UNIVERSITÄT PASSAU

Search-Based Software Engineering

Introduction

Gordon Fraser
Lehrstuhl für Software Engineering II

Software Engineering

As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

— *Edsger Dijkstra, The Humble Programmer, Communications of the ACM, 1972*

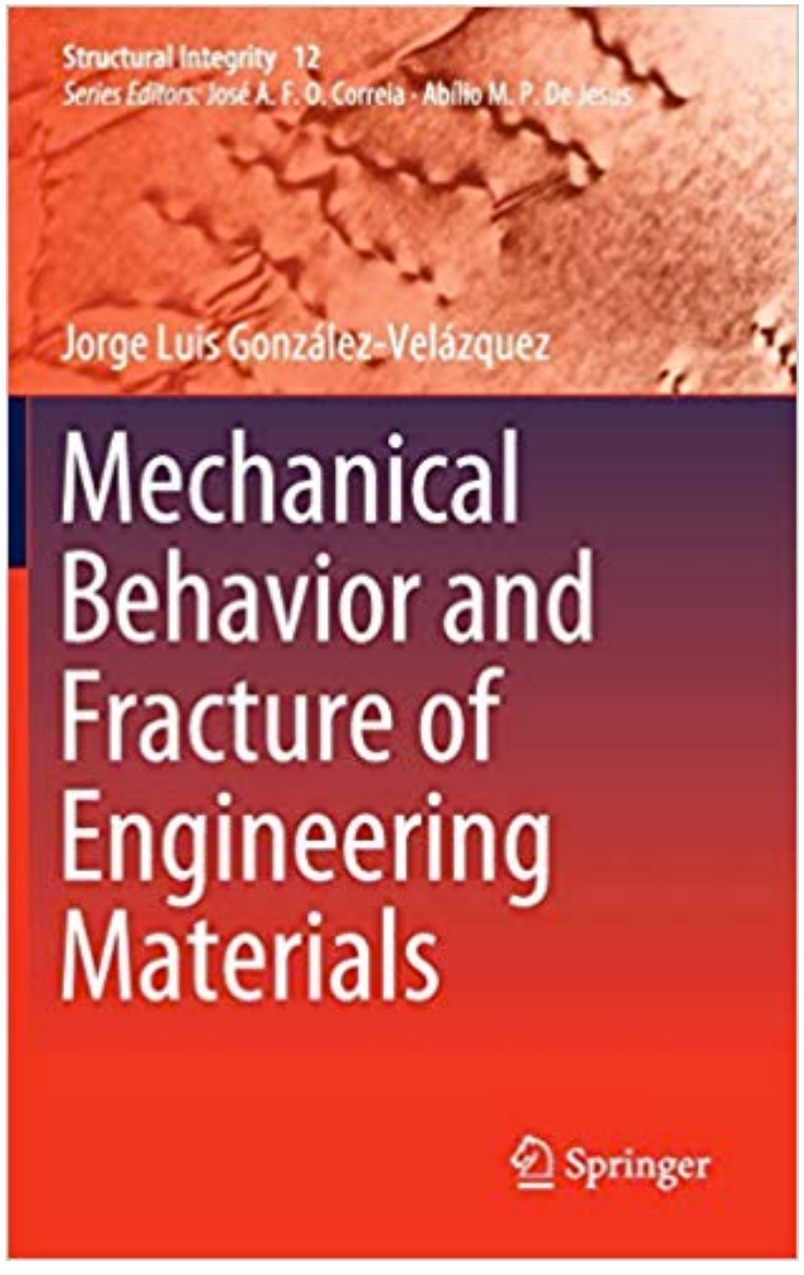
Software Engineering

- NATO conference 1968 concluded that software engineering should use the philosophies and paradigms of established engineering disciplines, to solve the problem of software crisis.
- But what do “established engineering principles” do to improve quality?
  - Theory
  - Simulation
  - Optimisation

Theory

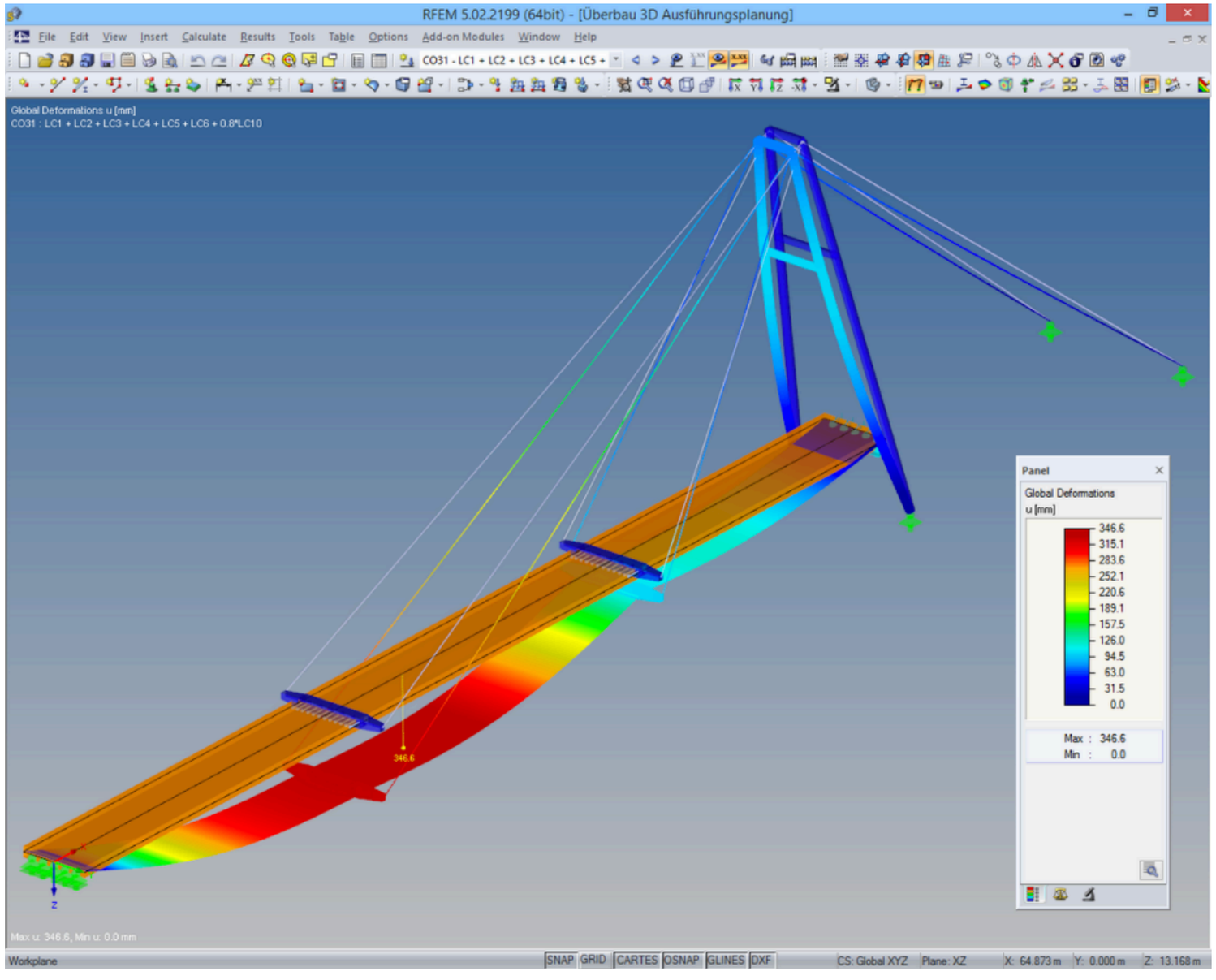
When does a steel beam break?

- Stress: force per unit area
- Tensile strength: the maximum stress a material can resist



Simulation

Given the physical laws as the foundation, it is possible to build simulations.



Optimisation

…and with simulation optimisation follows naturally.

It is, simply, trial and error, which is only possible because we have the simulation environment.

Creating Models of Truss Structures with Optimization

Jeffrey Smith
Carnegie Mellon University

Jessica Hodgins
Carnegie Mellon University

Irving Oppenheim
Carnegie Mellon University

Andrew Witkin
Pixar Animation Studios

Abstract

We present a method for designing truss structures, a common and complex category of buildings, using non-linear optimization. Truss structures are ubiquitous in the industrialized world, appearing as bridges, towers, roof supports and building exoskeletons, yet are complex enough that modeling them by hand is time consuming and tedious. We represent trusses as a set of rigid bars connected by pin joints, which may change location during optimization. By including the location of the joints as well as the strength of individual beams in our design variables, we can simultaneously optimize the geometry and the mass of structures. We present the details of our technique together with examples illustrating its use, including comparisons with real structures.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; G.1.6 [Numerical Analysis]: Optimization—Nonlinear programming; G.1.6 [Numerical Analysis]: Optimization—Constrained optimization

Keywords: Physically based modeling, truss structures, constrained optimization, nonlinear optimization

1 Introduction

A recurring challenge in the field of computer graphics is the creation of realistic models of complex man-made structures. The standard solution to this problem is to build these models by hand, but this approach is time consuming and, where reference images are not available, can be difficult to reconcile with a demand for visual realism. Our paper presents a method, based on practices in the field of structural engineering, to quickly create novel and physically realistic truss structures such as bridges and towers, using simple optimization techniques and a minimum of user effort.

"Truss structures" is a broad category of man-made structures, including bridges (Figure 1), water towers, cranes, roof support trusses (Figure 10), building exoskeletons (Figure 2), and temporary construction frameworks. Trusses derive their utility and distinctive look from their simple construction: rod elements (beams) which exert only axial forces, connected concentrically with welded or bolted joints.

These utilitarian structures are ubiquitous in the industrialized world and can be extremely complex and thus difficult to model. For example, the Eiffel Tower, perhaps the most famous truss structure in the world, contains over 15,000 girders connected at over 30,000 points [Harriss 1975] and even simpler structures, such as railroad bridges, routinely contain hundreds of members of varying lengths. Consequently, modeling of these structures by hand can be difficult and tedious, and an automated method of generating them is desirable.

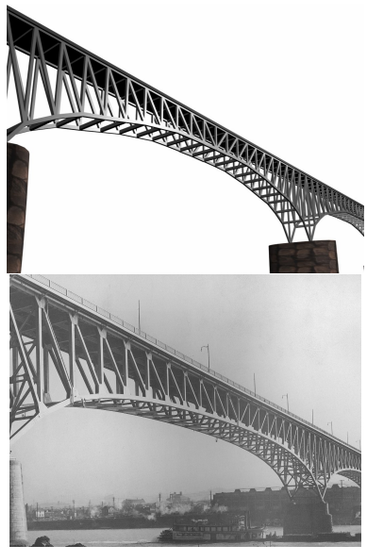


Figure 1: A cantilever bridge generated by our software, compared with the Homestead bridge in Pittsburgh, Pennsylvania.

1.1 Background

Very little has been published in the graphics literature on the problem of the automatic generation of man-made structures. While significant and successful work has been done in recre-

{jeffrey|jkh}@cs.cmu.edu, ijo@andrew.cmu.edu, aw@pixar.com

Software “Engineering”

	Theory	Simulation	Optimisation	Product
Bridge Building	*Abstract*	*Computation*	*Computation*	*Real World*
Software Engineering	*Best Practices*	*?*	*?*	*Computation*

Software “Engineering”

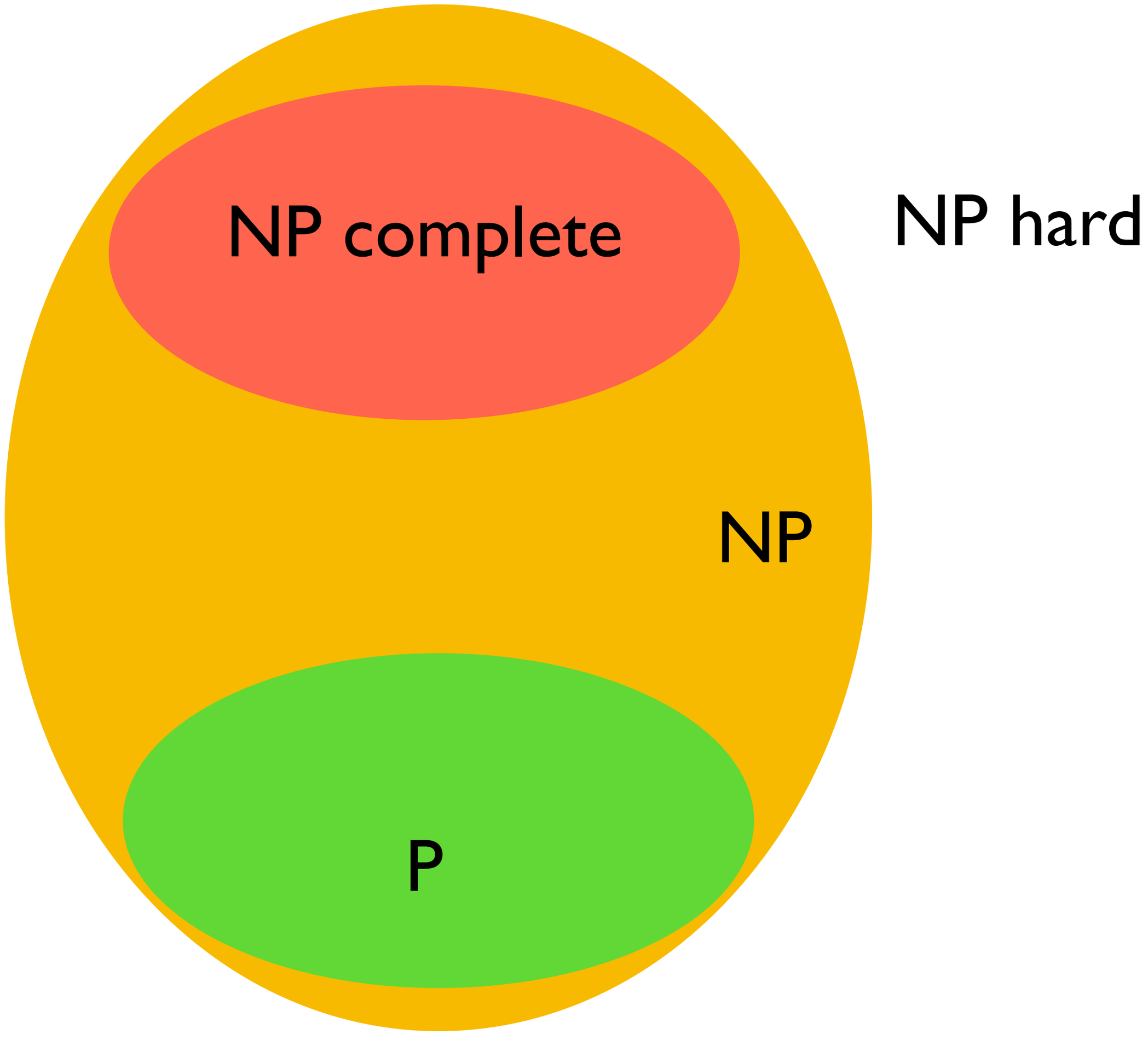
	Theory	Simulation	Optimisation	Product
Bridge Building	*Abstract*	*Computation*	*Computation*	*Real World*
Software Engineering	*Best Practices*	*Search-Based Software Engineering*		

Contents

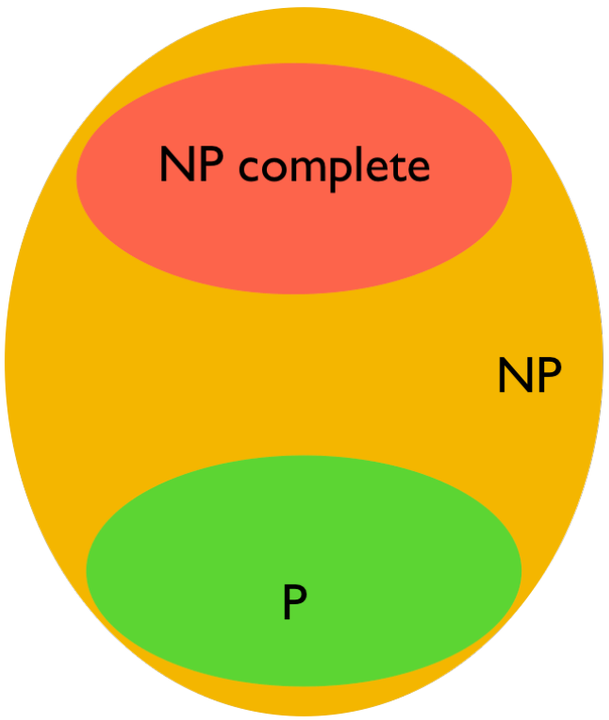
- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes


NP complete
NP hard
NP
P

Decision Problems



- Complexity theory is built around the concept of decision problems, i.e., problems that require a *yes* or *no* answer.
  - *P* is a decision problem if the instances I_P of *P* can be partitioned into two sets: the set of positive instances Y_P and the set of negative instances N_P.
  - Algorithm *AP* solves P if, for all $i \in Y_P$ it yields "*yes*" as an answer and for all instances $i \in N_P$ it yields "*no*" as an answer.
- For problems in NP, a "*yes*" answer can be checked in polynomial time.

Example: Connection of a Graph

- Let *G(V, E)* be a graph with *V* representing the set of vertices and *E* the set of edges

- *G* is connected if there is a path of finite length between any two arbitrary vertices $v_1, v_2 \in V$.

- The decision problem *P* consists of answering the following question: *Is the graph G connected?*

- An answer can be found in time O(|*V*| + |*E*|) (polynomial in the size of the graph) by breadth first search

Example: Hamiltonian Cycles

- Let *G(V, E)* be a graph with *V* representing the set of vertices and *E* the set of edges

- A Hamiltonian cycle is a path that contains all the vertices of *G* and visits each of them only once, before returning to the start vertex.

- Problem: *Does a graph G contain a Hamiltonian cycle?*

- No polynomial time algorithm is known for solving this problem.

- Given a path $\{v_1, v_2, \ldots, v_N, v_1\}$ of *G* it is easy to check if it is a Hamiltonian cycle.

Optimisation Problems

- Decision Problem: Does a graph *G* contain a Hamiltonian cycle of length $L \leq K$?
- Optimisation Problem: Given *G* with the set of vertices $V=\{v_1, \ldots, v_n\}$ and the *n x n* matrix of distances $d_{ij} \in \mathbb{Z}+$, find a permutation $\Pi(V)$ of *V* such that the corresponding tour length $L(\Pi)=\Sigma_{i=1}^{n-1} d_{vi,vj+1} + d_{vn,vi}$ is minimal.
  - Hamiltonian cycle decision problem is NP-complete
  - Travelling Salesman Problem is at least as hard.
- Optimisation problem: Finding a solution that maximises (or minimises) a given criterion called objective function
- A solution may be required to obey certain constraints to be feasible (admissible)
- Let *S* be a finite set of feasible solutions to a problem, and *f* a cost function $f : S \rightarrow R$, then an instance of an optimisation problem *P* asks for $x \in S$ such that:
  - $f(x) \geq f(s), \forall s \in S$

How to Solve Optimisation Problems?

- Special cases:
  - Worst case scenario doesn't always appear in practice
  - Some problems have solvable scenarios, e.g. 2-SAT
  - Simplex algorithm for linear programming problems has exponential complexity, but usually is quick
- Brute-force computation:
  - Increase in computer power may allow enumerating all admissible solutions
- Parallel computing:
  - Not all algorithms are easily parallelisable

Metaheuristics

heuristic | ˌhjʊ(ə)ˈrɪstɪk |

adjective

enabling a person to discover or learn something for themselves. *a 'hands-on' or interactive heuristic approach to learning.*

• Computing proceeding to a solution by trial and error or by rules that are only loosely defined.

Metaheuristics

heuristic | ˌhjʊ(ə)ˈrɪstɪk |

adjective

enabling a person to discover or learn something for themselves. *a 'hands-on' or interactive heuristic approach to learning.*

• Computing proceeding to a solution by trial and error or by rules that are only loosely defined.

