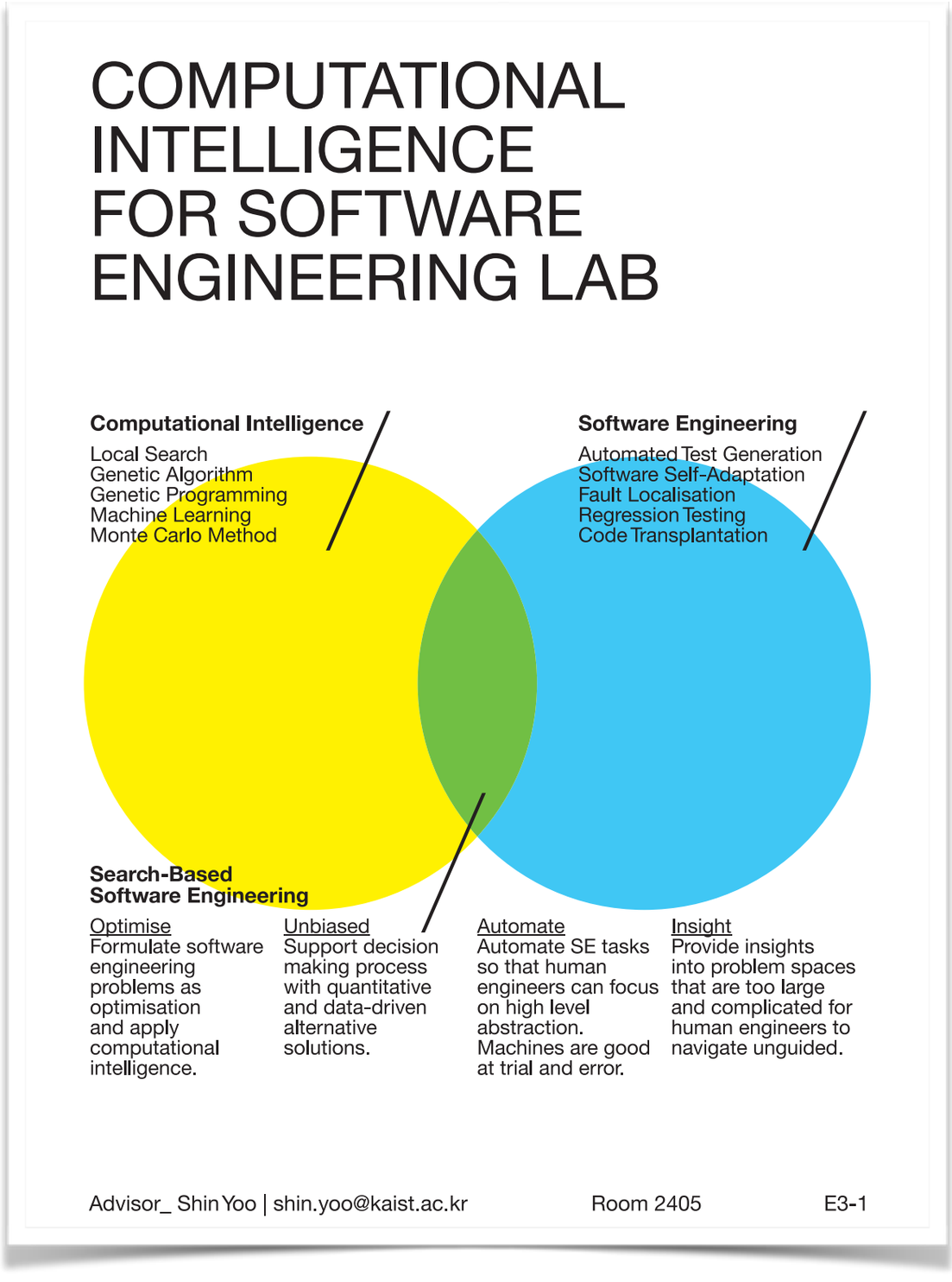
Introduction to SBSE

Shin Yoo
School of Computing, KAIST

Me

- Shin Yoo, joined KAIST CS in August 2015
  - PhD at King's College London, UK
  - Assistant Professor at University College London, UK
- COINSE (Computational Intelligence for Software Engineering) Lab
- Research interest: SBSE, regression testing, automated debugging, evolutionary computation, information theory, program analysis…
- shin.yoo@kaist.ac.kr



What is SBSE?

