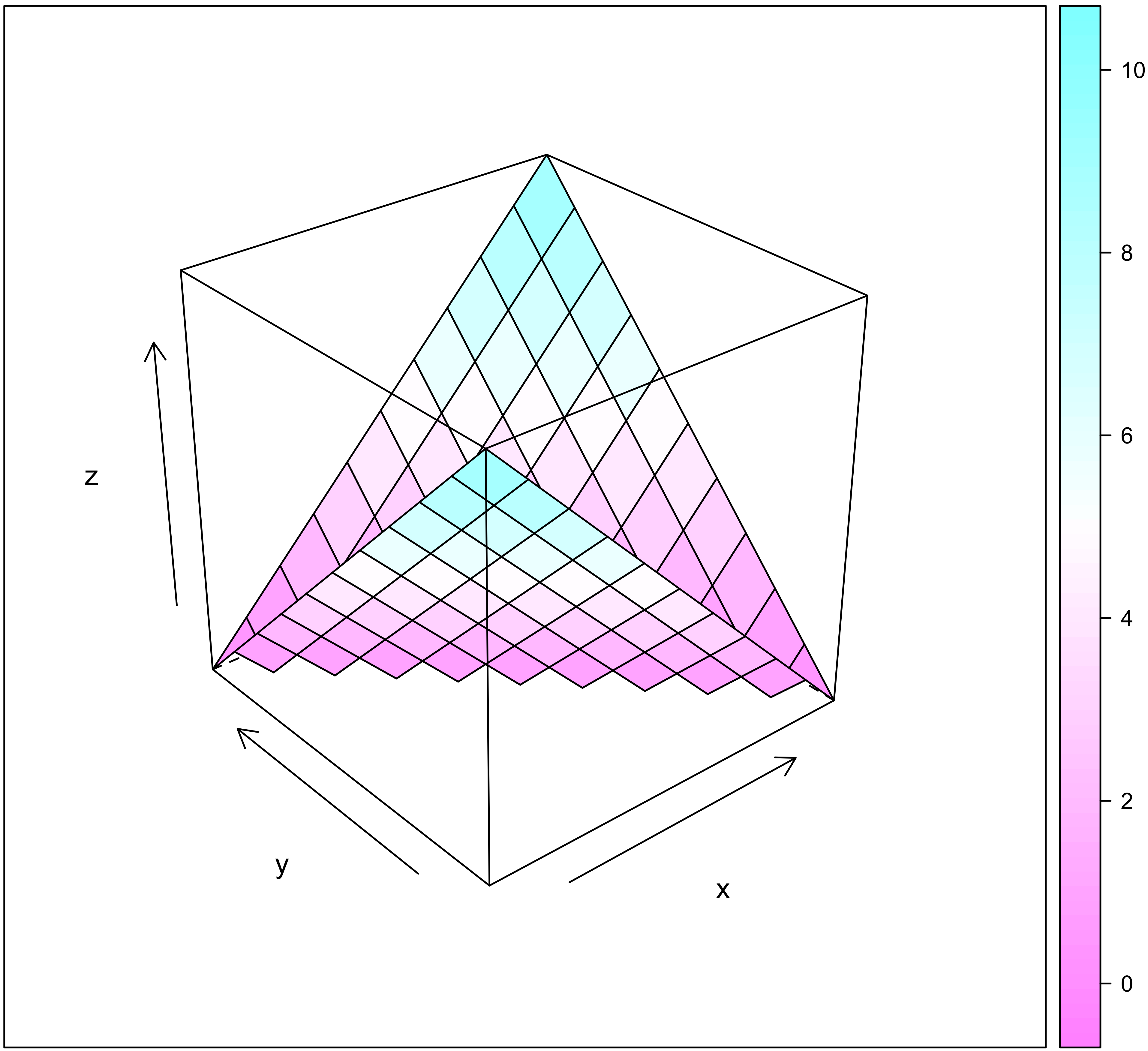
Solution Space
y
x
0
2
4
6
8
10
0
2
4
6
8
10



A single point in fitness landscape
fitness
x
y
0
1
2
3
4
5
6
7
8
9
10

Fitness for Fake Problem

- Given `(x, y)`, **how far** are we from **solving the problem**?
  - We solve the problem when `x + y == 10`
  - If the current sum of x and y are s, we are `|10 - s|` far away from solving the solution
  - `f(x, y) = |10 - (x + y)|`
  - Minimise the above function until it becomes 0.