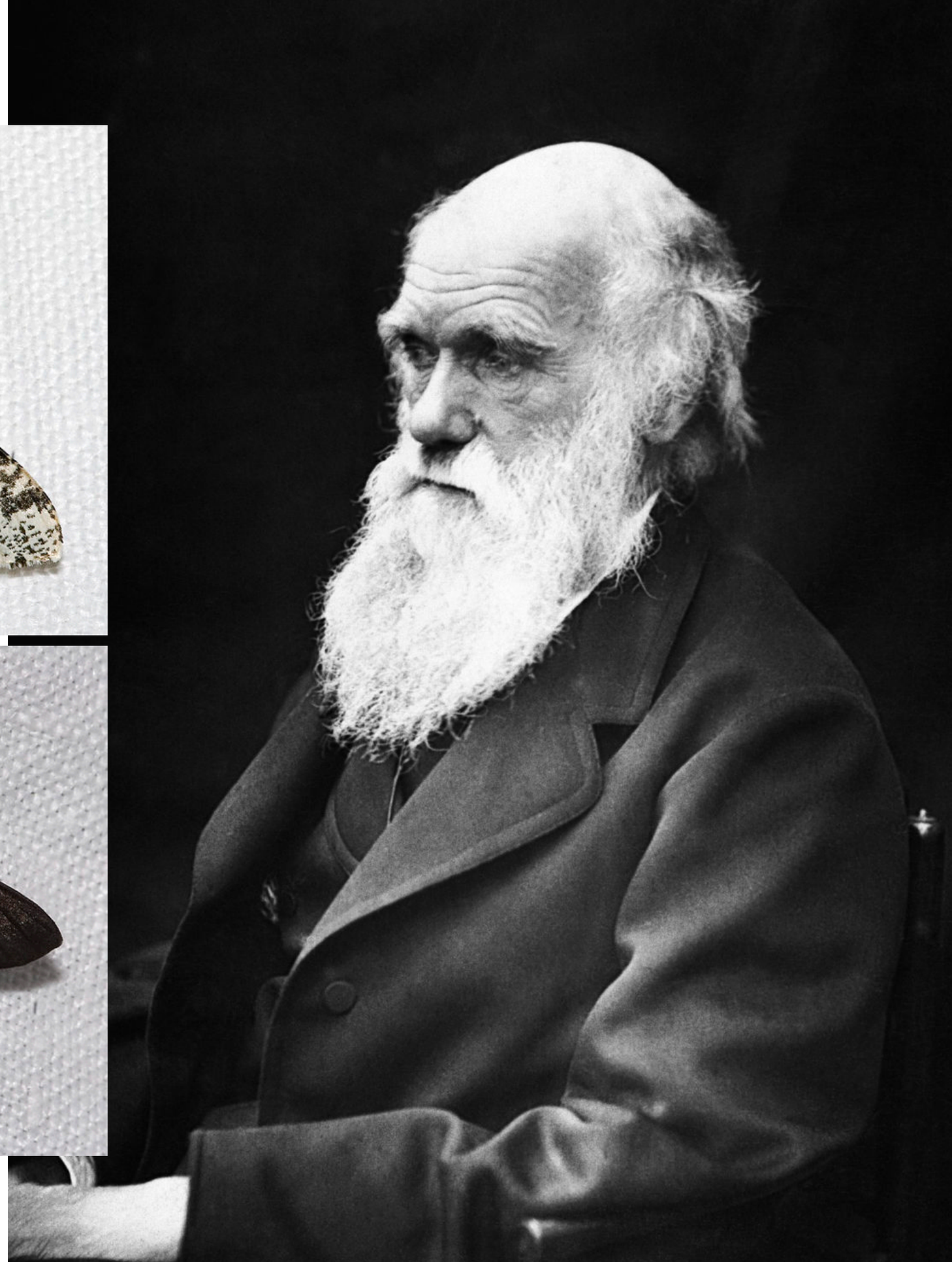
meta- | ˈmɛtə | (also **met-** before a vowel or h)

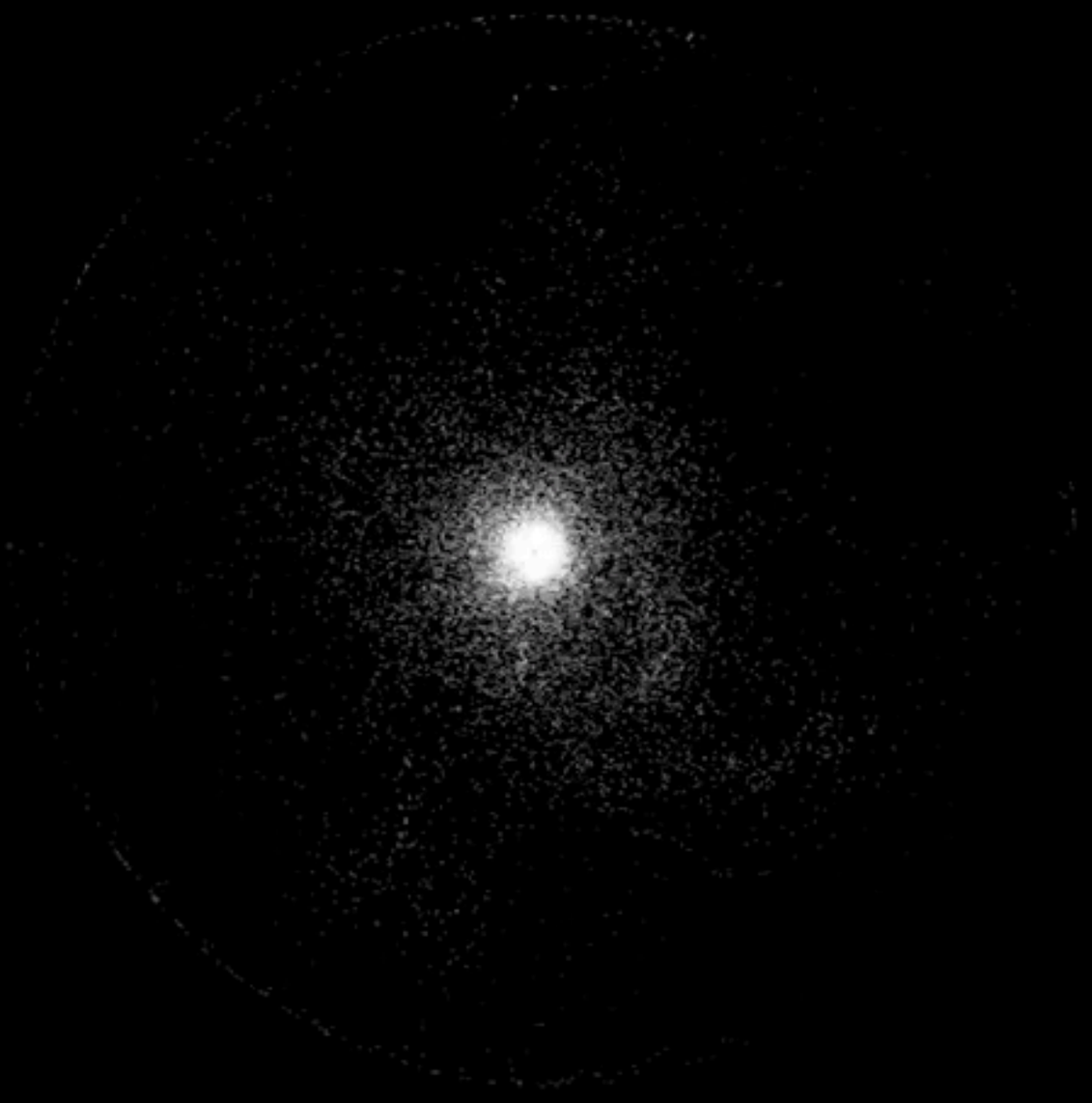
combining form

1 denoting a change of position or condition: *metamorphosis.*
2 denoting position behind, after, or beyond: *metacarpus.*
3 denoting something of a higher or second-order kind: *metalanguage* | *metonym.*
4 Chemistry denoting substitution at two carbon atoms separated by one other in a benzene ring, e.g. in 1,3 positions: *metadichlorobenzene.* Compare with ORTHO- and PARA-[1] (SENSE 2).
5 Chemistry denoting a compound formed by dehydration: *metaphosphoric acid.*

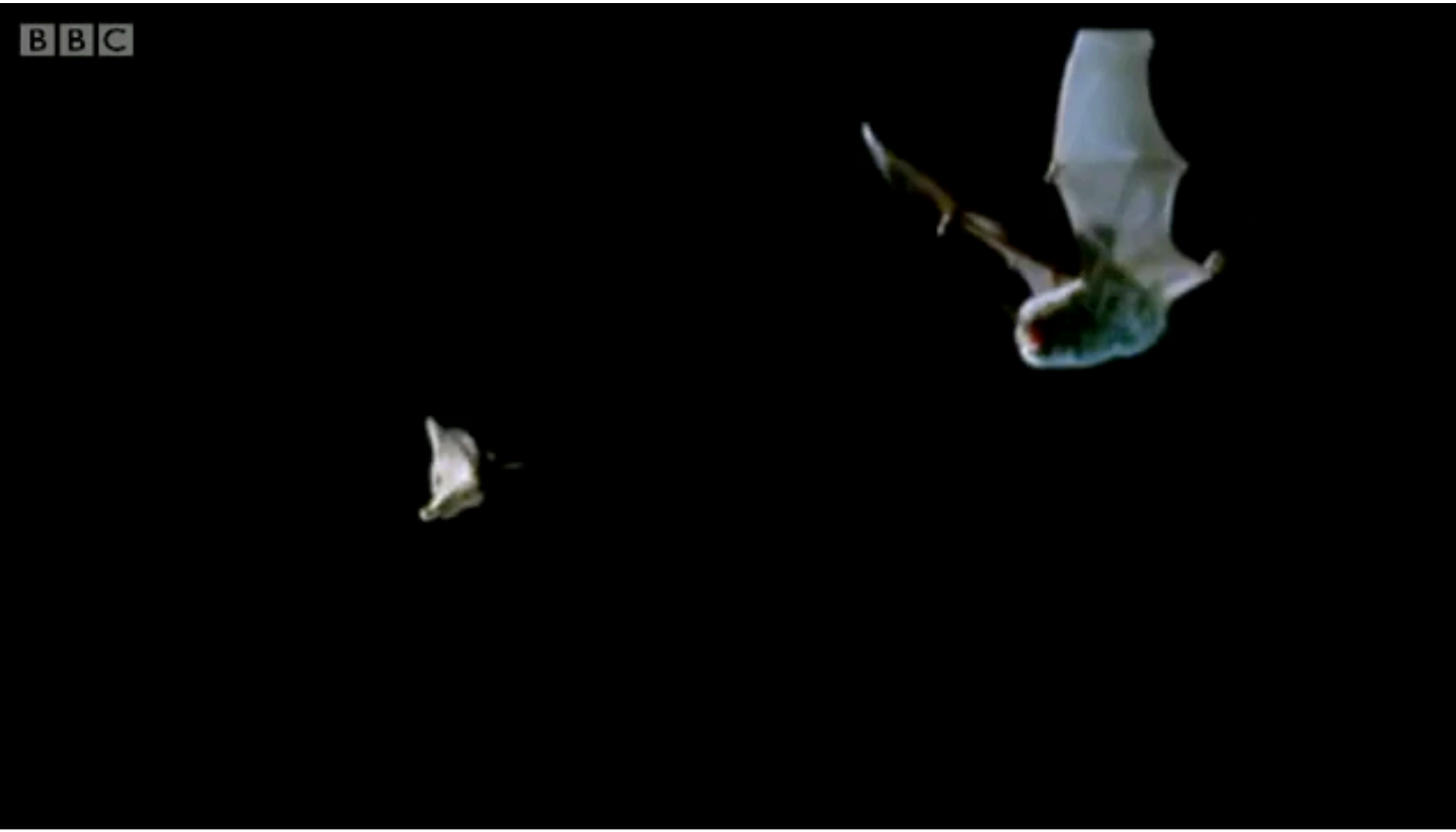
Metaheuristics

- Approximation algorithms that provide good or acceptable solutions within an acceptable computing time
- No formal guarantees about the quality of solutions or global optimality
- Usually non-deterministic
- Not problem specific
- Smart trial and error






Smithsonian
CHANNEL


BBC

Example: Super Mario

Classic Nintendo Games are
(Computationally) Hard

Greg Aloupis* Erik D. Demaine† Alan Guo†‡ Giovanni Viglietta§

February 10, 2015

Abstract

We prove NP-hardness results for five of Nintendo's largest video game franchises: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. Our results apply to generalized versions of Super Mario Bros. 1–3, The Lost Levels, and Super Mario World; Donkey Kong Country 1–3; all Legend of Zelda games; all Metroid games; and all Pokémon role-playing games. In addition, we prove PSPACE-completeness of the Donkey Kong Country games and several Legend of Zelda games.

1 Introduction

A series of recent papers have analyzed the computational complexity of playing many different video games [1, 4, 5, 6], but the most well-known classic Nintendo games have yet to be included among these results. In this paper, we analyze some of the best-known Nintendo games of all time: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. We prove that it is NP-hard,

Example: Super Mario

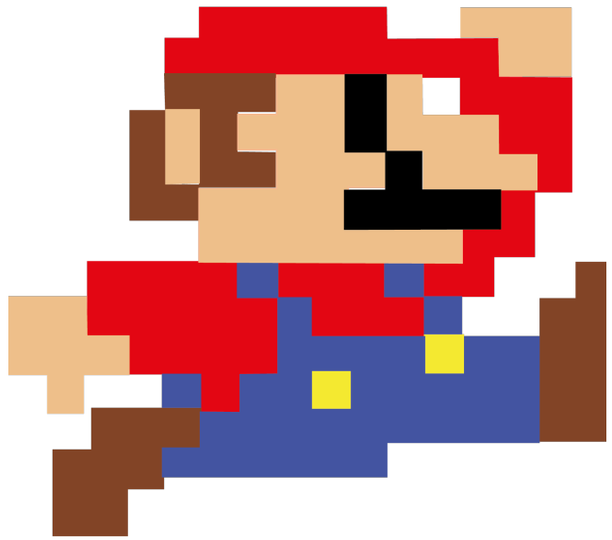
Player A

- Read the game manual to see which button does what.

- Google level map and get familiar with it.

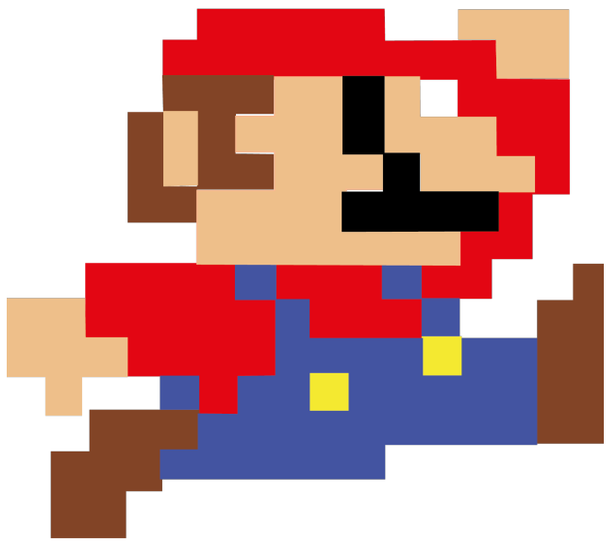
- Carefully, very carefully, plan when to press each button, for how long.

- Grab the controller and execute the plan.



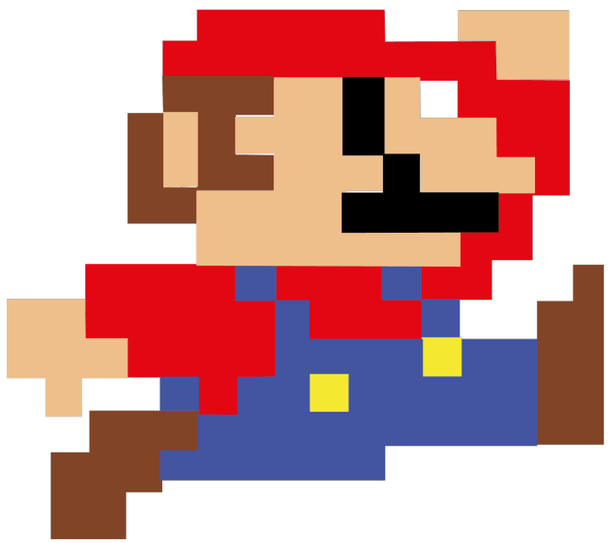
Example: Super Mario

Player B



- Grab the controller.
- Play.
- Die.
- Repeat until level is cleared.
- Intelligence lies in how differently you die next time.

Similarity to Metaheuristic Search



- You make small changes to your last attempt ("okay, I will press A slightly later this time").

- You combine different bits of solutions ("Okay, jump over here, but then later do not jump over there").

- You accidentally discover new parts of the map ("Oops, how did I find this secret passage?")

Key Ingredients

- What are we going to try this time? (representation)
- How is it different from what we tried before? (operators)
- How well did we do this time? (fitness/objective function)
- Minor (but critical) ingredients: constraints

Example: Super Mario

- Representation: a list of (button, start_time, end_time)
- Operators: change button type in one tuple, increase/ decrease start_time or end_time
- Fitness function: the distance you travelled without dying

Example: Super Mario

- Representation: neural network
- Operators: change weights, change network structure
- Fitness function: the distance you travelled without dying

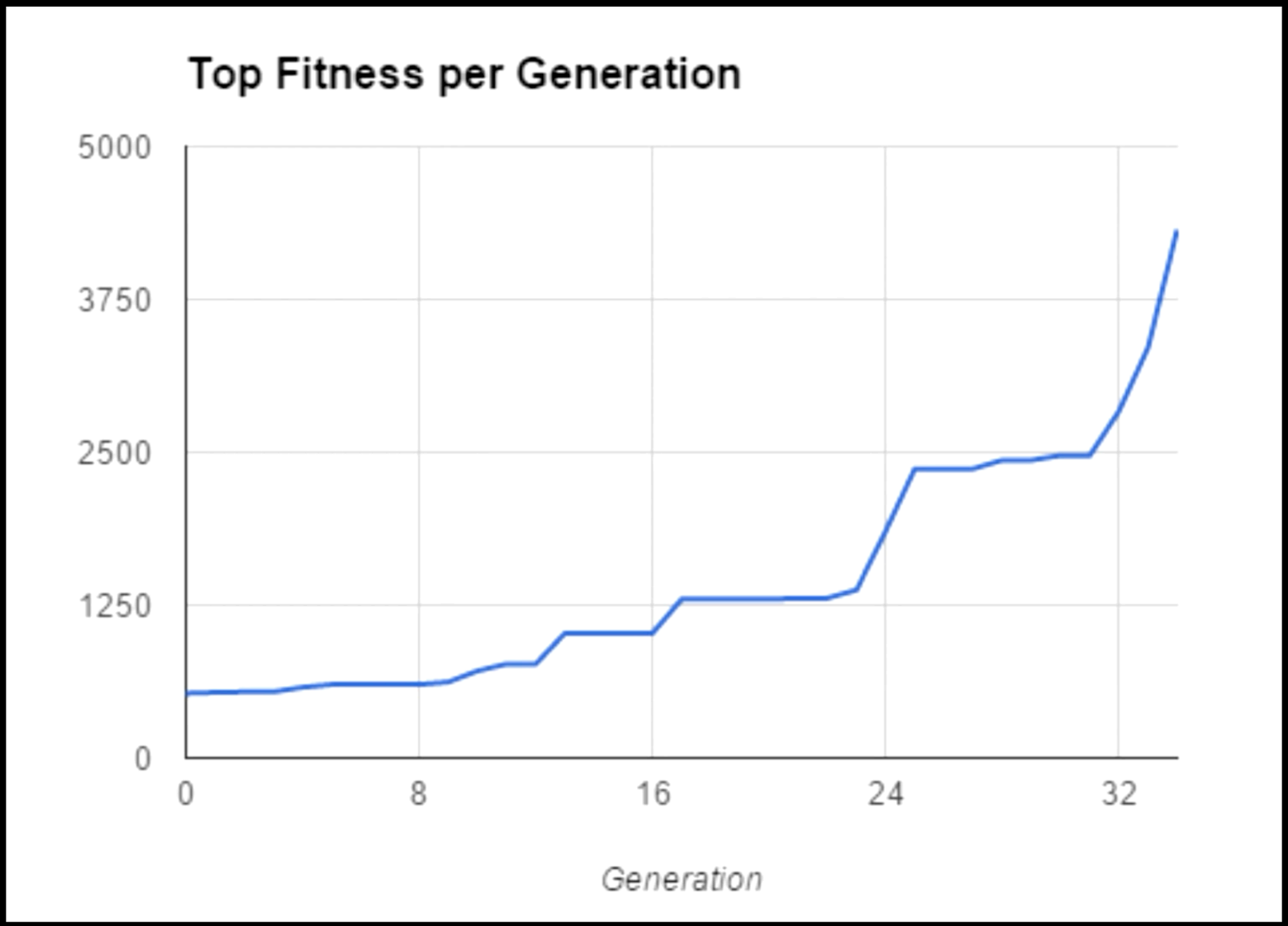
Gen 34 species 14 genome 14 (37%)
Fitness: 35 Max Fitness: 4322