(wait, before that…)

What is SE?

Software Crisis

- 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

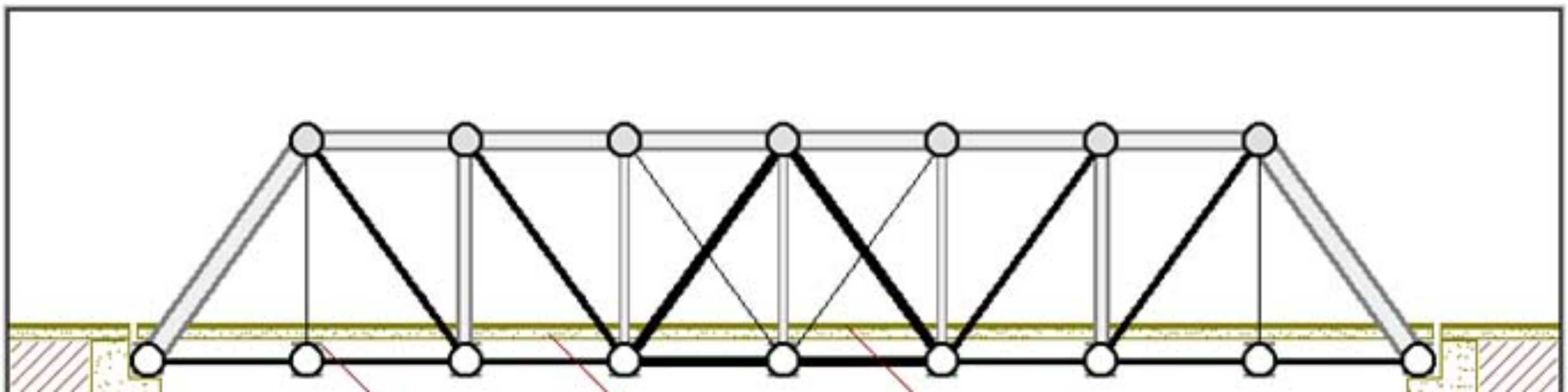
NATO conference 1968

- …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

Let’s say we want to build a steel bridge.



Theory

- When does a steel beam break?
- Stress: force per unit area
- Tensile strength: the maximum stress a material can resist

Stress is the force **per** unit area.

$$\text{stress} = \frac{\text{force}}{\text{area}}$$

hat does stress mean?

Stress allows us to get a fair comparison of the effects of a force on different samples of a material. A tensile force will stretch and, possibly, break the sample. However, the force needed to break a sample will vary - depending on the cross sectional area of the sample. If the cross sectional area is bigger, the breaking force will be bigger. However, the breaking *stress* will always be the same because the stress is the force per unit area.

In picture 4.2, sample A breaks with a force of 60 kN. Sample B breaks with a force of 120 kN because it has twice the area (you can think of it as being two pieces of sample A next to each other, with each one needing a force of 60 KN to break it). Although the force is bigger for sample B, the stress is the same for both samples – it is 60 kN cm^{-2}.

So breaking **stress** is a more useful measurement than breaking **force** because it is constant f a given material. It allows us to fairly compare the strengths of different materials.

For example:
The force needed to break a piece of steel wire with a cross sectional area of 2×10^{-6} m^2 is 2400 N.
a) What is its breaking stress?
b) What force would be needed to break a steel bar with a cross section of 5×10^{-4} m^2?

a) The breaking stress = breaking force ÷ area
= 2400 ÷ 2×10^{-6}
= 1,200,000,000 N m^{-2}
= 1.2 GPa.