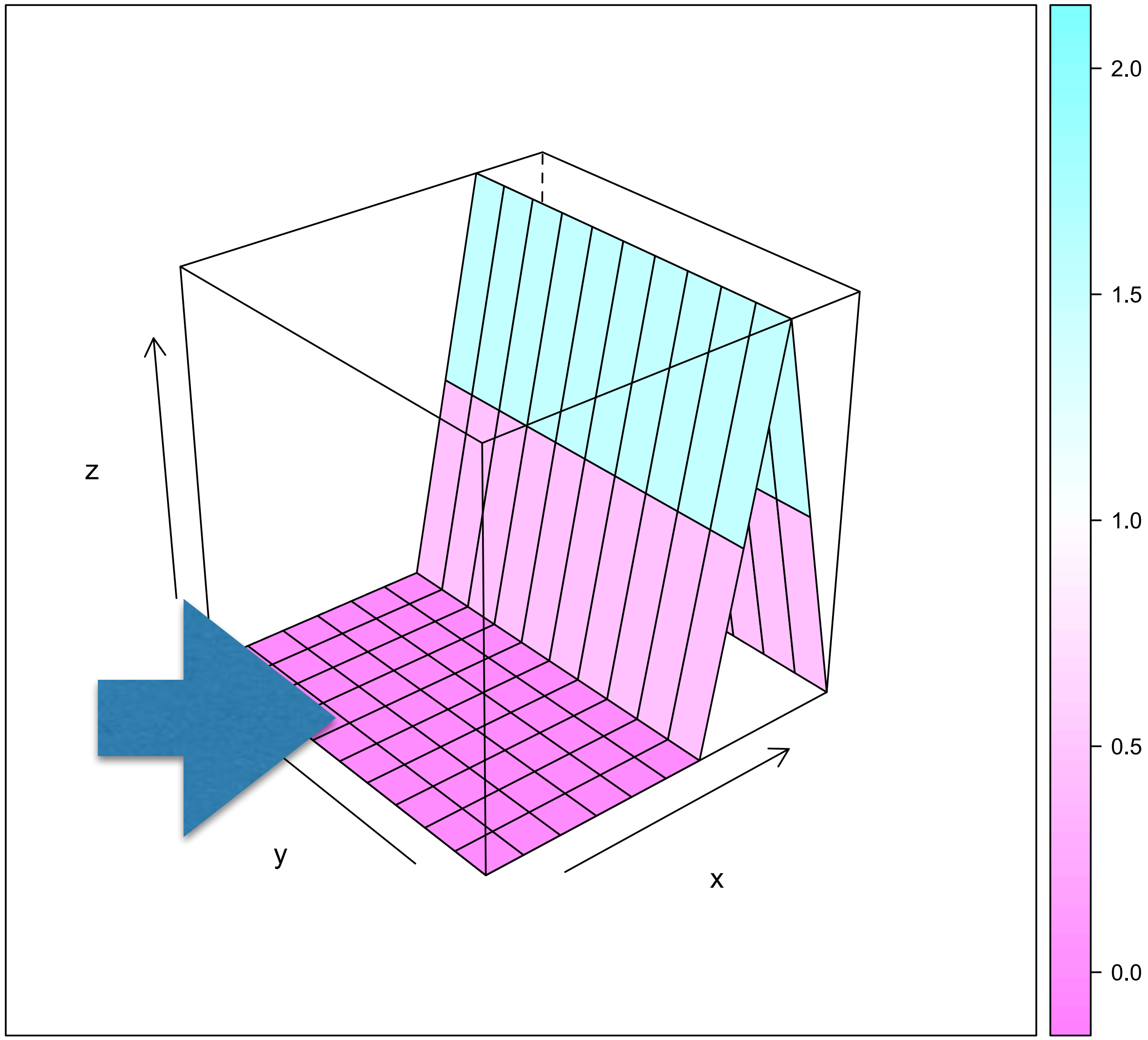
z
y
x
10
8
6
4
2
0

Properties of Landscape

- Size: small/large but also finite/(effectively) infinite
- Flatness: is there a large plateau?
- Ruggedness: how many local optima should we expect?
- Discreteness: continuous numeric, discrete numeric, combinatoric