A
B
X
Y
Up
Down
Left
Right

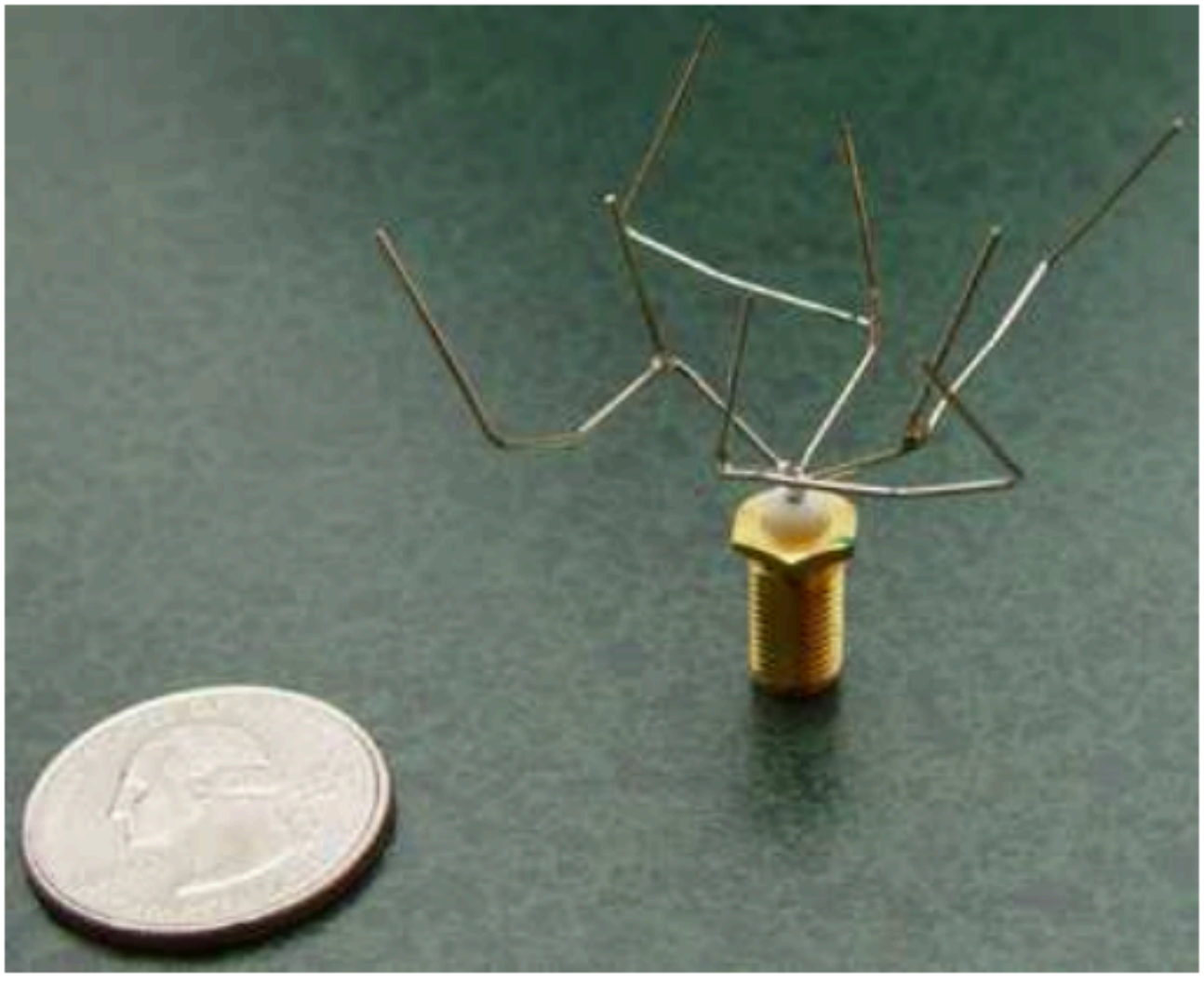
Gen 0 species 24 genome 1 (7%)
Fitness: 2 Max Fitness: 528

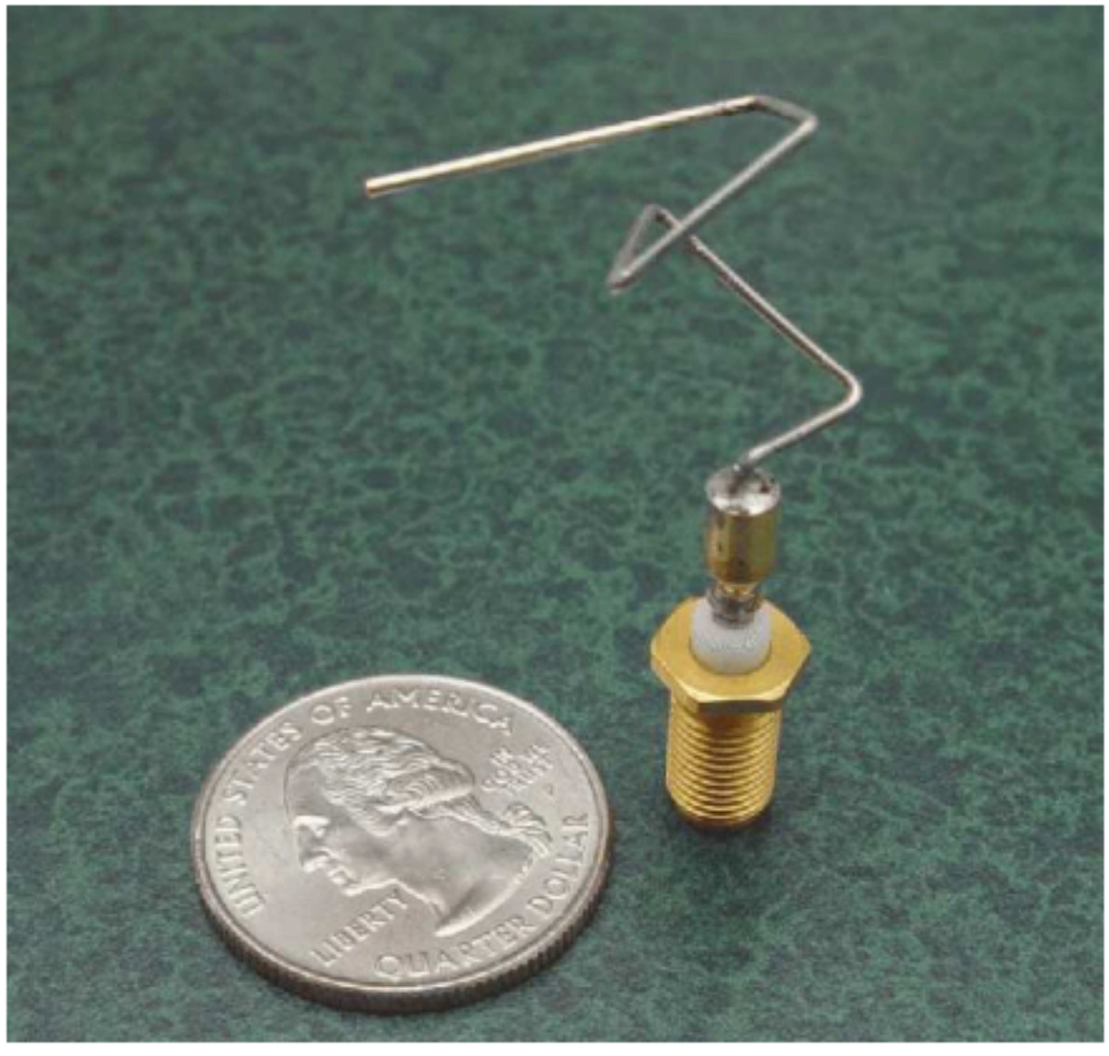
A
B
X
Y
Up
Down
Left
Right

Loaded state: DP1.state
Loaded state: DP1.state


Top Fitness per Generation
5000
3750
2500
1250
0
0
8
16
24
32
Generation

NASA's Space Technology 5 (ST5) mission: Evolved antenna





Contents

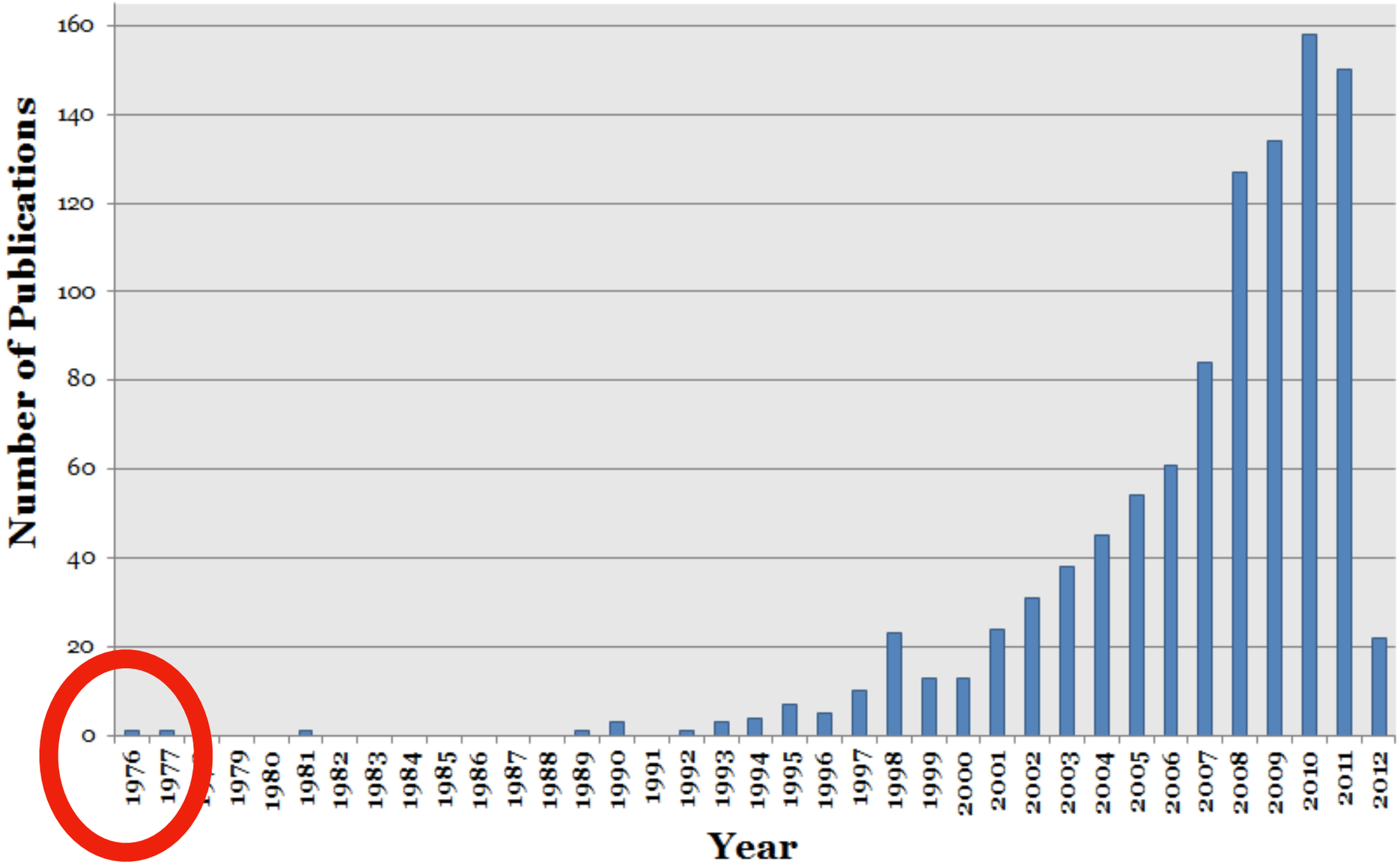
- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

SBSE

- A large movement(?) that seeks to apply various optimisation techniques to software engineering problems (NOT search engines or code search)
- Still relatively young (by academic standards)



source: SEBASE publications repository
http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/

Structural Testing

Webb Miller David Spooner

Automatic Generation of Floating-Point Test Data IEEE Transactions on Software Engineering, 1976

Automatic Generation of Floating-Point Test Data

WEBB MILLER AND DAVID L. SPOONER

Abstract—For numerical programs, or more generally for programs with floating-point data, it may be that large savings of time and storage are made possible by using numerical maximization methods instead of symbolic execution to generate test data. Two examples, a matrix factorization subroutine and a sorting method, illustrate the types of data generation problems that can be successfully treated with such maximization techniques.

Index Terms—Automatic test data generation, branching, data constraints, execution path, software evaluation systems.

Introduction

RESEARCH in program evaluation and verification has only rarely (e.g., [1]) begun with the explicit requirement that the program deal with real numbers as opposed to integers. This may be an oversight since there are theoretical results which suggest the desirability of this assumption. Specifically, a general procedure of Tarski [2] shows that certain properties, undecidable (in the technical sense) for "integer" programs, are decidable for "numerical" programs. Examples of this phenomenon arise when one asks if there exists a set of data driving execution of a certain kind of program down a given path.

Moreover, there is practical evidence supporting the case for automatic verification of special properties of numerical programs. Proving "numerical correctness," i.e., verifying a satisfactory level of insensitivity to rounding error, is sometimes much easier than proving that the program performs properly in exact arithmetic. The ideal and contaminated results can often be meaningfully compared with only minimal understanding of the program. Simple, portable, general-purpose software [3], [4] can easily provide answers which have eluded specialists in roundoff analysis. This work [3], [4] also shows the possible advantage of using, e.g., numerical maximization methods to do the verification, avoiding the alternative of using, e.g., computer symbolic manipulation [5], [6].

This paper considers automatic test data generation, a problem which arises in such fields as automatic software evaluation systems [7] and in automatic roundoff analysis [4]. Our contention is that automatic test data generation is sometimes best formulated and solved as a numerical maximization problem. The reader should be warned that our scheme is only "heuristic" in that it is not guaranteed to produce a set of test data executing a given path whenever such data exist. (On the other hand, we know of no guaranteed data generation scheme whose execution time does not, in the worst case, grow at least exponentially with the length of the execution path.)

Manuscript received September 9, 1975; revised February 23, 1976. This work was supported in part by the National Science Foundation under Grant GJ-42968.

W. Miller is with the Department of Computer Science, Pennsylvania State University, University Park, PA 16802.

D. L. Spooner was with the Department of Computer Science, Pennsylvania State University, University Park, PA 16802. He is now with the Department of Computer Science, Cornell University, Ithaca, NY.

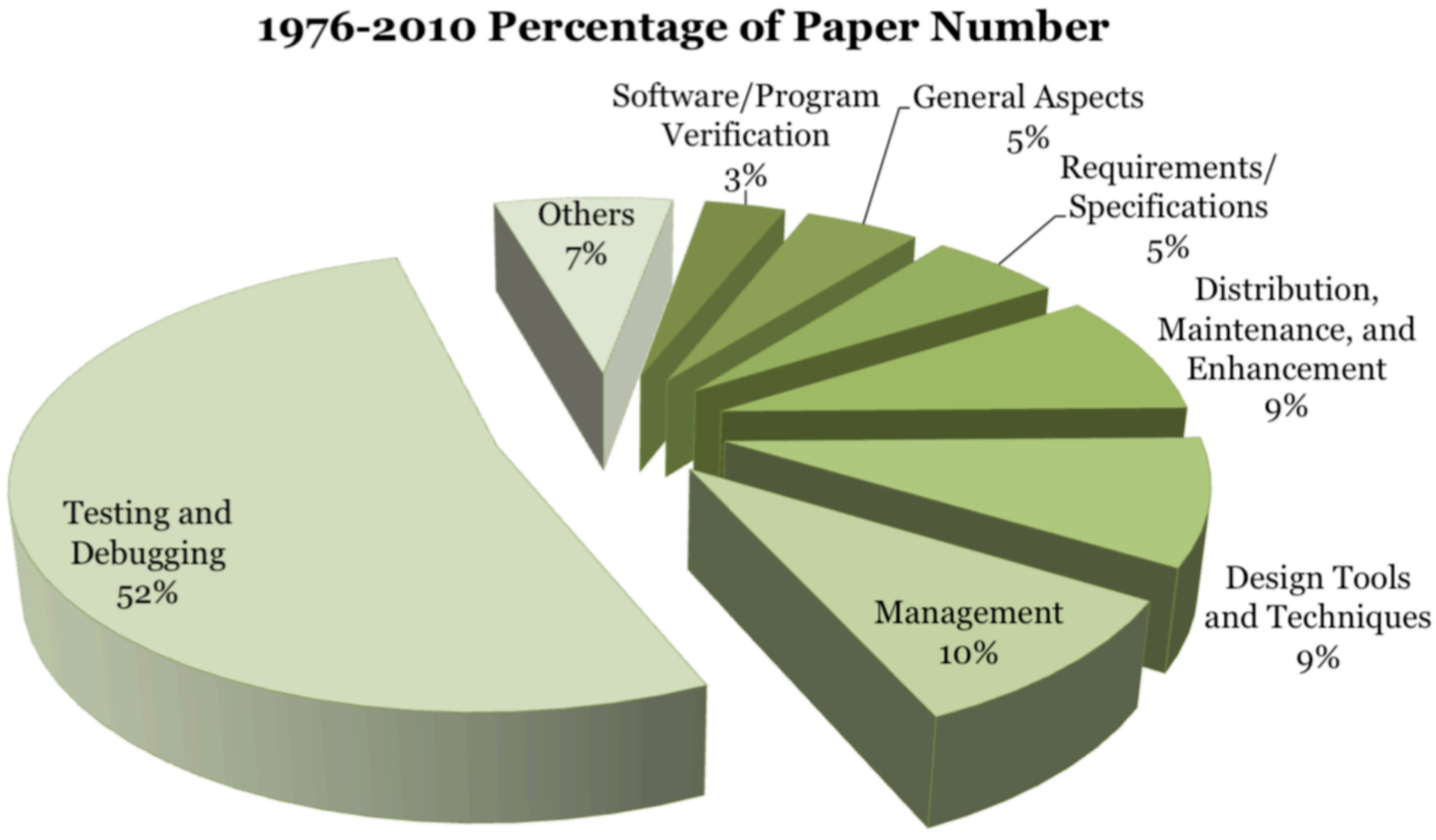
Numerical Maximization Methods for Generating Test Data

Given the problem of generating floating-point test data our approach begins by fixing all integer parameters of the given program (e.g., the dimensions of the data in a matrix program or the number of iterations in an iterative method) so that the only unresolved decisions controlling program flow are comparisons involving real values. Then, as will be seen, an execution path takes the form of a straight-line program of floating-point assignment statements interspersed with "path constraints" of the form $c_i = 0$, $c_i > 0$, or $c_i \geqslant 0$. Each c_i is a data-dependent real value possibly defined in terms of previously computed results. For instance, a path which takes the true branch of a test "IF(X.NE.Y)" has a constraint $c > 0$, where, e.g., c = ABS(X − Y) or $c = (X - Y)^2$. (We will not discuss in any detail the philosophical and practical difficulties associated with equality tests when computation is contaminated by rounding error. Nor will we consider the problem of (automatically or manually) generating the straight-line program; we have nothing new to add on this subject.)