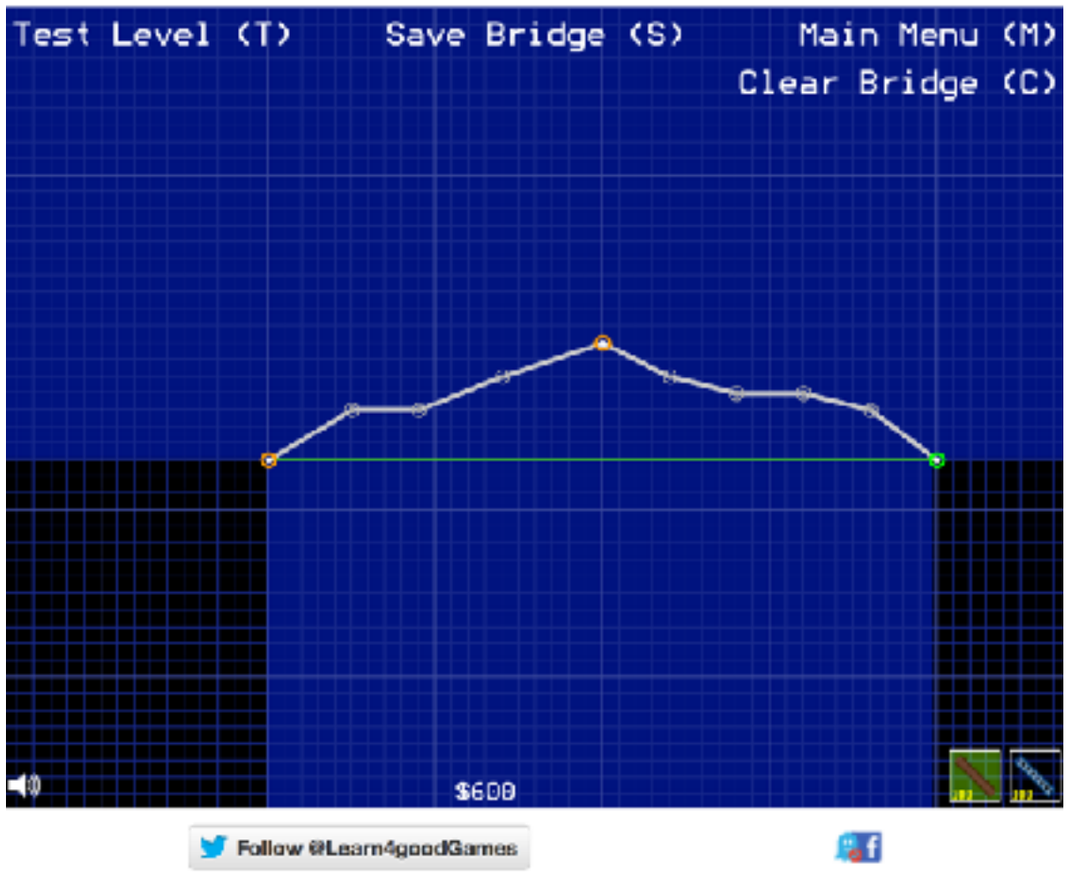
b) To break the steel bar, the force needed = breaking stress x area
= $1.2 \times 10^{9} \times 5 \times 10^{-4}$
= 600,000 N

nsile strength

A tensile test is used to find out what happens when a material such as steel is stretched. A stee bar is placed in a device that pulls one end away from the other fixed end. The **tensile strengt** is the maximum stress that the bar can withstand before breaking.

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)

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

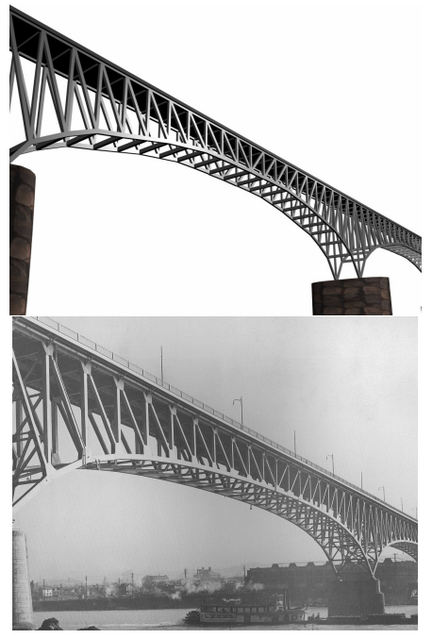


Figure 1: A cantilever bridge generated by our software, compared with the Homestead bridge in Pittsburgh, Pennsylvania.

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.

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-

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

Theory of software engineering is (somewhat) lacking.

Simulation is done via **modelling**.

Optimisation?

We are probably the only engineering principle, in which the materials for product, simulation, and optimisation are **the same**.

Search-Based Software Engineering

- 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)

Why optimisation?