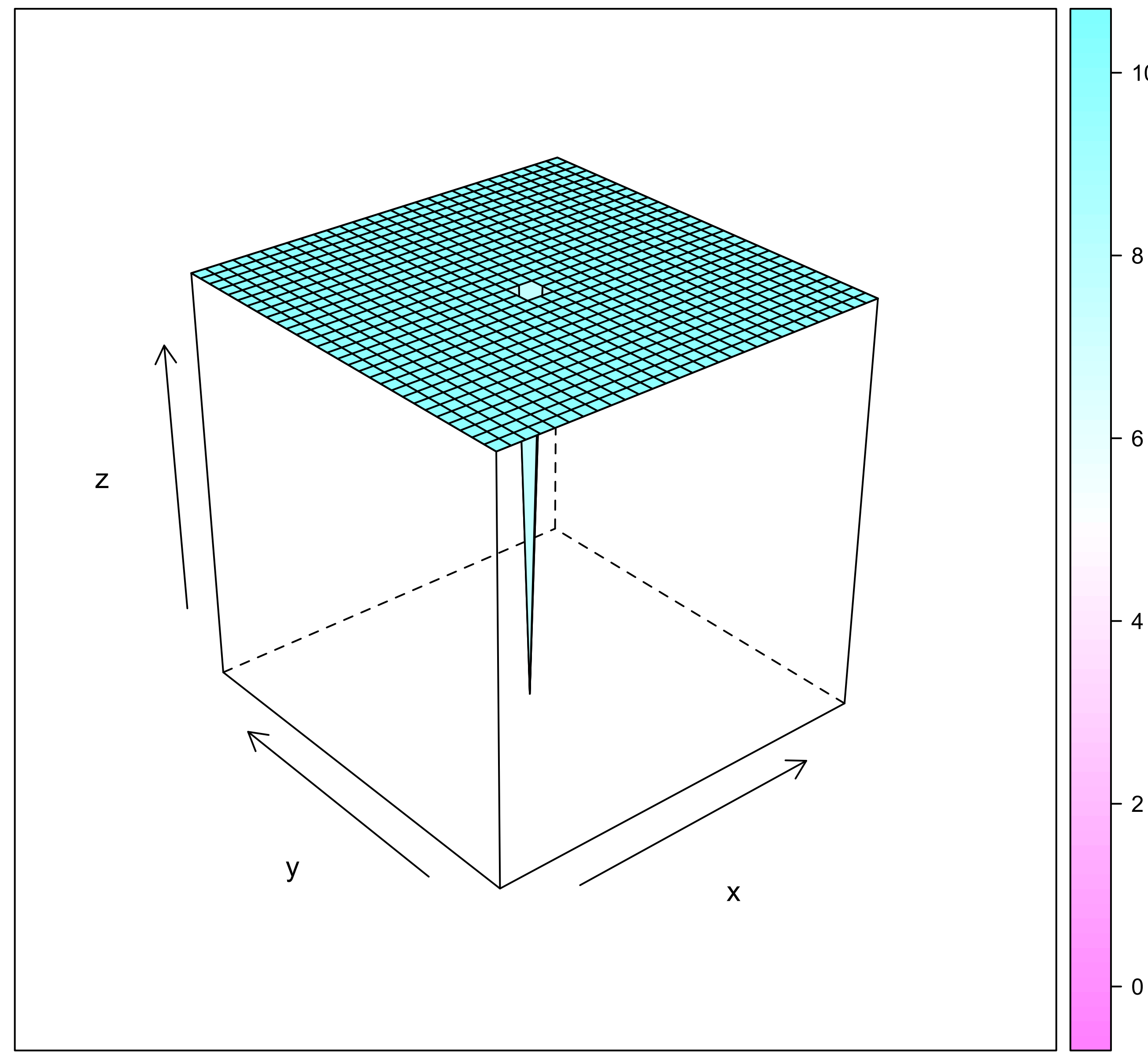
Plateau

- Large, flat region that does not exhibit any gradient.
- Suppose current solution as well as others generated by operators all fall in a plateau.
- There is no guidance; hard to escape.



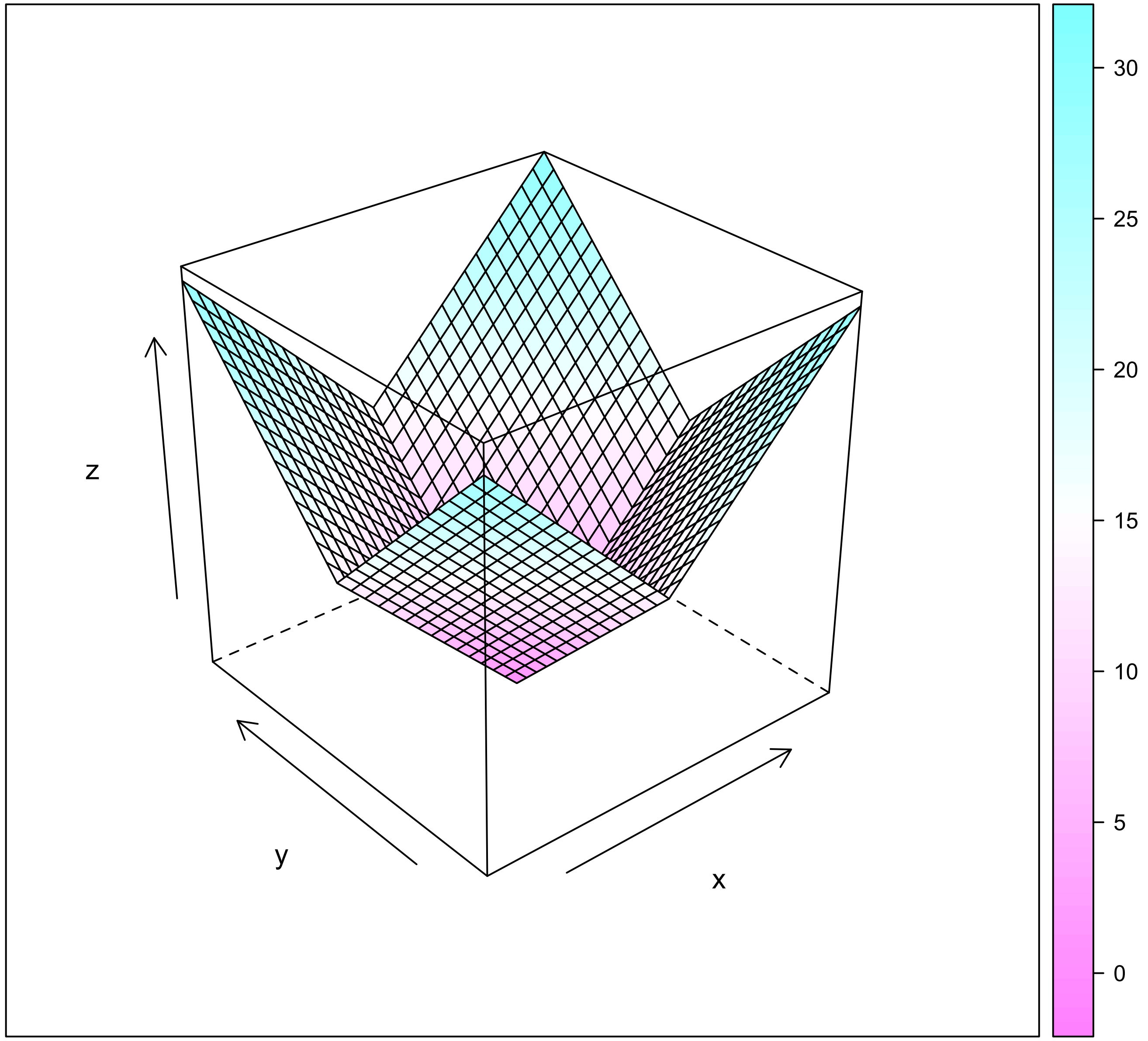
Needle in the Haystack

- Worst landscape to search.
- Can be avoided by transforming the problem and/or designing better fitness functions
- To search for (x, y) = (15, 15):
  - `f1(x, y) = (x==15 && y == 15) ? 0 : 10`



Needle in the Haystack

- `f2(x, y) = |x-15| + |y-15|`



(..later application in testing)

```
bool flag = (x == 42);
...
if(flag){
   //do some computation
   //that needs to be tested
}
```