The situation is clarified by an example. Consider the following subprogram of Moler [8].

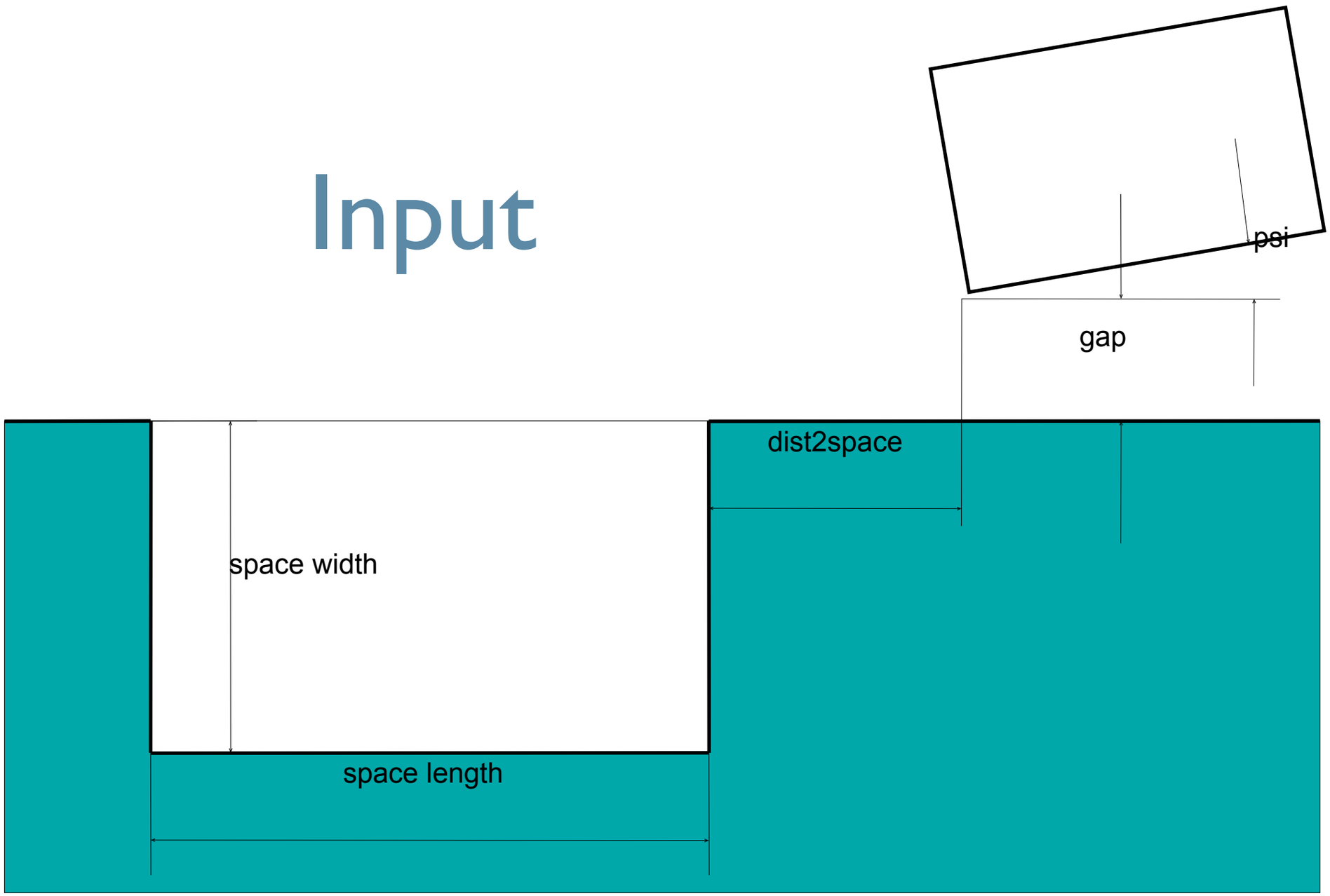
```
      SUBROUTINE DECOMP(N,NDIM,A,IP)
      REAL A(NDIM,NDIM),T
      INTEGER IP(NDIM)
C
C MATRIX TRIANGULARIZATION BY GAUSSIAN ELIMINATION.
C INPUT..
C    N = ORDER OF MATRIX.
C    NDIM = DECLARED DIMENSION OF ARRAY A.
C    A = MATRIX TO BE TRIANGULARIZED.
C OUTPUT..
C    A(I,J), I.LE.J = UPPER TRIANGULAR FACTOR, U.
C    A(I,J), I.GT.J = MULTIPLIERS = LOWER TRIANGULAR
C                     FACTOR, I-L.
C    IP(K), K.LT.N = INDEX OF K-TH PIVOT ROW.
C    IP(N) = (-1)**(NUMBER OF INTERCHANGES) OR 0.
C USE 'SOLVE' TO OBTAIN SOLUTION OF LINEAR SYSTEM.
C DETERM(A) = IP(N)*A(1,1)*A(2,2)*...*A(N,N).
C IF IP(N)=0, A IS SINGULAR, SOLVE WILL DIVIDE BY ZERO.
C INTERCHANGES FINISHED IN U, ONLY PARTLY IN L.
```

Problem Domains



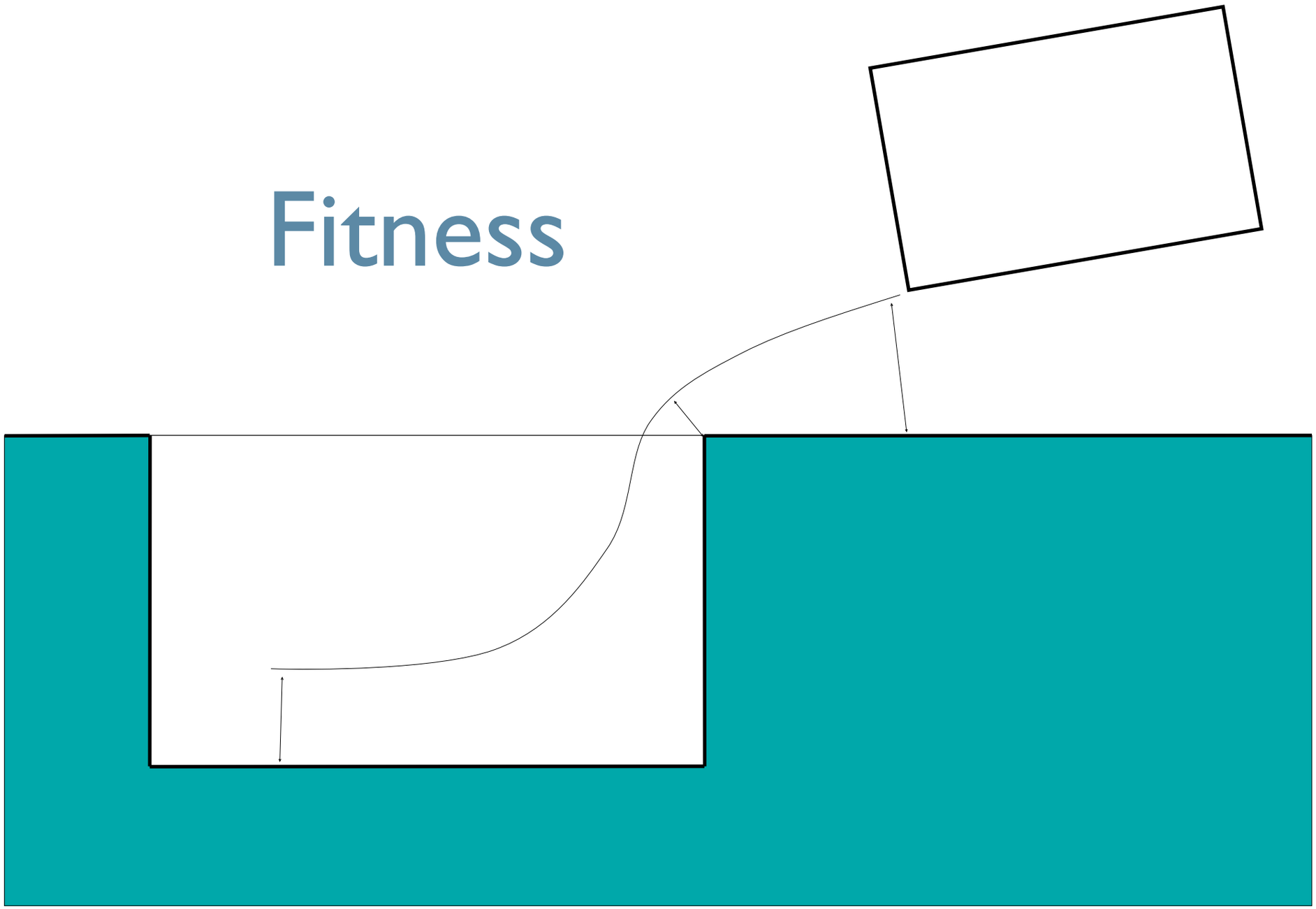
SBST Example: Daimler Autonomous Parking System