- **Automate** SE tasks (either fully, or at least until human engineers can attend to the issue)

- Gain **insights** into complicated problem domain that are either too large or too complicated for humans to understand

- **Unbiased** decision support that is data-driven

Our Toolbox

- Classical (exact) optimisation would be desirable but often cannot cope with the scale

- A heavy focus on stochastic optimisation, with a heavy emphasis on evolutionary computation and other nature-inspired algorithms (mostly due to historical reasons).

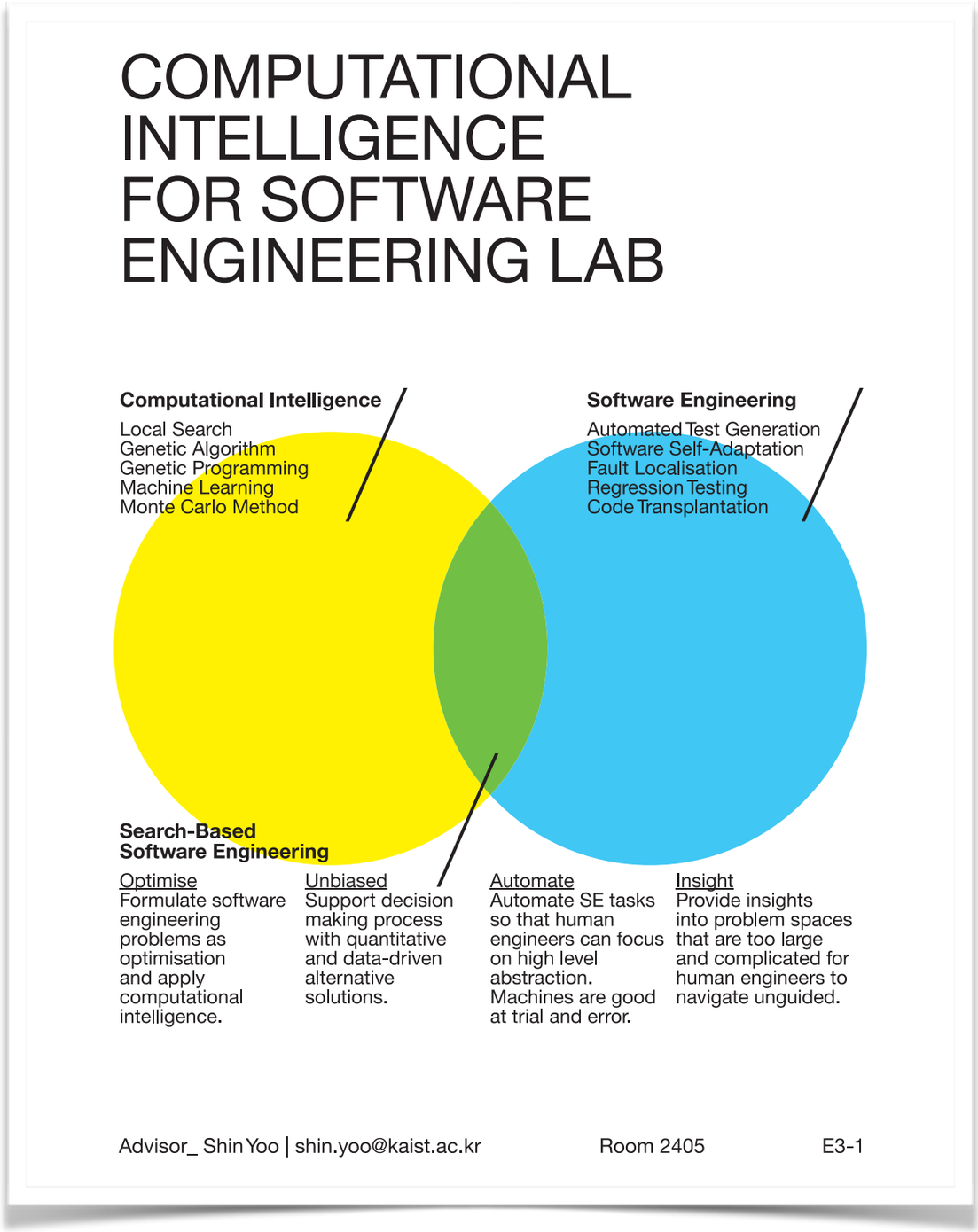
- But we are open to applications of any machine intelligence: neural nets and reinforcement learning are gaining much attention these days naturally.