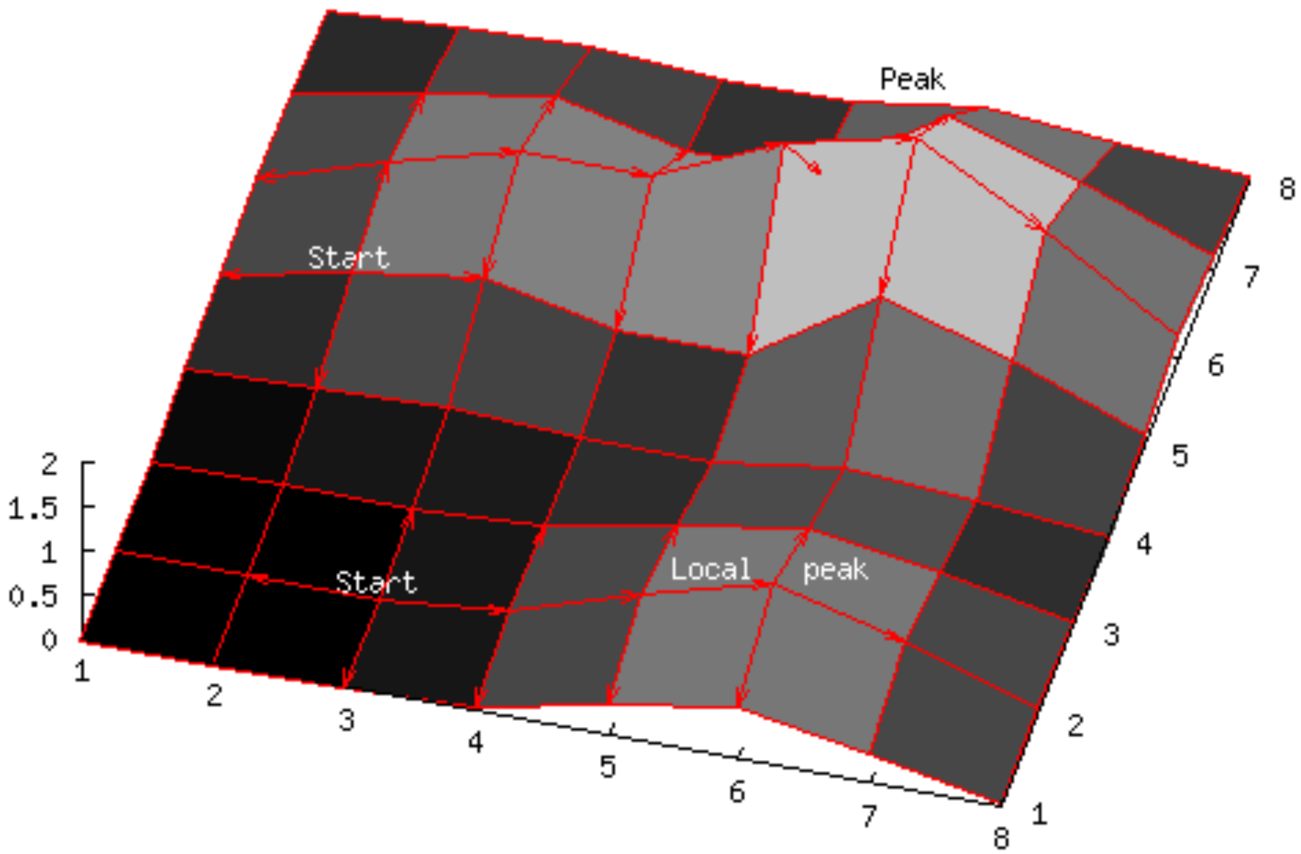
```
...
if(x == 42){
   //do some computation
   //that needs to be tested
}

...
if(|x -42| == 0){
   //do some computation
   //that needs to be tested
}
```

M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability trans- formation. IEEE Transactions on Software Engineering, 30(1):3–16, Jan. 2004.