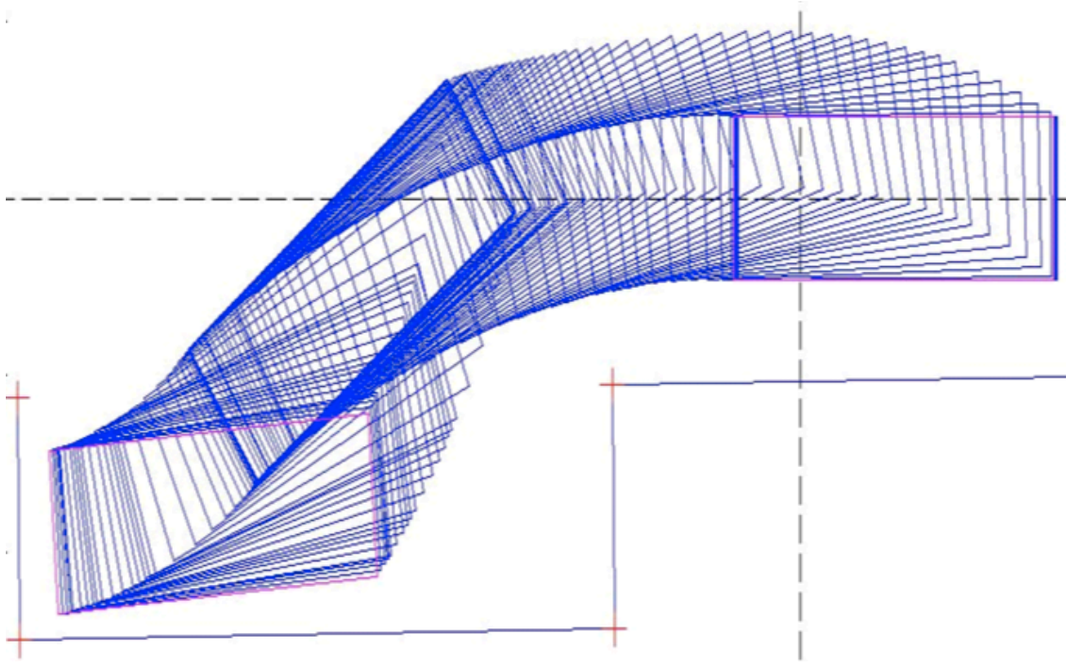
Input



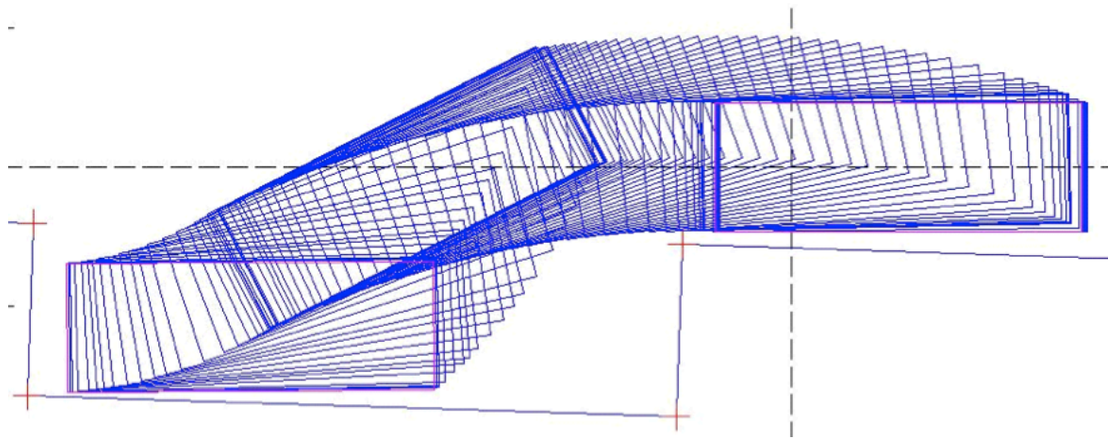
SBST Example: Daimler

Fitness



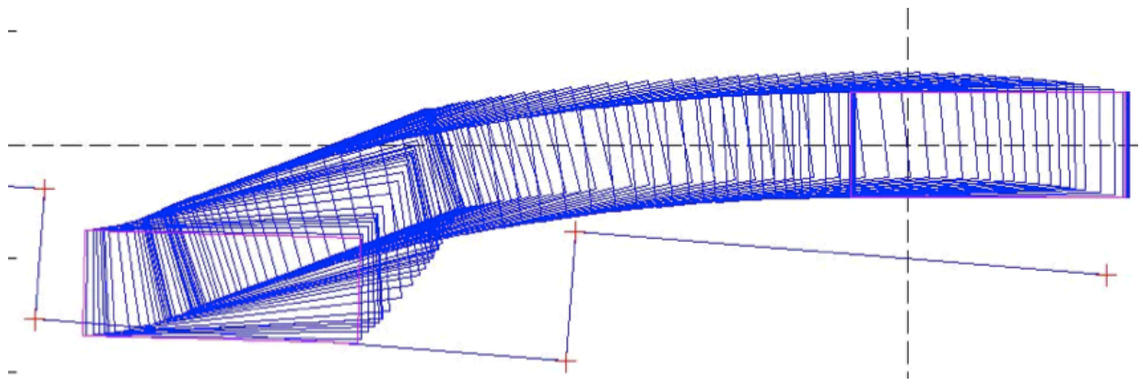


Generation 0



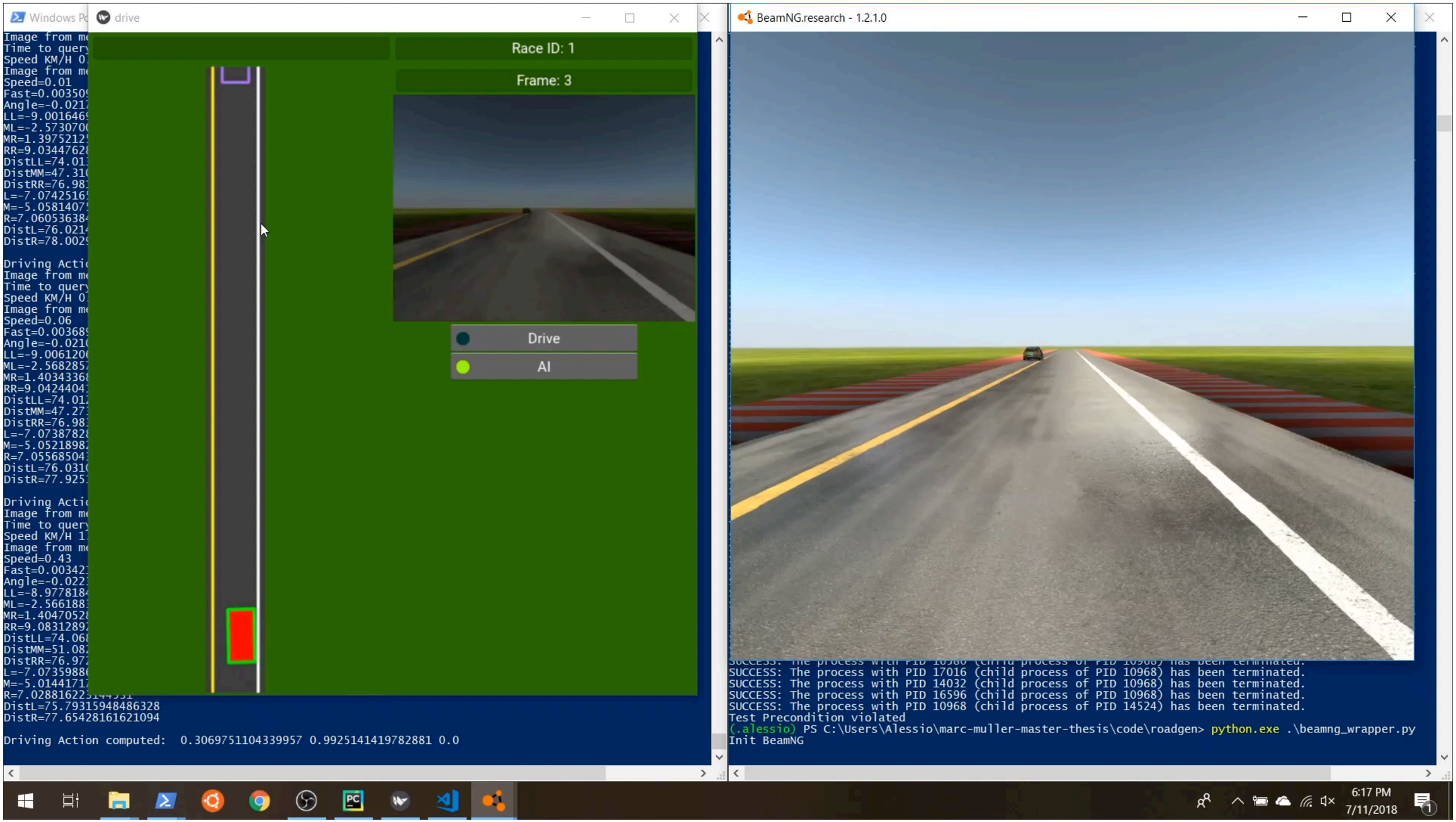
Generation 10

critical

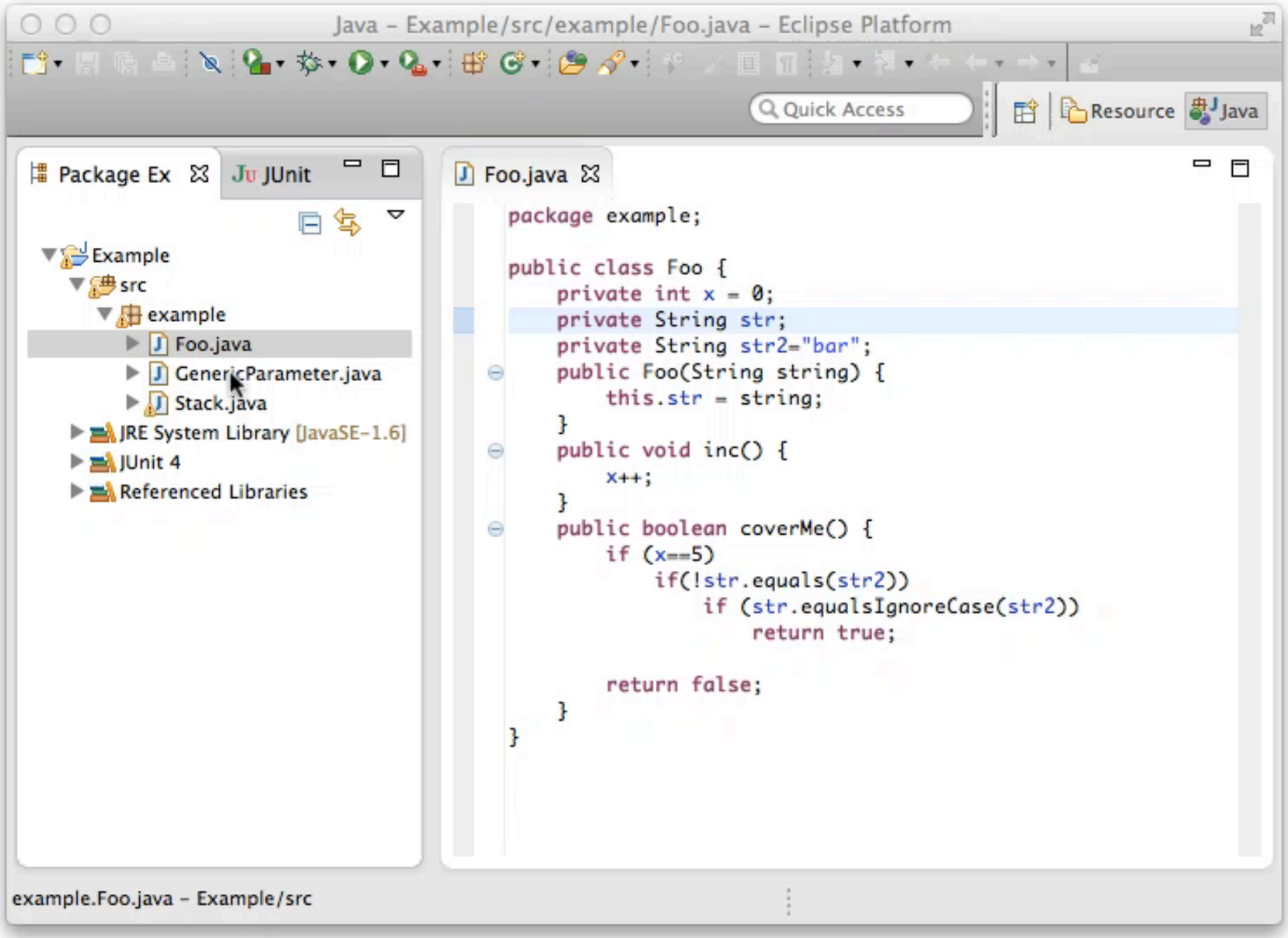


Generation 20

collision


drive
Race ID: 1
Frame: 3
Drive
AI
BeamNG.research - 1.2.1.0
Image from me
Time to query
Speed KM/H 0
Speed=0.01
Fast=0.003509
Angle=-0.021
LL=-9.001646
ML=-2.573070
MR=1.3975212
RR=9.0344762
DistLL=74.01
DistMM=47.31
DistRR=76.98
L=-7.0742516
M=-5.0581407
R=7.0605363
DistL=76.021
DistR=78.002
Driving Acti
Speed=0.06
Fast=0.003689
Angle=-0.021
LL=-9.006120
ML=-2.568285
MR=1.4034336
RR=9.0424404
DistLL=74.01
DistMM=47.27
DistRR=76.98
L=-7.0738782
M=-5.0521898
R=7.0556850
DistL=76.031
DistR=77.925
Speed=0.43
Fast=0.00342
Angle=-0.022
LL=-8.977818
ML=-2.566188
MR=1.4047052
RR=9.0831289
DistLL=74.068
DistMM=51.082
DistRR=76.972
L=-7.0735988
M=-5.0144171
DistL=75.79315948486328
DistR=77.65428161621094
Driving Action computed: 0.3069751104339957 0.9925141419782881 0.0
SUCCESS: The process with PID 17016 (child process of PID 10968) has been terminated.
SUCCESS: The process with PID 14032 (child process of PID 10968) has been terminated.
SUCCESS: The process with PID 16596 (child process of PID 10968) has been terminated.
SUCCESS: The process with PID 10968 (child process of PID 14524) has been terminated.
Test Precondition violated
(.alessio) PS C:\Users\Alessio\marc-muller-master-thesis\code\roadgen> python.exe .\beamng_wrapper.py
Init BeamNG
6:17 PM
7/11/2018

ASFAULT


Java – Example/src/example/Foo.java – Eclipse Platform
Quick Access
Resource
Java
Package Ex
JUnit
Example
src
example
Foo.java
GenericParameter.java
Stack.java
JRE System Library [JavaSE-1.6]
JUnit 4
Referenced Libraries
Foo.java
package example;
public class Foo {
private int x = 0;
private String str;
private String str2="bar";
public Foo(String string) {
this.str = string;
}
public void inc() {
x++;
}
public boolean coverMe() {
if (x==5)
if(!str.equals(str2))
if (str.equalsIgnoreCase(str2))
return true;
return false;
}
}
example.Foo.java – Example/src

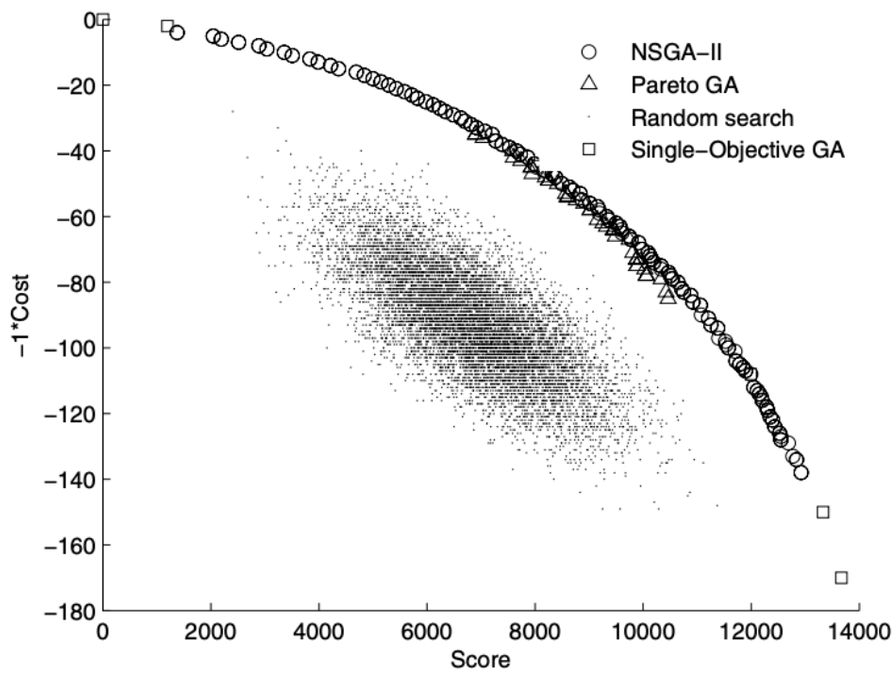
EVOSUITE

Requirements Engineering

- Next Release Problem: given cost and benefit (expected revenue) for each feature, what is the best subset of features to be released for budget B?

- 0-1 Knapsack (NP-complete)

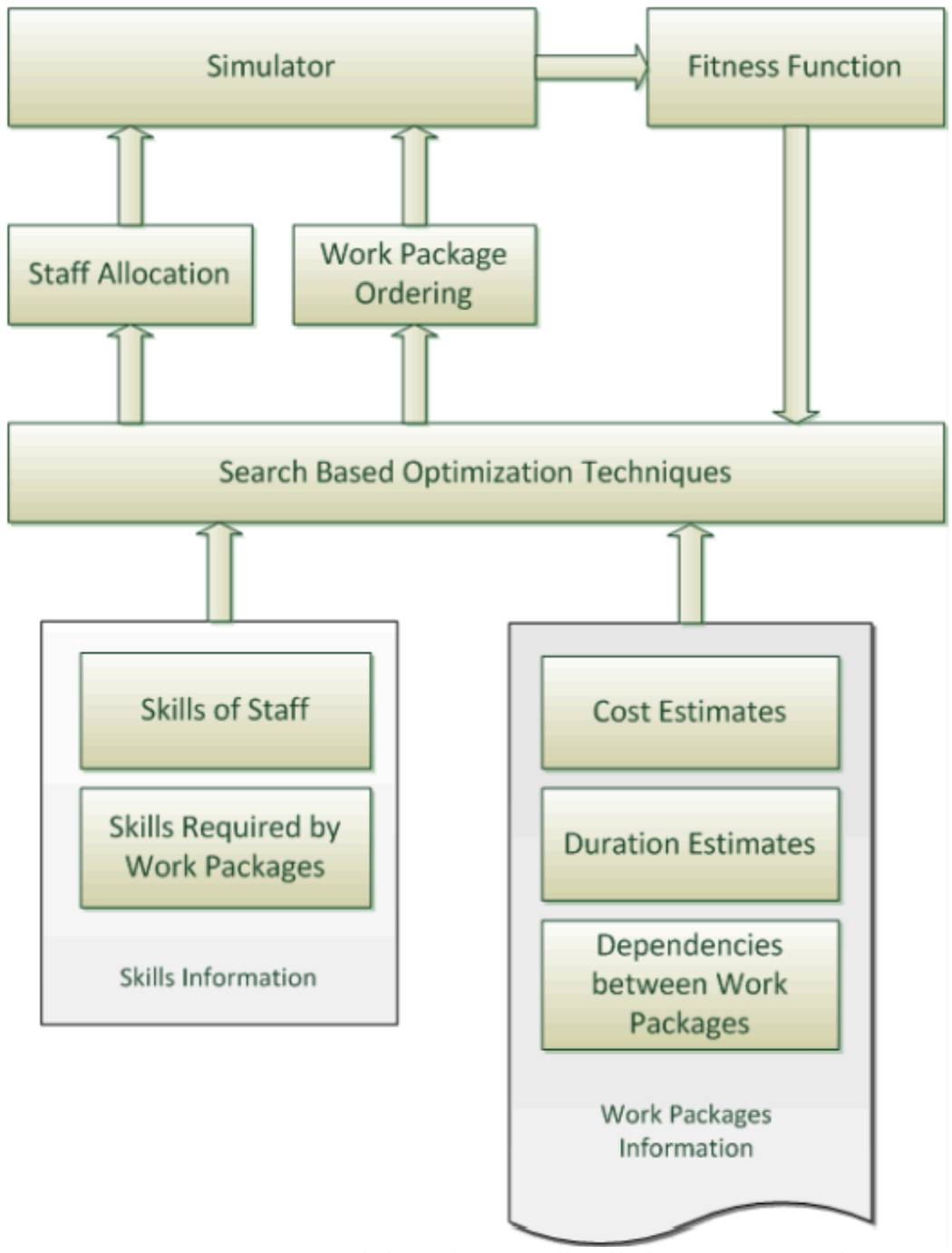
- But release decisions are more political than NP-complete



Project Management & Planning

- Quantitatively simulate and measure the communication overhead (linear? logarithmic?)

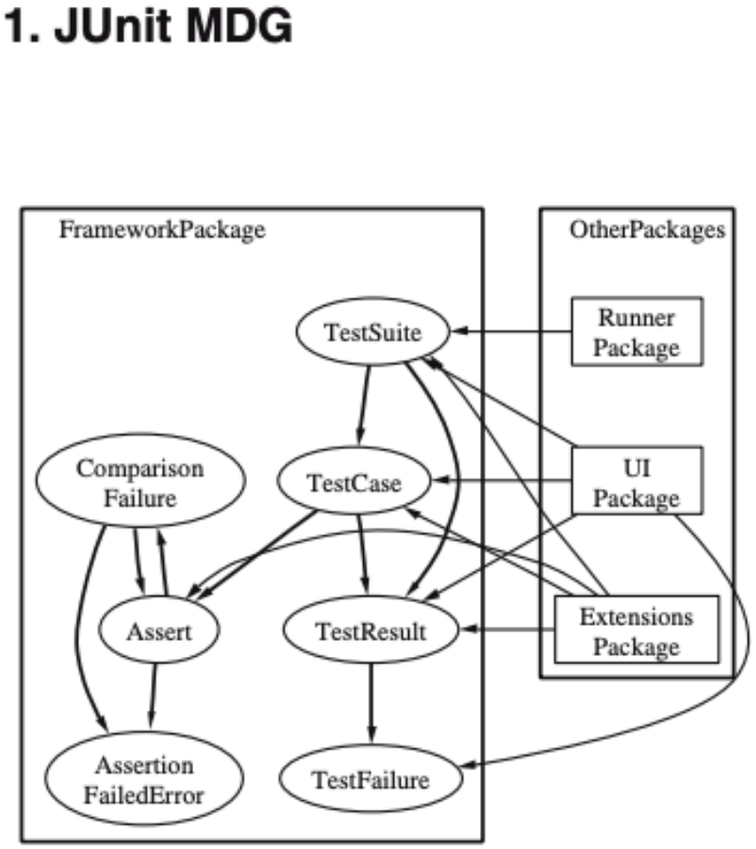
- Robust planning: search for the tradeoff between overrun risk, project duration, and amount of overtime assignment



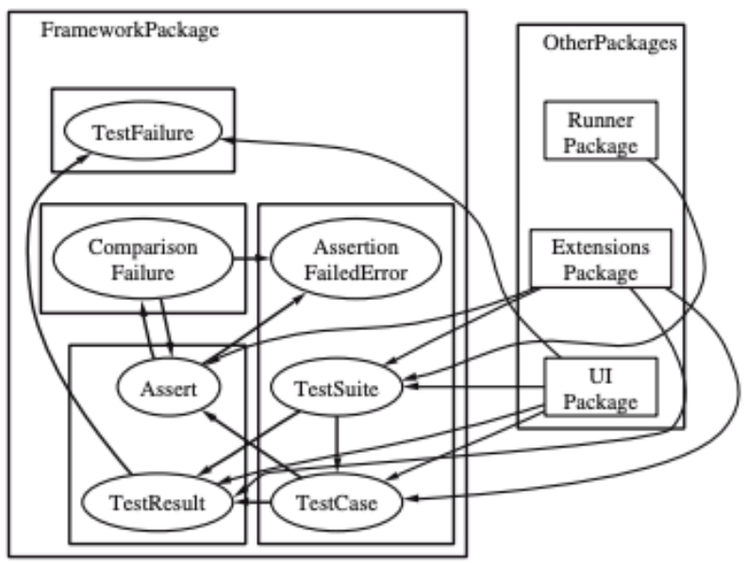
Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, *45*(1), 11.

Design/Architecture/Refactoring

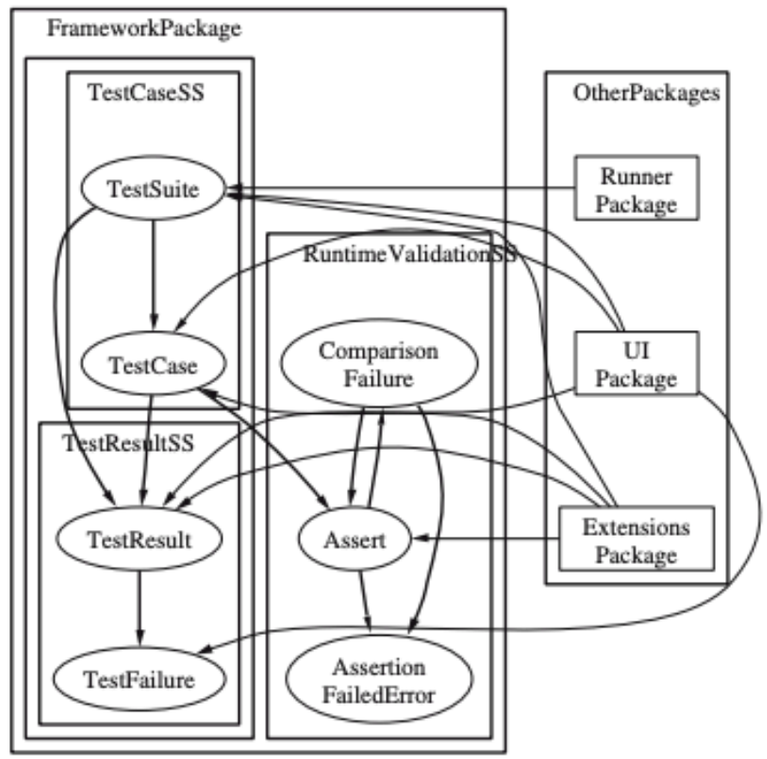
Cluster software models to achieve certain structural properties (cohesion/coupling).