Our Stance

- We stand at the intersection of computational intelligence and software engineering.

- Pragmatic application has stimulated theoretical results in computational intelligence, and vice versa.

- This lecture series will discuss a mixture of optimisation techniques and their applications in SE



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.*

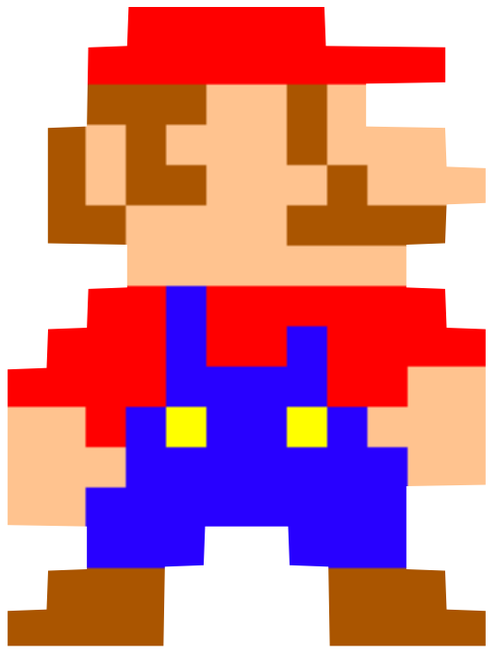
3 denoting something of a higher or second-order kind: *metalanguage* | *metonym.*

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.*

Meta-heuristic

- Strategies that guide the search of the acceptable solution
- Approximate and usually non-deterministic
- Not problem specific
- Smart trial and error



Let’s play Super Mario Bros.

http://arxiv.org/pdf/1203.1895v3.pdf

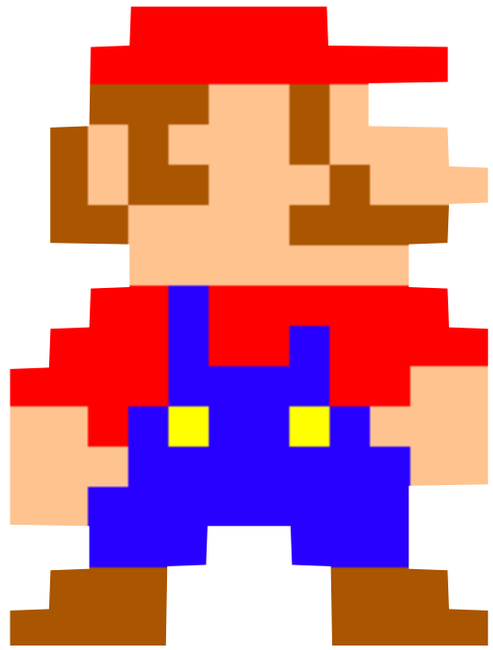
Classic Nintendo Games are (Computationally) Hard

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

February 10, 2015

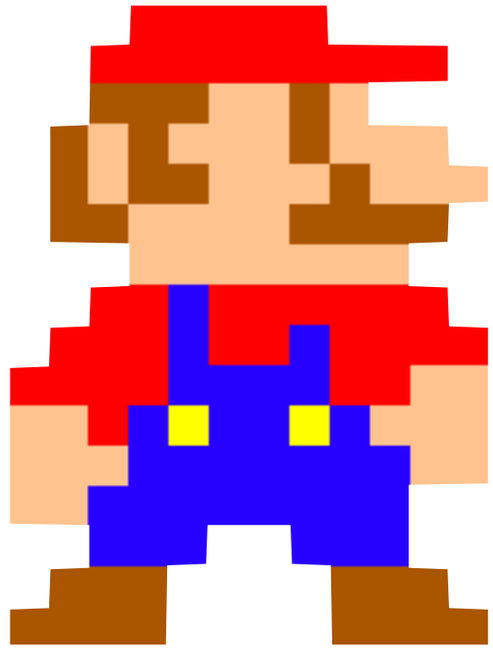
Player A

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



Player B

- Grab the controller.
- Play.
- Die.
- Repeat until level is cleared.