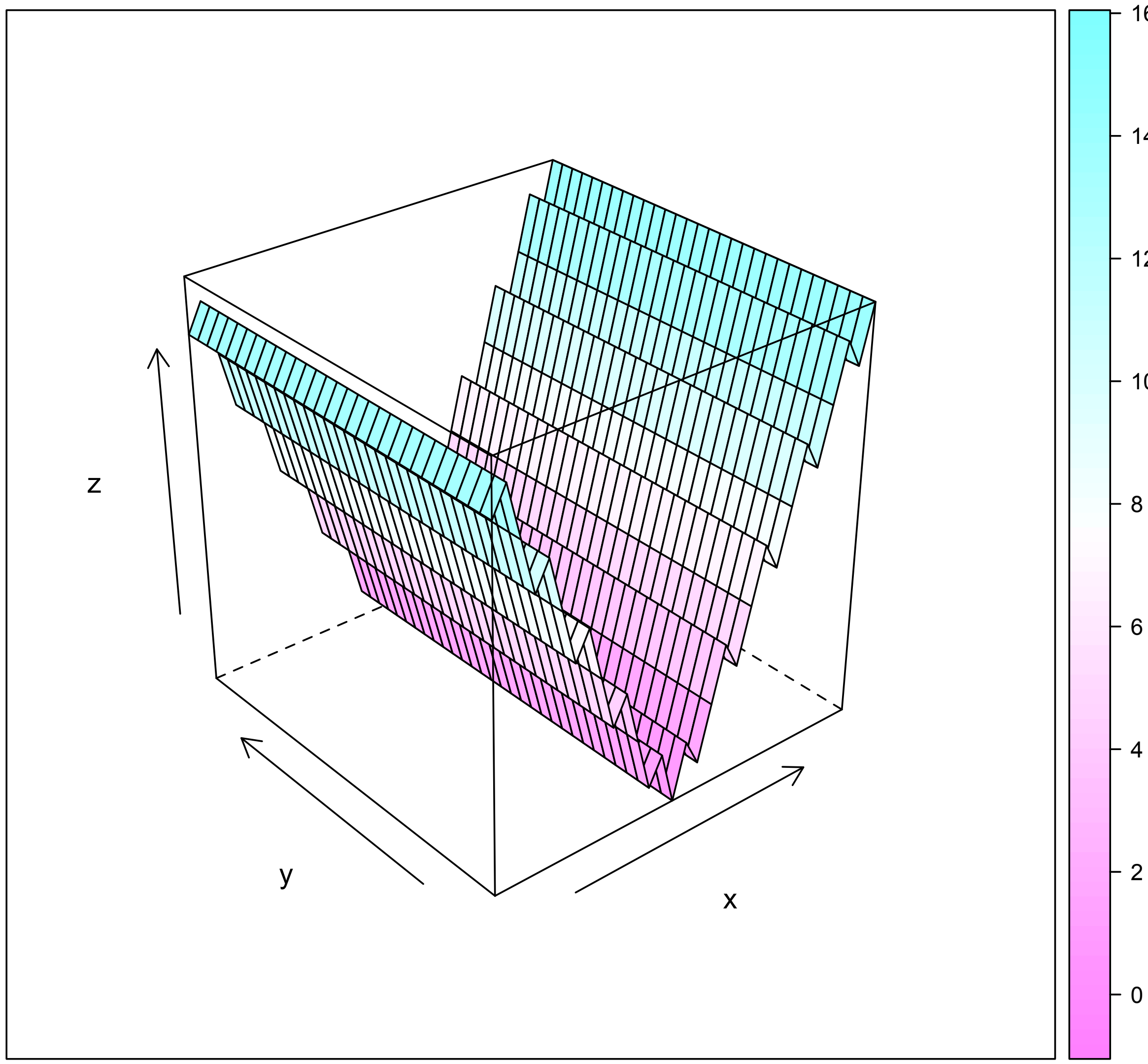
Local vs. Global Optima

- Local optima: better fitness than any surrounding region, but not the best possible fitness
- Global optima: better fitness than any other point in the landscape

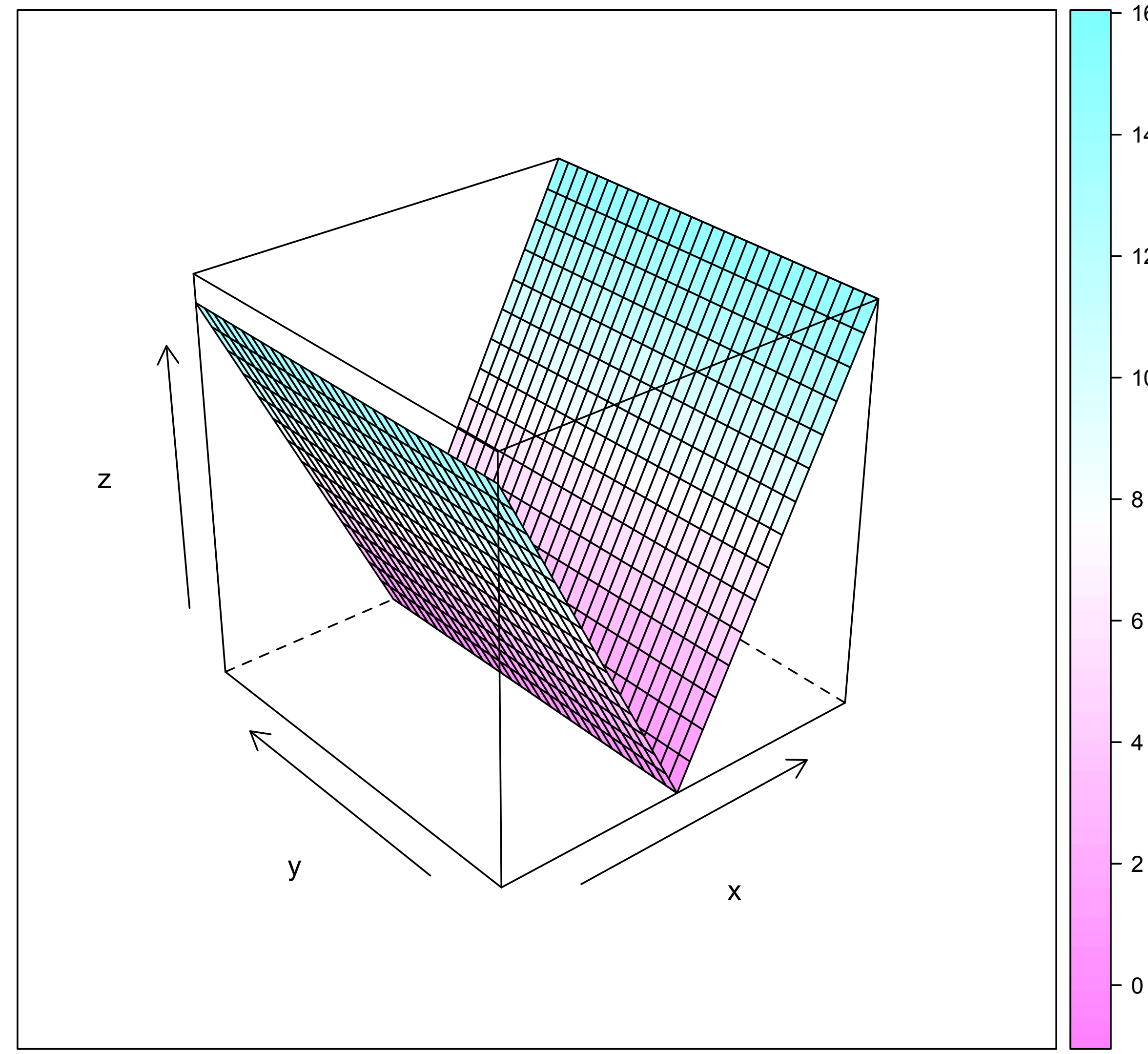


Foundation of Genetic Programmin

Ruggedness



Easy to get stuck
in one of many local optima



Smooth descent

Discrete Fitness Landscape

- In case of `(x, y) = (15, 15)`, it is (relatively) obvious what the neighbouring solutions are.
  - `(14, 15), (16, 15), (15, 14), (15, 16)`
  - `(16, 16), (14, 14), (16, 14), (14, 16)`
- What if we are searching for non-numeric solution?
  - Set membership (e.g. Do I include this requirement or not?, Do I execute this test case or not?)
  - Permutations (e.g. In which order should I execute this test suite?)
  - Highly structured data (e.g. To test this compiler, which program should I use as input?)