1. JUnit MDG



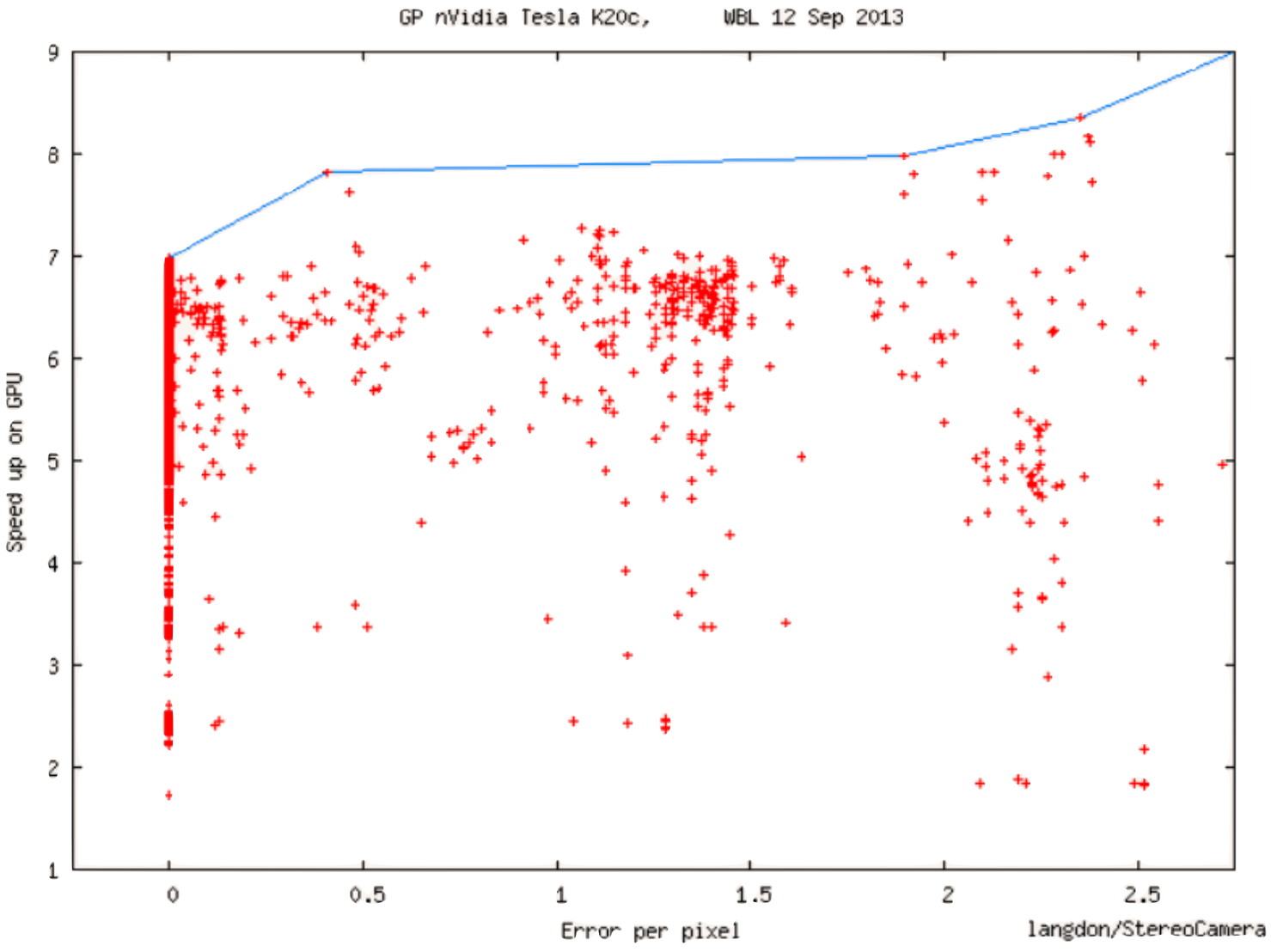
2. A Random Partition of the JUnit MDG

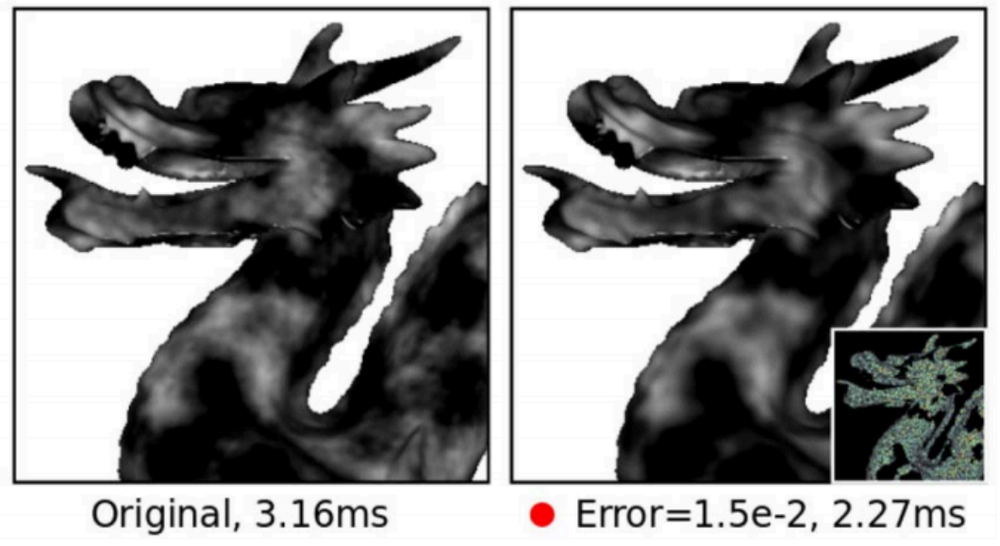


3. JUnit's MDG After Clustering

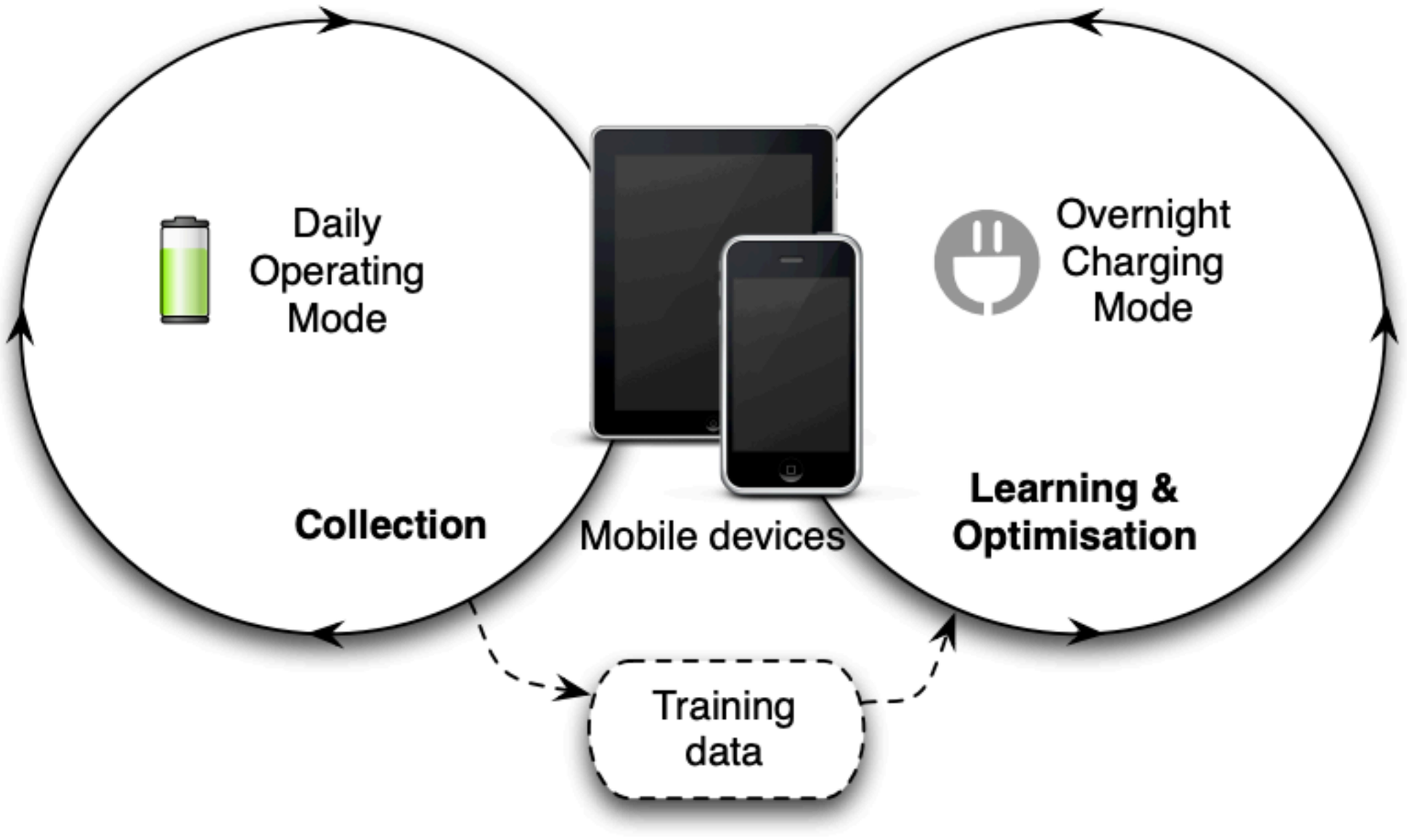
Genetic Improvement

- Given a source code, can we automatically improve its non-functional properties (such as speed)?





Genetic Improvement



Code Transplantation

- Software X has feature A, which you want to have in software Y. Can we automatically extract and transplant feature A from X to Y?

- *E.T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke. Automated software transplantation. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pages 257–269.*

yue-ucl-dell:1 (yuejia)

Buddy List

Buddies Accounts Tools Help

Buddies

- Daniel Lam
- Federica Sarro
- Michael Epitropakis
- Shoaib A
- Yang Li
- yuanyuanplus
- yueandalex@gmail.com

Available

yueplus

yueplus

Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system

SSBSE2014 Challenge

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Course Content

(Tentative Schedule)

15.10.: Introduction; Search Spaces and Fitness Landscapes

22.10.: Local Search Algorithms

29.10.: <no lecture>

05.11.: Local Search Algorithms

12.11.: Evolutionary Search

19.11.: Evolutionary Search

26.11.: Multi-Objective Optimisation

03.12.: Multi-Objective Optimisation

10.12.: Search-based testing

17.12.: Search-based testing

07.01.: Genetic programming

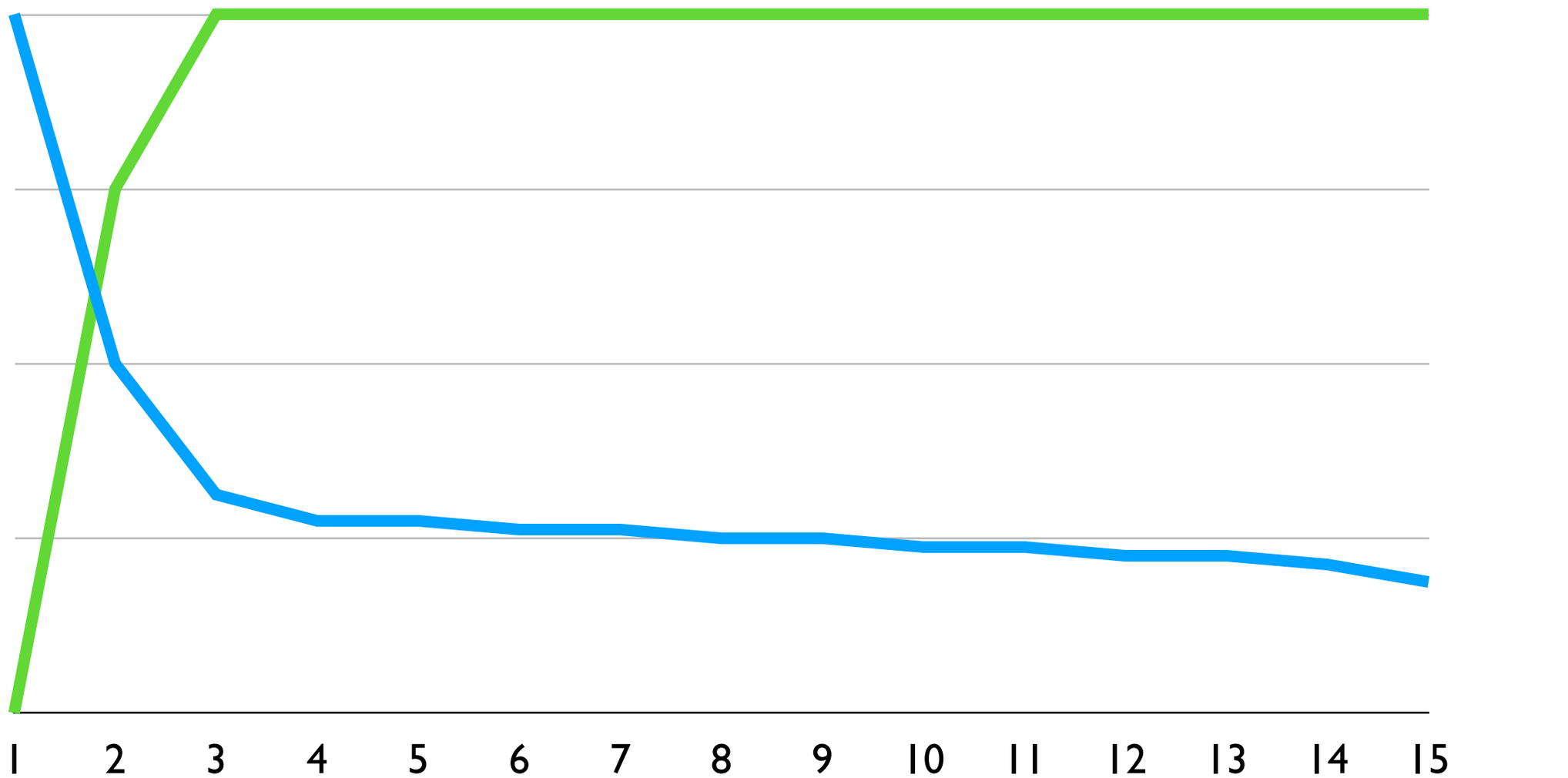
14.01.: Genetic programming

21.01.: Advanced Evolutionary Algorithms

28.01.: Neuroevolution

04.02.: Neuroevolution

Course Material



Grading

- Four assignments:
    - Each assignment requires implementing a search-based solution for a software engineering problem
        - Individual work, not group work
        - Assignment 1—3 Java. Assignment 4 Python (tentative)
- Grade is based on portfolio of assignment solutions
- Each assignment 25% of grade
- Optional warm-up task

Exercises

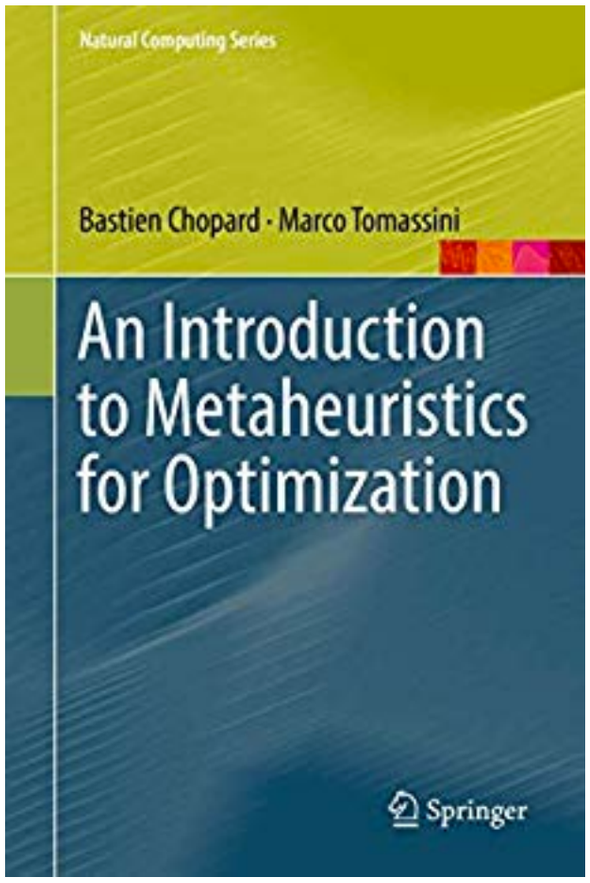
- First exercise: 16.10.
- Tentative plan for assignments:
  - Assignment 1 release 13.11.
  - Assignment 2 release 04.12.
  - Assignment 3 release 08.01.
  - Assignment 4 release 29.01.

Patric Feldmeier

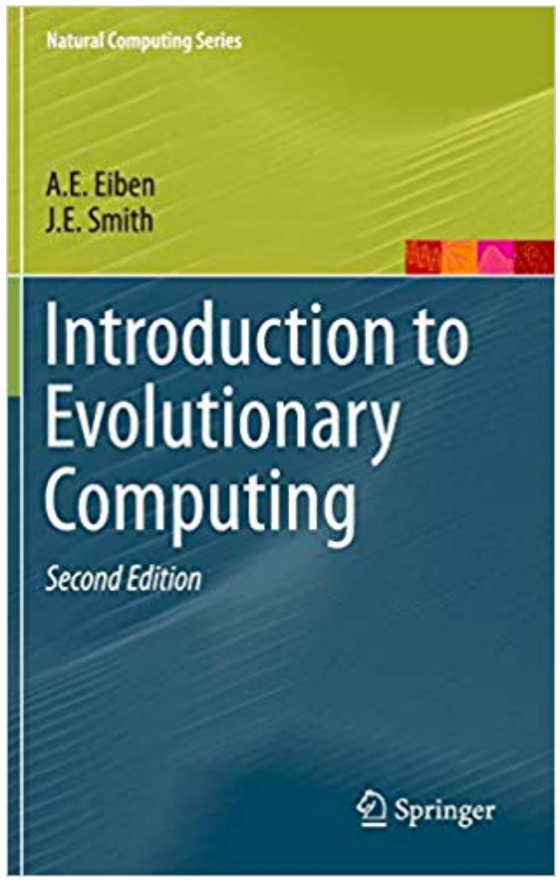
Registration

- 14.10.2024 - 04.11.2024
- …but assignment 1 release 13.11.
- → optional introductory task!

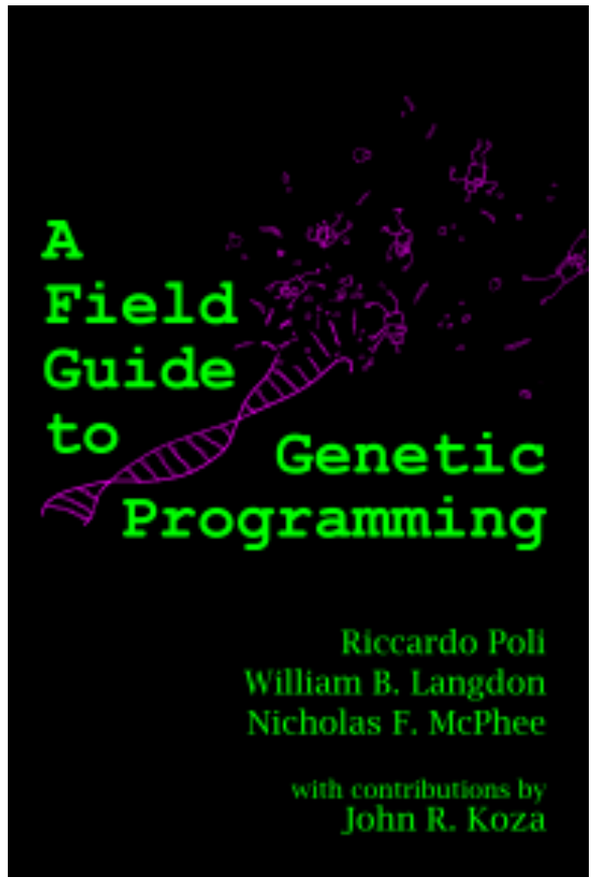
Literature



An Introduction to Metaheuristics for Optimization, B. Chopard, M. Tomassini, Springer



Introduction to Evolutionary Computing, A. E. Eiben, J. E. Smith, Springer



A Field Guide to Genetic Programming,
R. Poli, W. B. Langdon, & N. McPhee
(freely available online at
http://www.gp-field-guide.org.uk)

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes

Search Space

- Example problem:
  Find point in the $x = (x_1, x_2)$ plane that minimises $f(x) = x_1^2 + x_2^2$, subject to the constraint $x_1 + x_2 = 3$.

- Search space S:
  All $x \in S$ are admissible solutions (i.e. satisfy constraints)

- Optimal solutions: $f(x) \leq f(y), \forall y \in S$

- Size of optimisation problem: degrees of freedom

- Search space size $|S|$: Number of solutions in S

- If $|S|$ is infinite or uncountable, we have a continuous optimisation problem

- Combinatorial optimisation problem has a discrete, countable search space

Fitness Landscape

- Fitness landscape is a representation of *f* that preserves the neighbourhood topology
  = Search space with a fitness value for each configuration and a neighbourhood relationship.

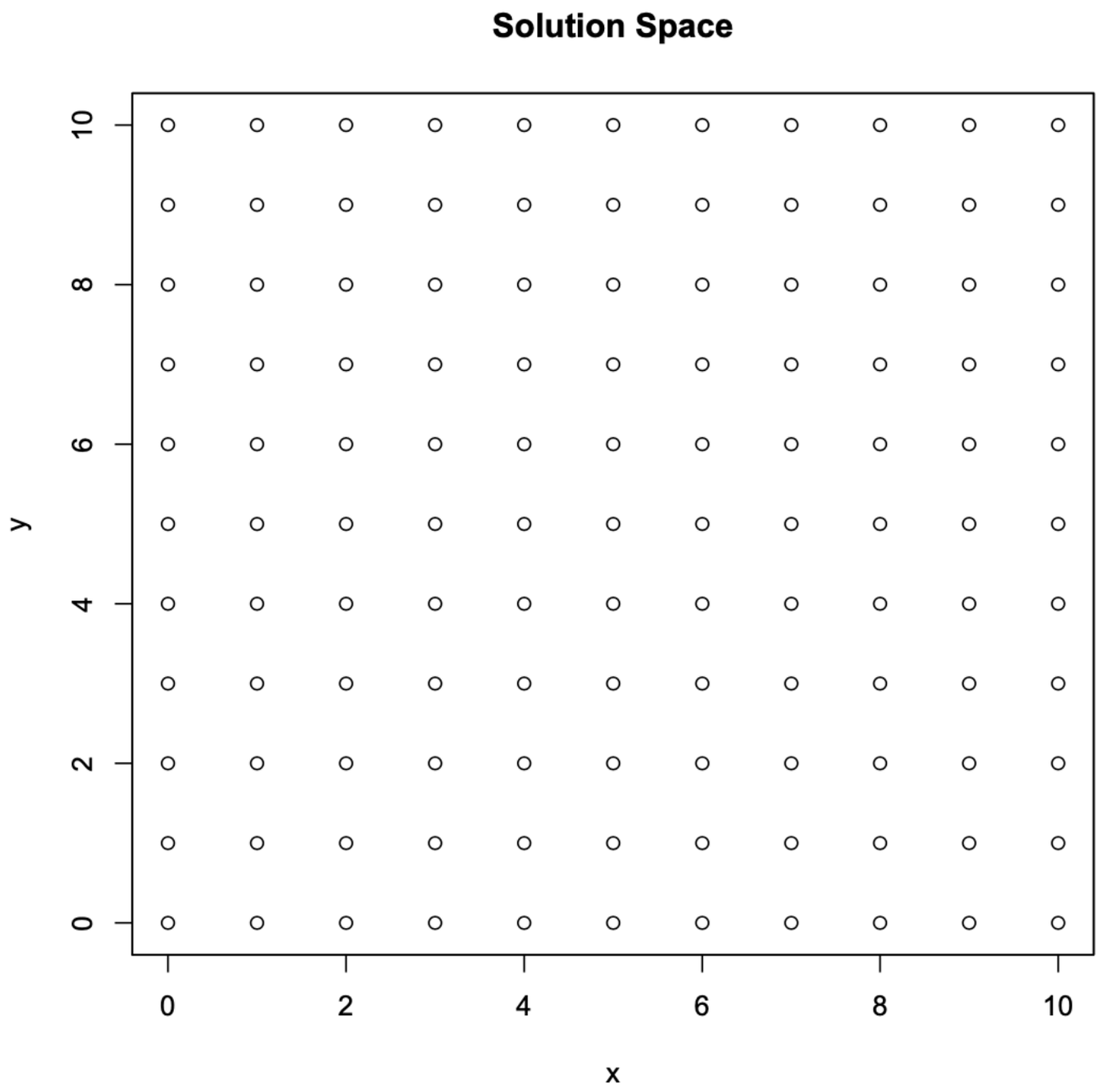
- Given a solution space S (a hyperplane), and a fitness function F, a fitness landscape is a hyper dimensional surface that represents $F: S \rightarrow \mathbb{R}$

- Optimum = points at which all first-order partial derivatives vanish ($\nabla F = 0$)

- Usually impossible to visualise.