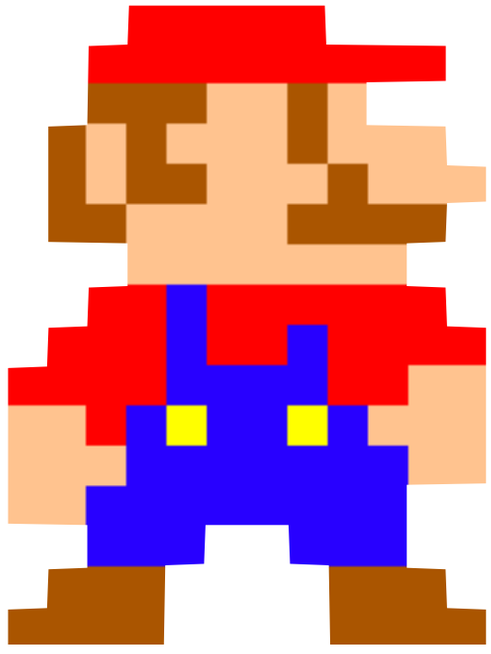
Intelligence lies in how differently you die next time.

Mario analogy goes a long way

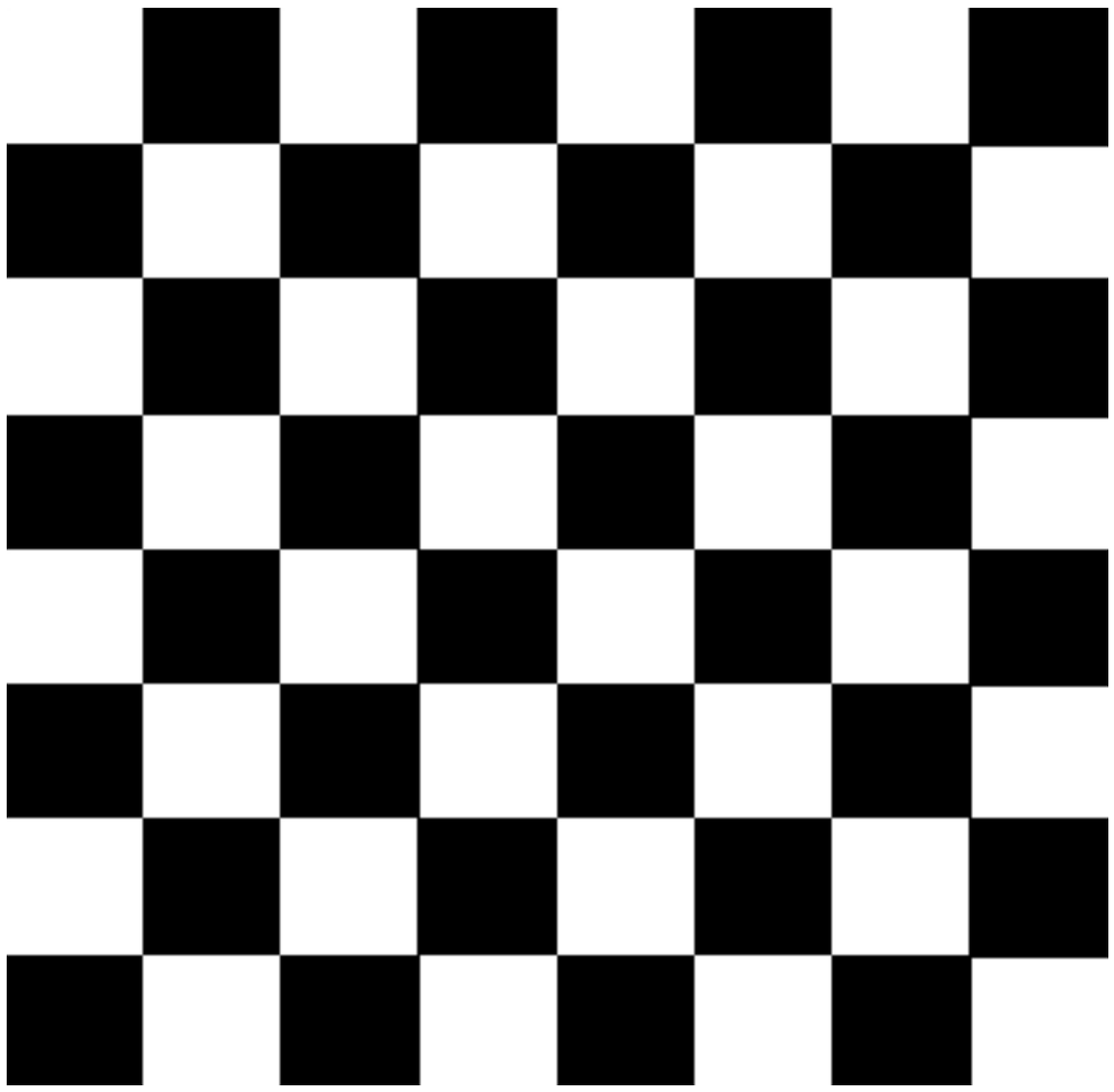
- You make small changes to your last attempt (“okay, I will press A slightly later this time”).