Summary

- A visual analogy of the relationship between solutions and fitness
- Landscape dictates how difficult the search will be, but you can influence how the landscape is constructed
- By random sampling and local random walks, you can get some feel for the shape of the landscape

Random Search

Random Search

- The polar opposite to the deterministic, examine-everything, search.
- Within the given budget, repeatedly generate a random solution, compare its fitness to the known best, and keep the best one.

Pros and Cons

- VERY easy to implement, inherently automatable, no bias at all.
- Depending on the problem, it may be extremely effective.
- No guidance at all: depending on the problem, it may take forever to obtain a meaningful solution.

Usage of Random Search

- The lack of any guidance provides two useful scenarios.
- First, random search should always be the default **sanity check** against your own search methodology: if it does not no better than random search, you are doing something wrong.

Usage of Random Search

- Somewhat ironically, random search is effective when the underlying problem does not give any guidance to begin with. For example:
    - “Search for the input to program A that will result in program crash”
    - In general, given an arbitrary program, you cannot measure the distance between the current program state and a crash!

Fuzz Testing

- Infinite Monkey Theorem: “Thousand monkeys at a thousand typewriters will eventually type out the entire works of Shakespeare"
- Basic idea: provide a stream of random input to the program, until it crashes (=our Shakespeare).
  - Either a stream of really random bits (naive), or
  - Well-formed input randomly mutated (more effective)

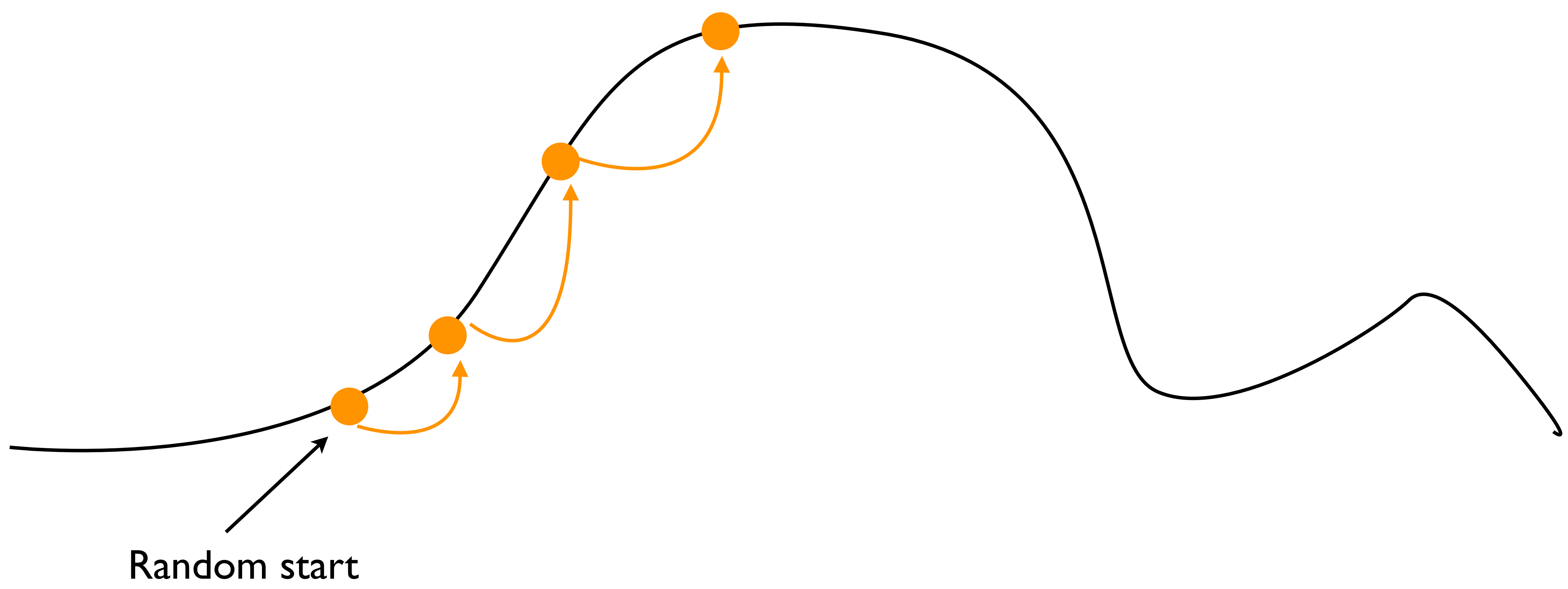
Local Search

If your problem forms a fitness **landscape**, what is optimisation?