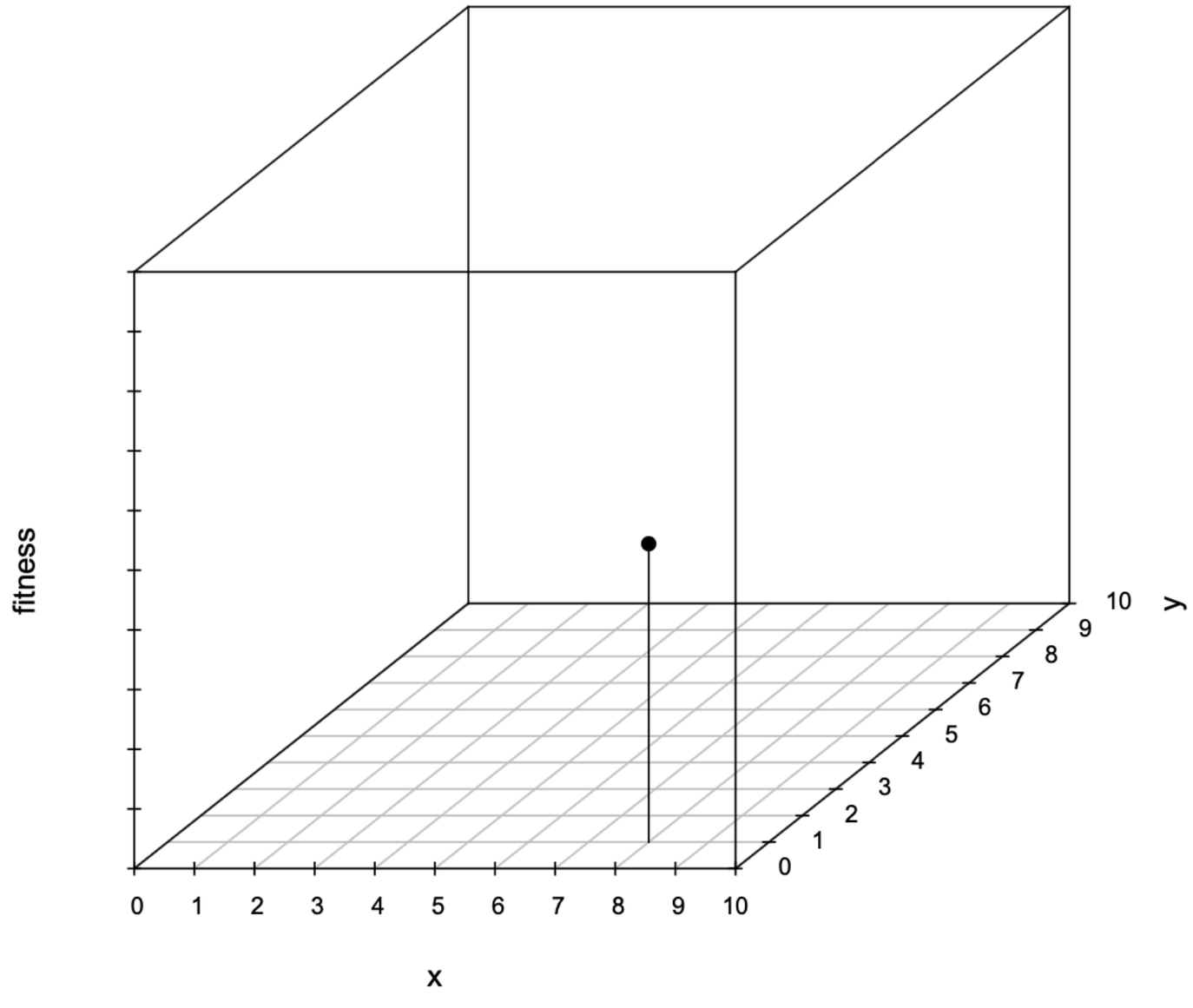
(Artificial) Example Problem

Given $0 \leq x \leq 10, 0 \leq y \leq 10$, find (x, y) such that $x + y = 10$.



(Artificial) Example Problem

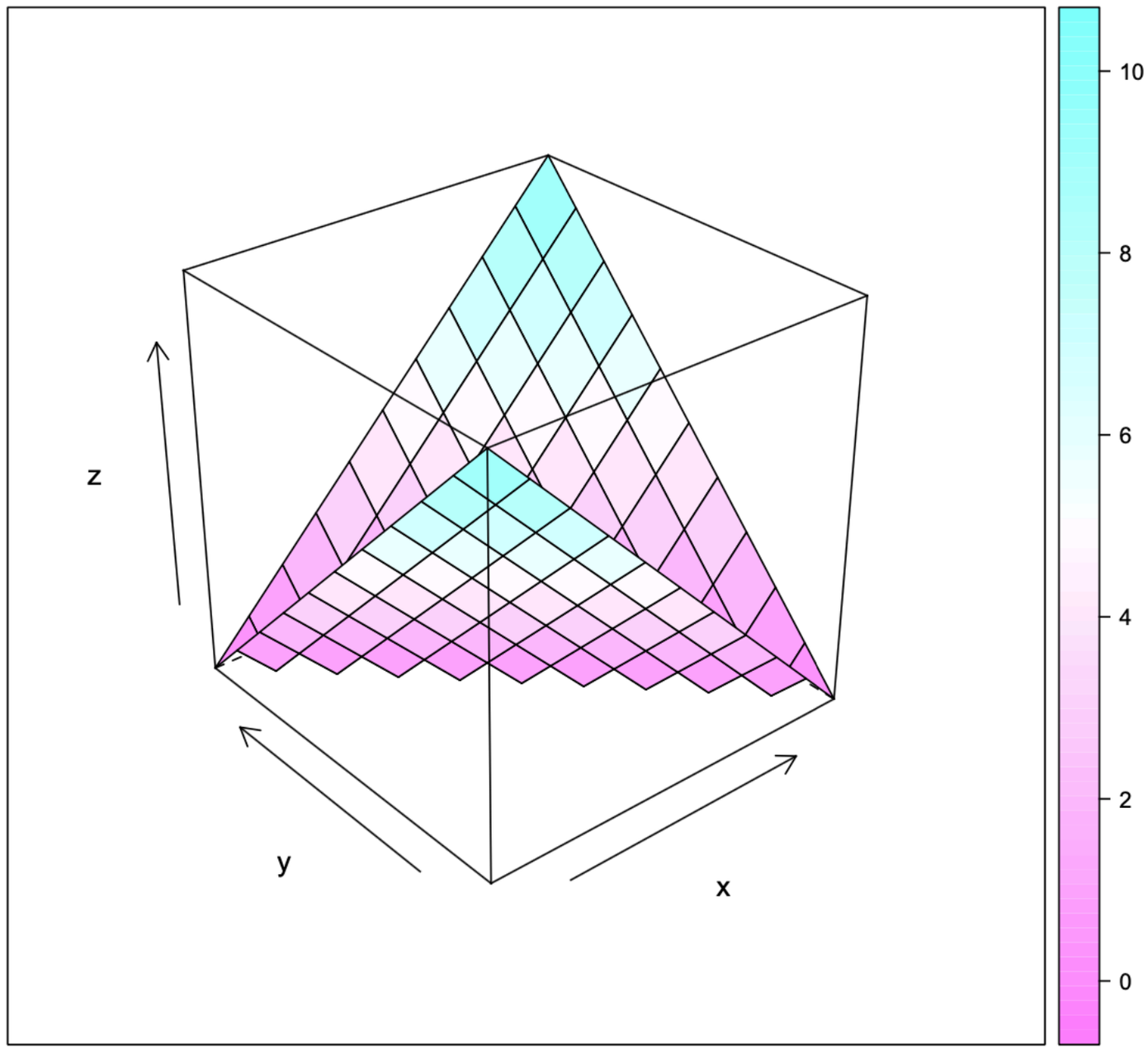
Individual point in fitness landscape

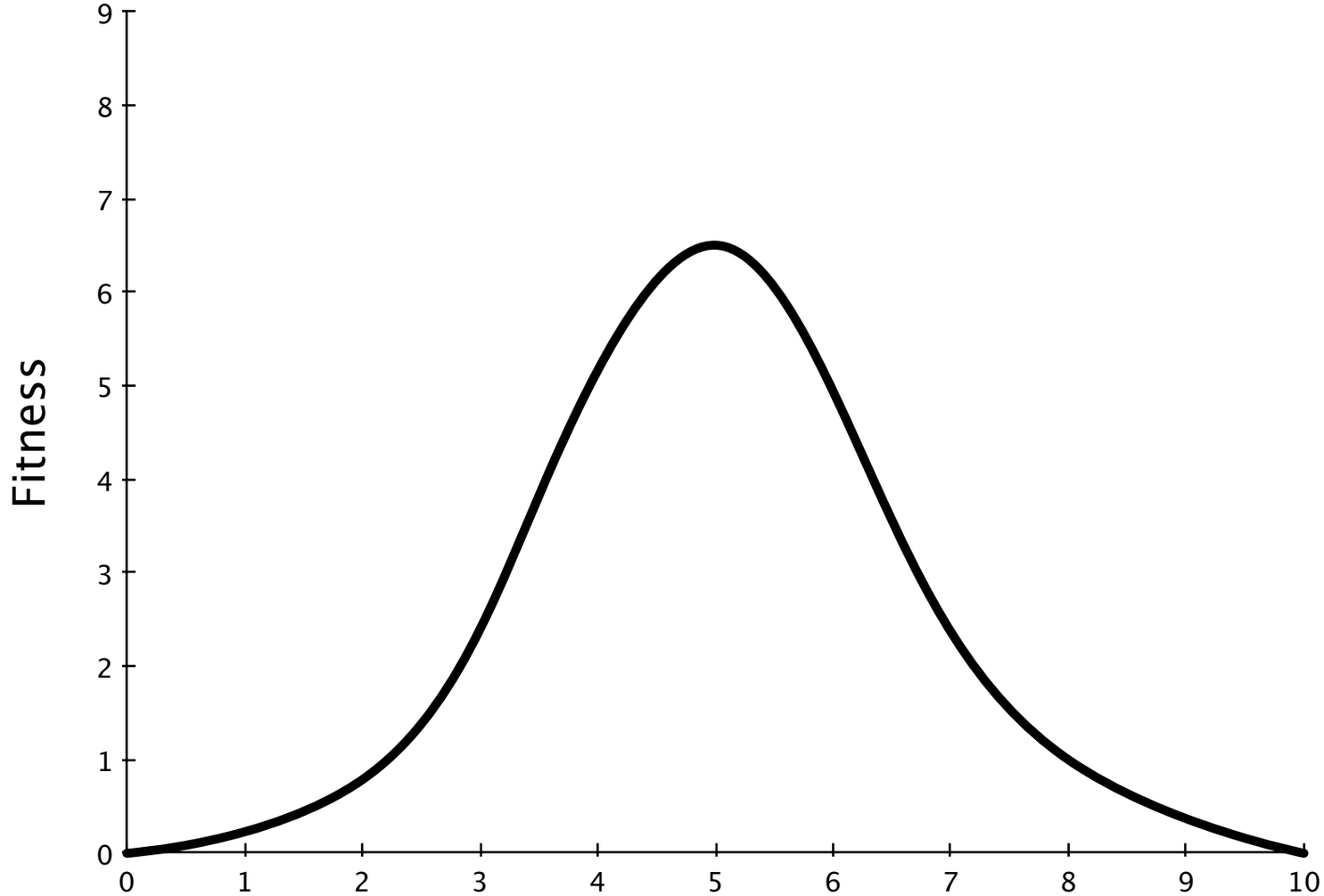


(Artificial) Example Problem

- 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 is 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