- 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?”)

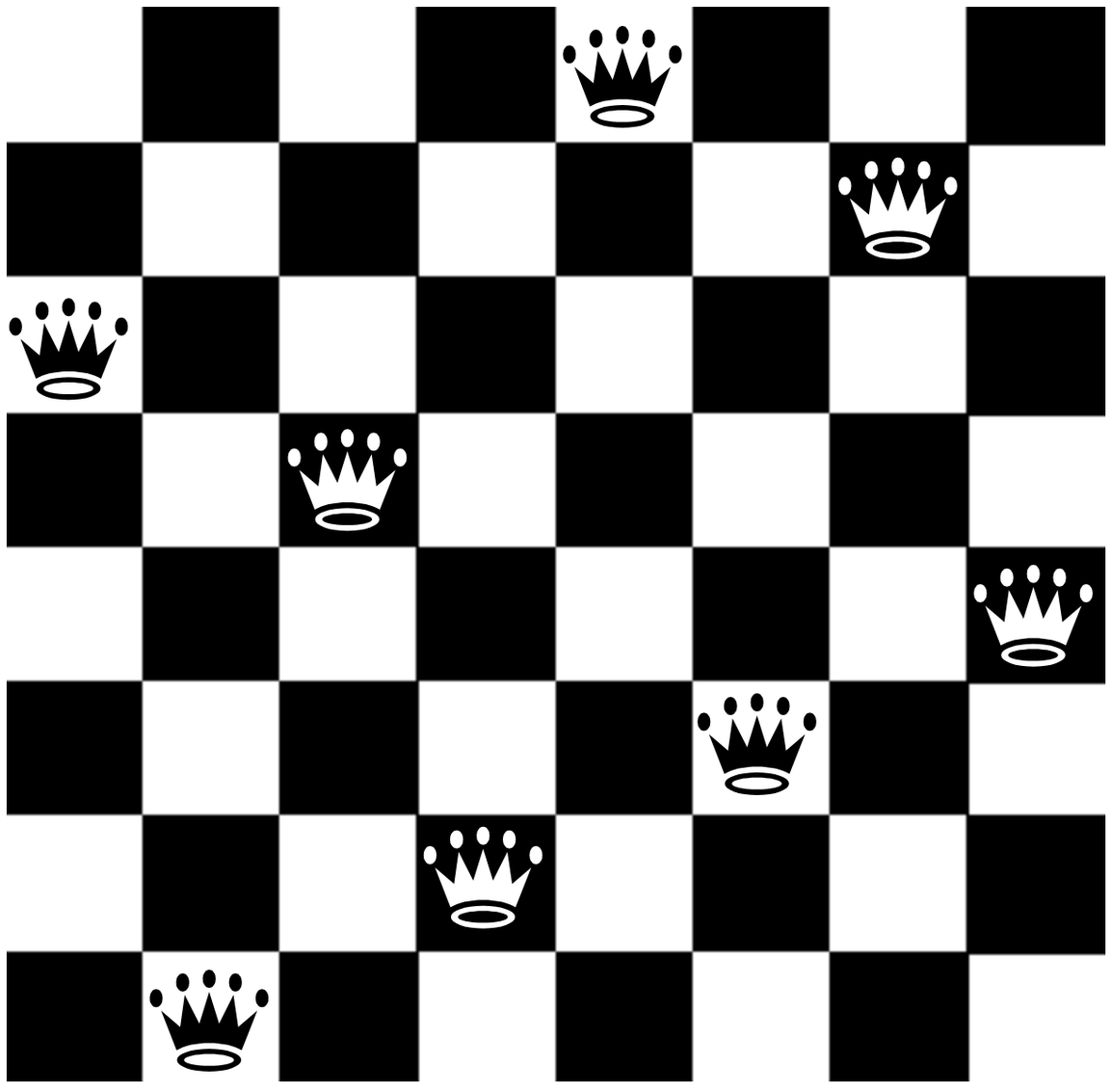


Eight Queens Problem