Local Search Loop

- Start with a single, random solution
- Consider the neighbouring solutions
- Move to one of the neighbours if better
- Repeat until no neighbour is better

Local Search



Hill Climbing Algorithm

- This particular variation is also known as the steepest-ascent hill climbing.
  - Why? :)
  - What other versions are there?

```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
            return s
```

Hill Climbing Algorithm

```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
            return s
```

Steepest Ascent

```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
                break
            return s
```

First Ascent

Hill Climbing Algorithm

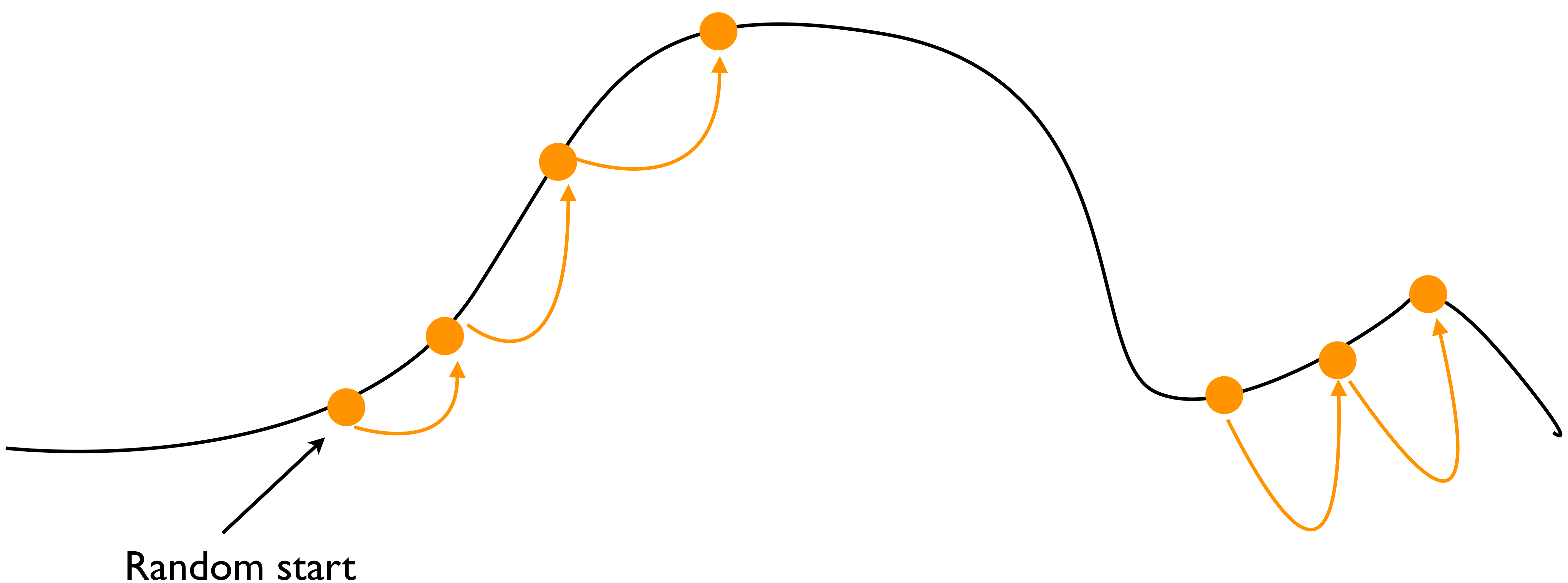
```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
                break
            return s
```

First Ascent

```
HillClimbing()
    s ← GetRandom()
    while True
        N ← GetNeighbours(s)
        N′ ← {n ∈ N | Fitness(n) > Fitness(s)}
        if |N′| > 0
            s ← RandomPick(N′)
        else
            break
    return s
```

Random Ascent

Local Search



Ascent Strategy

- Not possible to know which one is better.
- IF the current solution is near a local optimum, slowing down the ascent may (or may not) be better.
- Regardless of strategy, hill climbing monotonically climbs, until it reaches local/global optimum; it never goes <u>down</u>.

Pros/Cons

- Pros: **cheap** (fewer fitness evaluations compared to GAs), **easy** to implement, repeated applications can give insights into the landscape, suitable for solutions that need to be built through small incremental changes

- Cons: more likely to get **stuck in local optima**, **unable to escape** local optima

Simulated Annealing

- Big question: how do we escape local optima, if we are one?
  - Thought 1: we never know whether we are climbing a local or a global optimum!
  - Thought 2: assuming that there are more local than global optima, it makes sense to escape.
  - Thought 3: but not always - when we stop, we want to stop near the top of SOME optimum.

Annealing (풀림)

- Keep metal in a very high temperature for a long time, and then slowly cool down: it then becomes more workable.

- At high temperature, atoms are released from internal stress by the energy; during the cool-down, they form new nucleates without any strain, becoming softer.

Simulated Annealing

- Introduce “temperature” into local search: start with a high temperature, and slowly cool down.
  - When the temperature is high, the solution (like atom) is unstable and can make random moves (i.e. escapes).
  - As the temperature decreases, the energy level gradually gets lower, and escapes become more infrequent.

Simulated Annealing

$\textsc{SimulatedAnnealing}()$

$s = s_0$
$T \leftarrow T_0$
for $k = 0$ **to** n
  $s_{new} \leftarrow \textsc{GetRandomNeighbour}(s)$
  if $P(\mathrm{F}(s), \mathrm{F}(s_{new}), T) \geq \mathit{random}(0,1)$ **then** $s \leftarrow s_{new}$
$T \leftarrow \textsc{Cool}(T)$
return s