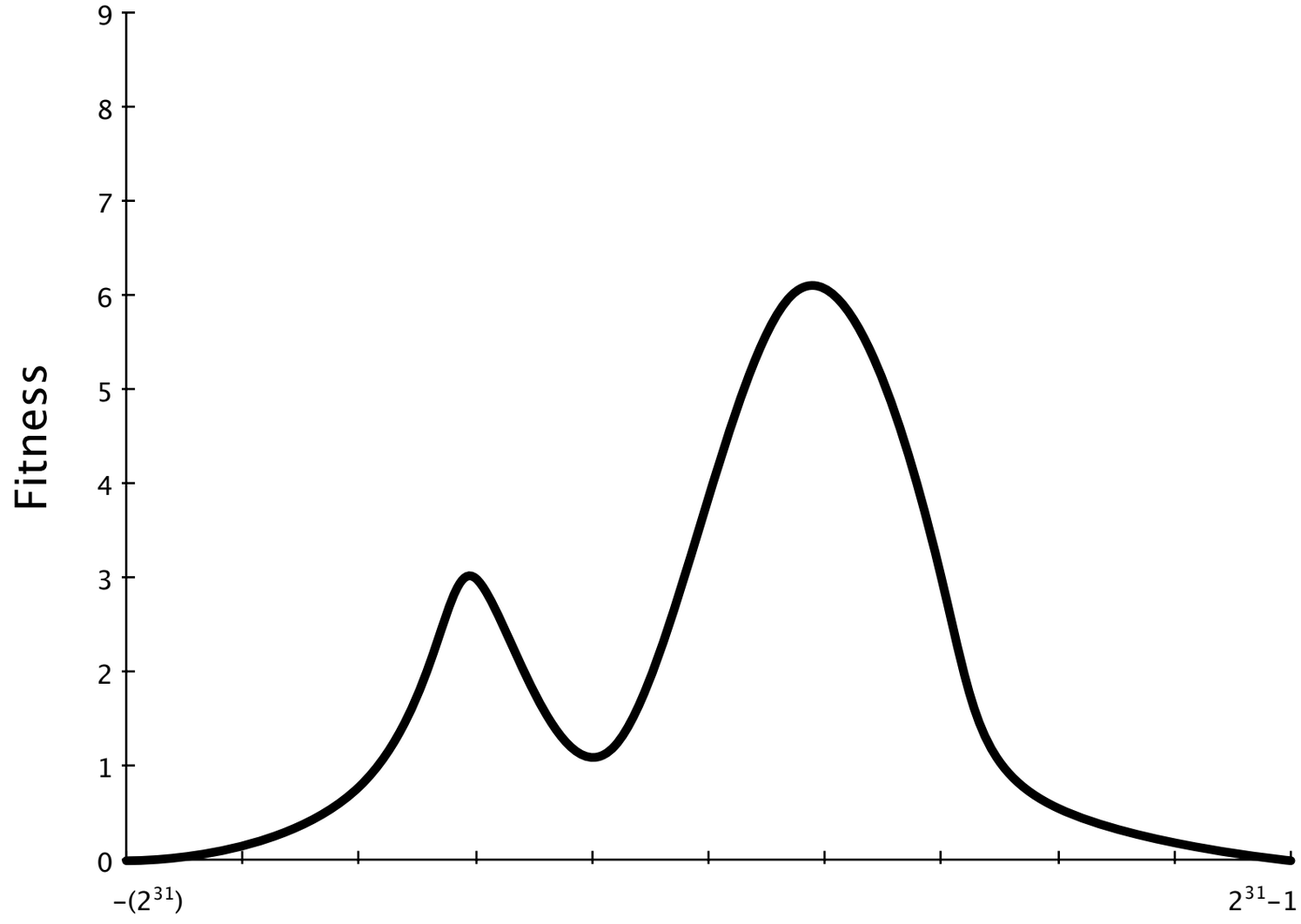
(Artificial) Example Problem





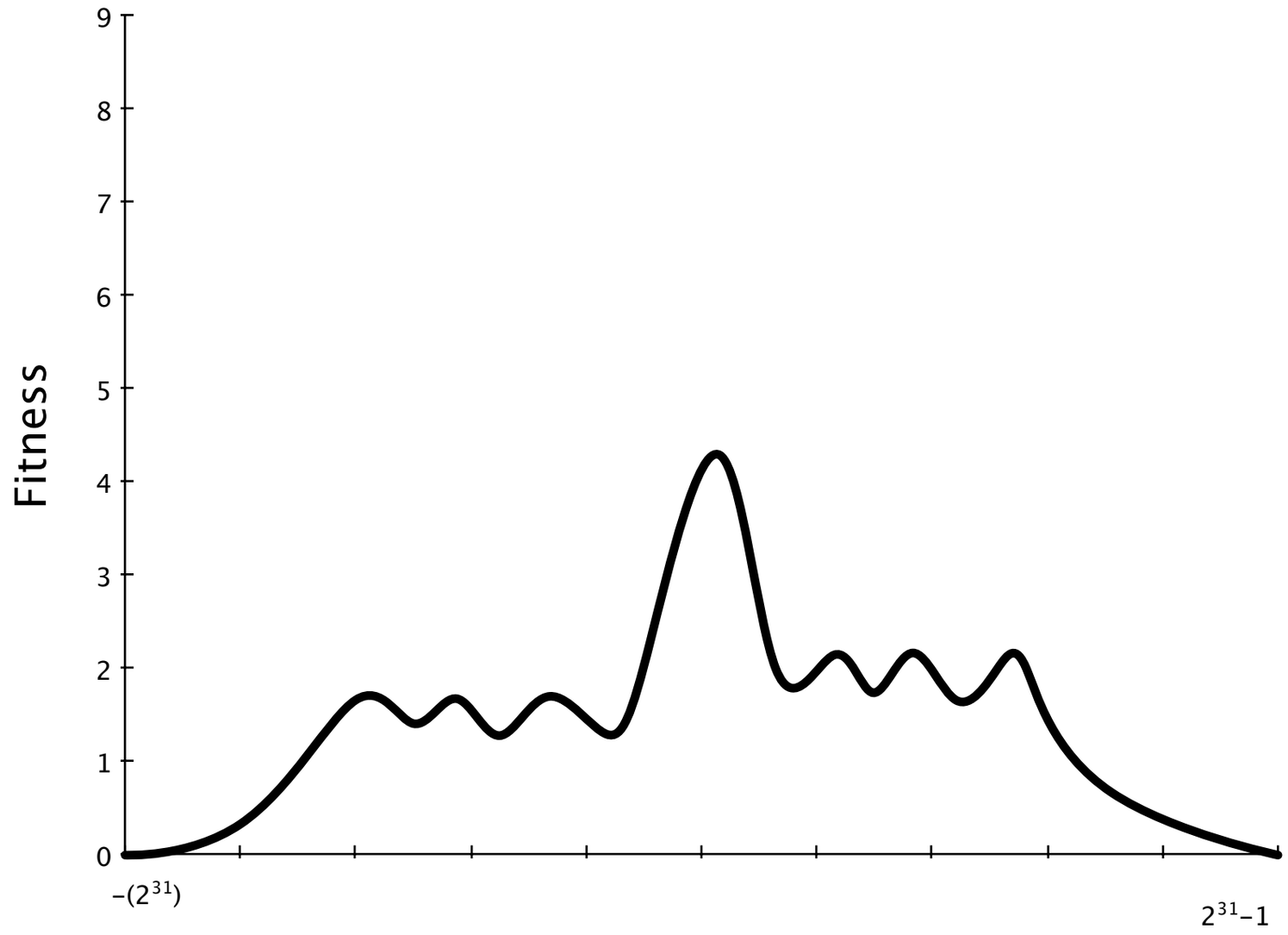
$f: \mathbb{R} \rightarrow \mathbb{R}$

Single-peak or few-peak landscapes are easy



$f: \mathbb{R} \rightarrow \mathbb{R}$

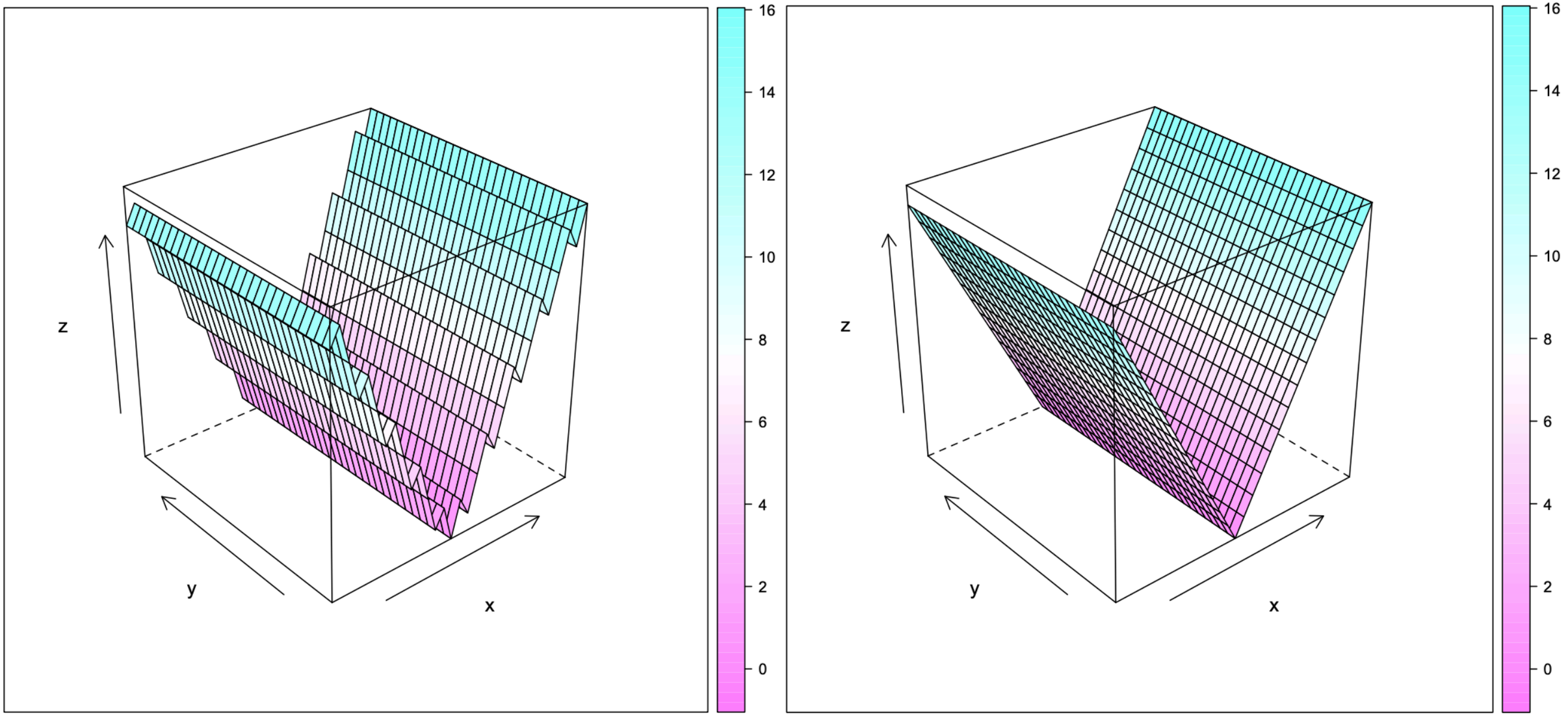
Local optimum



$f: \mathbb{R} \rightarrow \mathbb{R}$

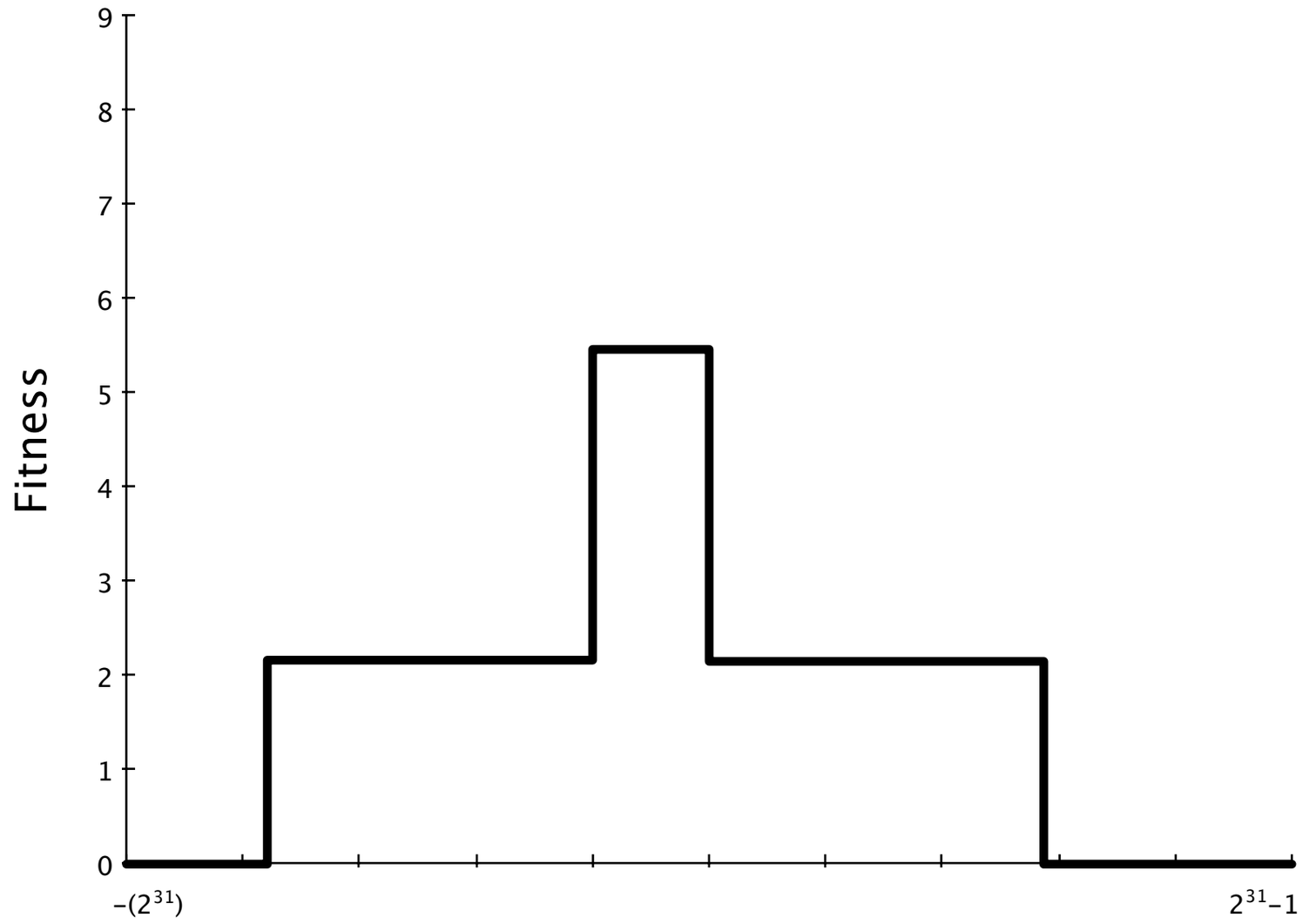
Very rugged landscapes are difficult

Ruggedness



Easy to get stuck
in one of many local optima

Smooth descent

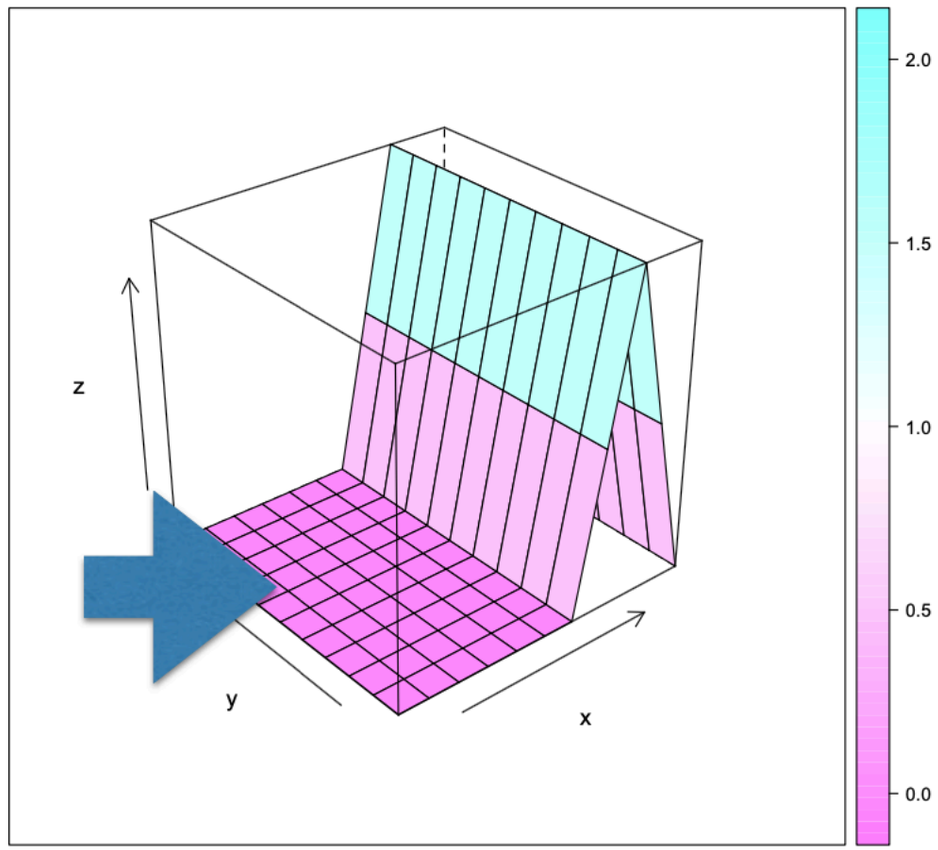


$f: \mathbb{R} \rightarrow \mathbb{R}$

Plateaus: Very flat landscapes are also difficult

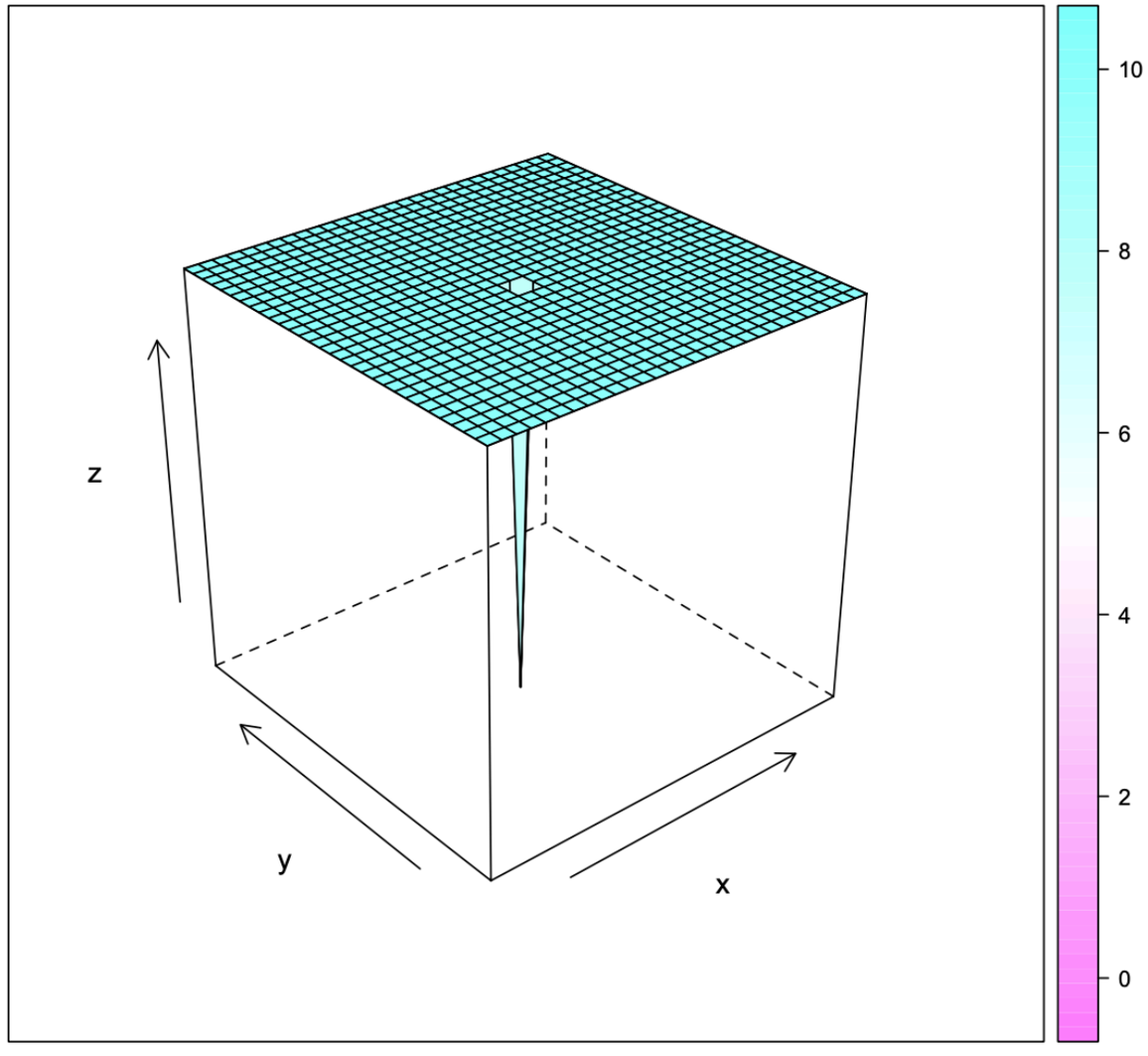
Plateau

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



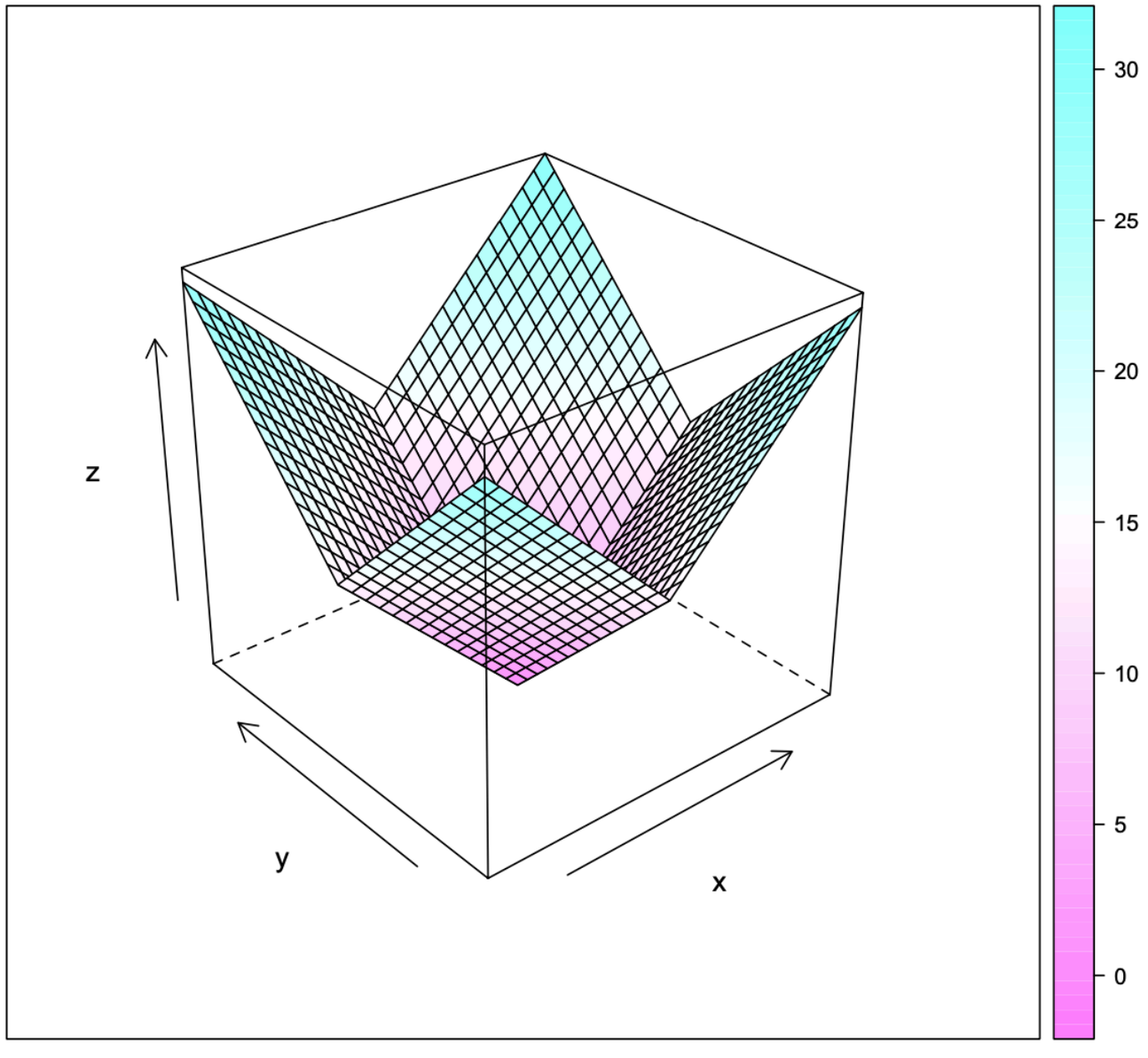
Needle in the Haystack

- 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|



Metaheuristics

- Makes no hypothesis on the mathematical properties of the objective function such as continuity or derivability. The only requirement is that $f(x)$ can be computed for all $x \in S$.

- They have a few parameters that influence the quality of the solutions found and speed of convergence, but the optimal values of the parameters are generally unknown.

- A starting point for the search process must be specified. Usually the initial solution is chosen randomly.

- A stopping condition must also be built into the search. This is usually based on CPU time or number of evaluations.

- They are generally easy to implement and can usually be parallelised efficiently.

- Balances exploitation and exploration: Exploitation explores the neighbourhood of an already promising solution in the search space; exploration tries to visit regions of the search space not already seen.

Random Search

- Next point in the search is chosen uniformly at random in the whole search space S
- Solution having the best fitness is kept
- Probability of finding global optimum is small, if *S* is large (*1*/|*S*|)
- Very easy to implement, inherently automatable, no bias at all.
- Random search should always be the default sanity check against your own search methodology: if it does not do better than random search, you are doing something wrong.
- 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 a program crash"
  - In general, given an arbitrary program, you cannot measure the distance between the current program state and a crash!

Example: Max One

- Maximise: $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$
- Obvious solution: $x=(1,1,\ldots,1)$.
- (Coding example)

Local Search Algorithms

- Common algorithmic principle:

  1. For a given problem we define a search space S and an objective function $f: S \rightarrow R$

  2. Neighbourhood $V(x)$ for $x \in S$: Set of solutions y that one can reach from x in one step.

     - Usually defined implicitly by a set of transformations T_i that generate y starting from x: if y in $V(x)$ then there is an i for which $y = T_i(x)$. (Transformations = "moves")

  3. Exploration operator U: Applied on a current solution x_0 generates the next point to explore in the search trajectory. Operator U makes use of the fitness values in the neighbourhood to generate the next solution.

  4. Exploration process $x_0 \rightarrow x_1 \in V(x) \rightarrow x_2 \in V(x_1) \rightarrow \ldots$ continues until a stopping criterion is met.

- The result is either the last x_n in the trajectory, or the x_i found along the trajectory that produces the best fitness value.

- Efficiency depends on the choice of $V(x)$, coding of solutions, choice of U, etc.

Contents

- Solving Optimisation Problems
- Search-Based Software Engineering
- Course Contents and Organisation
- Search Spaces and Fitness Landscapes