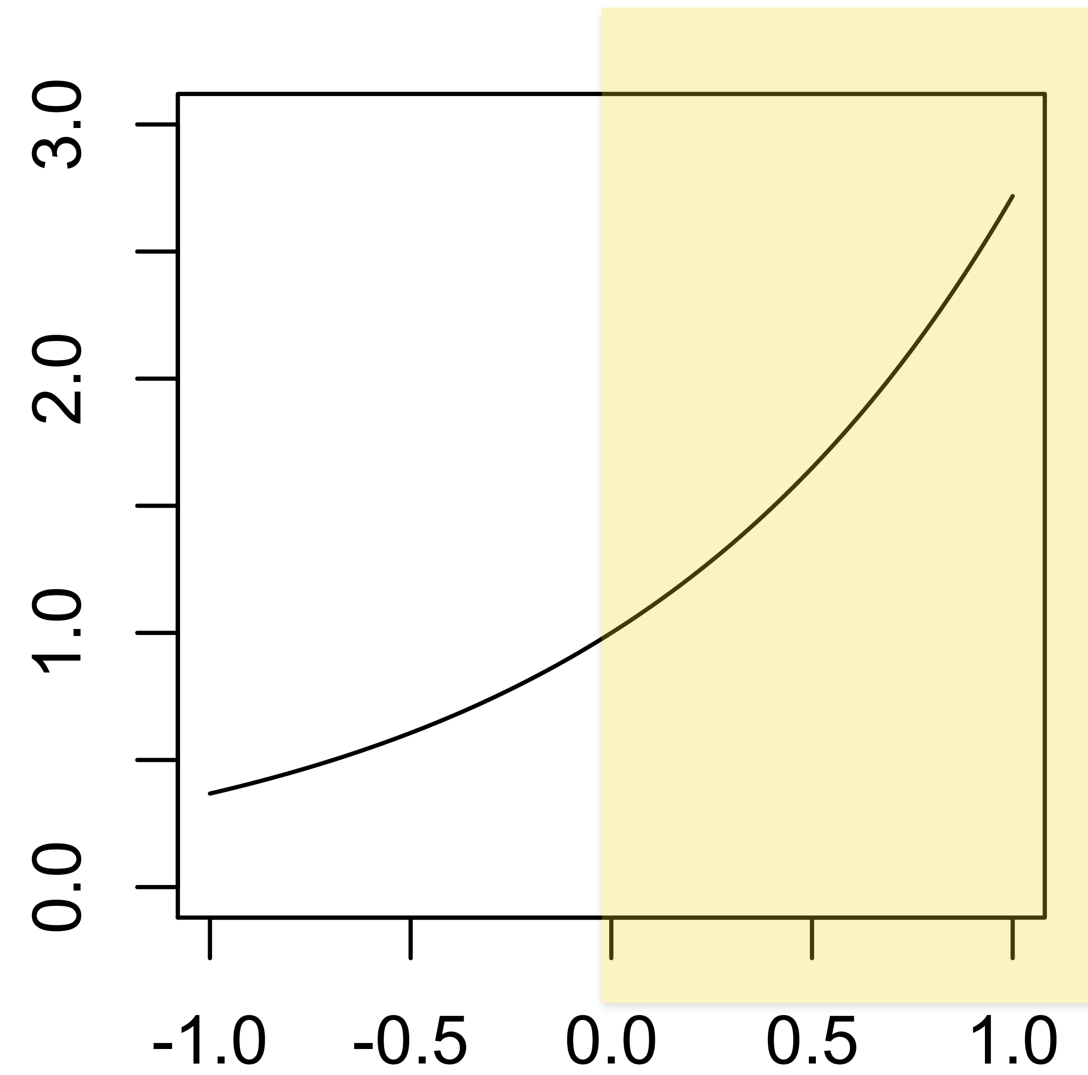
$\mathrm{P}(F(s)$,$F(s_{new})$, $T)$

if $F(s_{new}) > F(s)$ **then return** 1.0
else return $e^{\frac{F(s_{new}) - F(s)}{T}}$

Acceptance Probability

- Borrowed from metallurgy
- When new solution is better ($F(s_{new}) > F(s)$), always accept ($P > 1$)
- When new solution is equally good, accept
- When new solution is worse:
  - more likely to accept small downhill movement
  - gets smaller as temperature drops

$$P = e^{\frac{F(s_{new}) - F(s)}{T}}$$



Cooling Schedule

Temperature at time step t as a function of t:

- Linear: $T(t) = T_0 - \alpha t$
- Exponential: $T(t) = T_0 \alpha^t (0 < \alpha < 1)$
- Logarithmic: $T(t) = \frac{c}{\log{(t+d)}}$

  - With large c, this can be very slow cooling
  - There is an existence proof that says logarithmic will find the global optimum in infinite time… huh?
  - It becomes essentially a random search
  - Theoretically interesting, but practically not so much.

Tabu Search

- Another attempt to escape local optima
- Two exceptions to local search:
  - It is possible to accept a worse move
  - Remember “visited” solutions and avoid coming back

Tabu Search

```
TabuSearch()
    s ← s_0
    s_best ← s
    T ← [] // tabu list
    while not stoppingCondition()
        c_best ← null
        foreach c ∈ GetNeighbours(s)
            if (c ∉ T) ∧ (F(c) > F(c_best)) then c_best ← c
        s ← c_best
        if F(c_best) > F(s_best) then s_best ← c_best
        append(T, c_best)
        if |T| > maxTabuSize then removeAt(T, 0)
    return sBest
```

Tabu list is a FIFO queue: with the `maxTabuSize` we can control the memory span of the search.

Random Restart

- Search budget is usually given in limited time (“terminate after 5 minutes”) or in number of fitness evaluation (“terminate after 5000 fitness evaluations”)
- If a local search reaches optima and budget remains? Start again from another random solution and keep the best answer across multiple runs.

Search Radius

- For local search algorithms to be effective: the search space may be large, but the search radius should be reasonably small

- Search radius: the number of moves required to go **across** the search space

Search Radius: TSP

- Travelling Salesman Problem: what is the shortest path that visits all N cities?
  - Search Space: N! (e.g 2,432,902,008,176,640,000 when N = 20)
  - Search Radius: at most N(N-1)/2 swaps to change any permutation of cities to any other (e.g. 190 when N=20)

Summary

- Local search: direct use of fitness landscape concept, with various mechanism to escape local optima.
- Easy to implement, easy to understand what is going on; good for insights into landscape
- Design of search space (especially discrete one) affects the performance of search