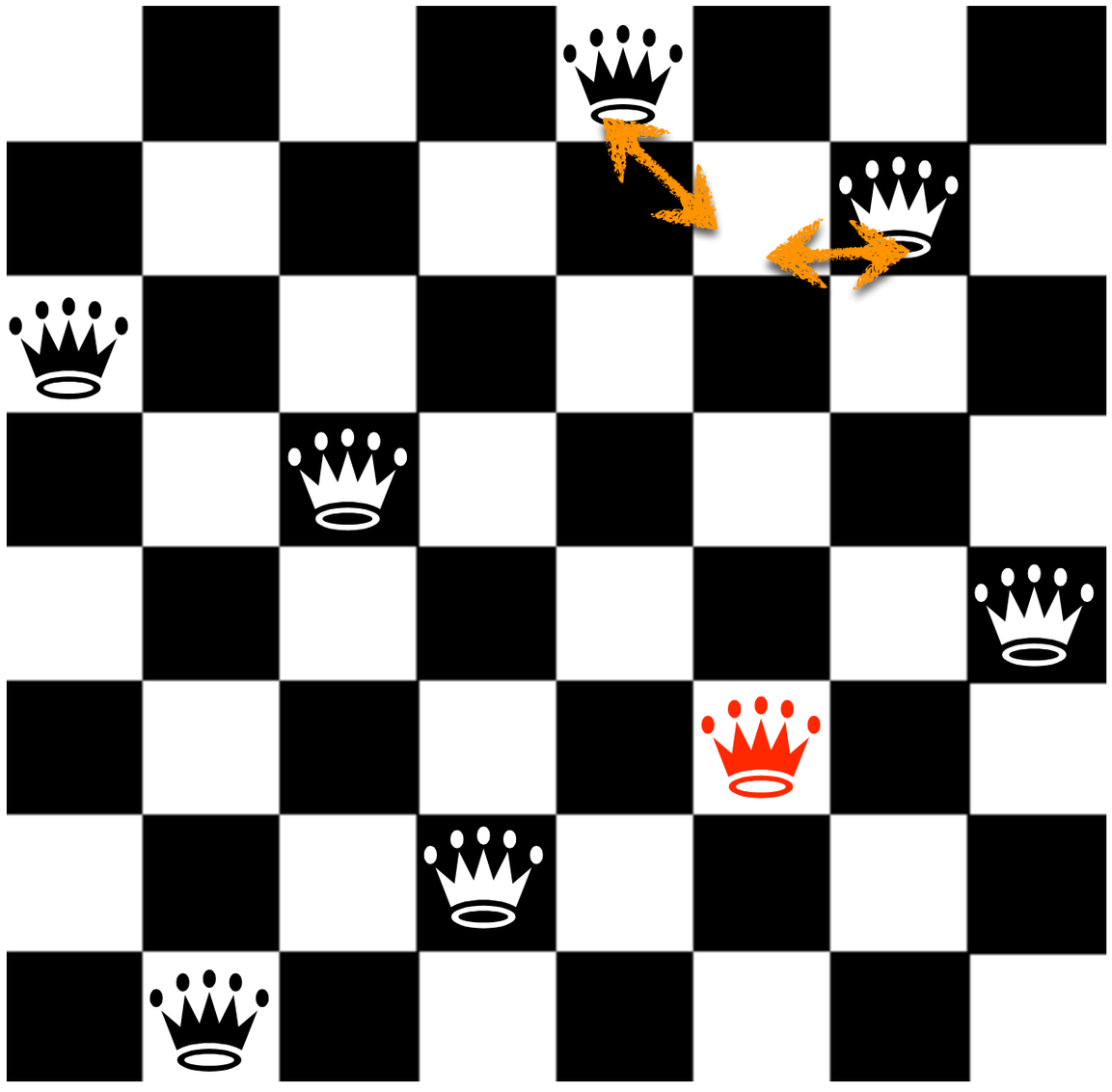
Place 8 queens on a chessboard so that no pair attacks each other.

Eight Queens Problem



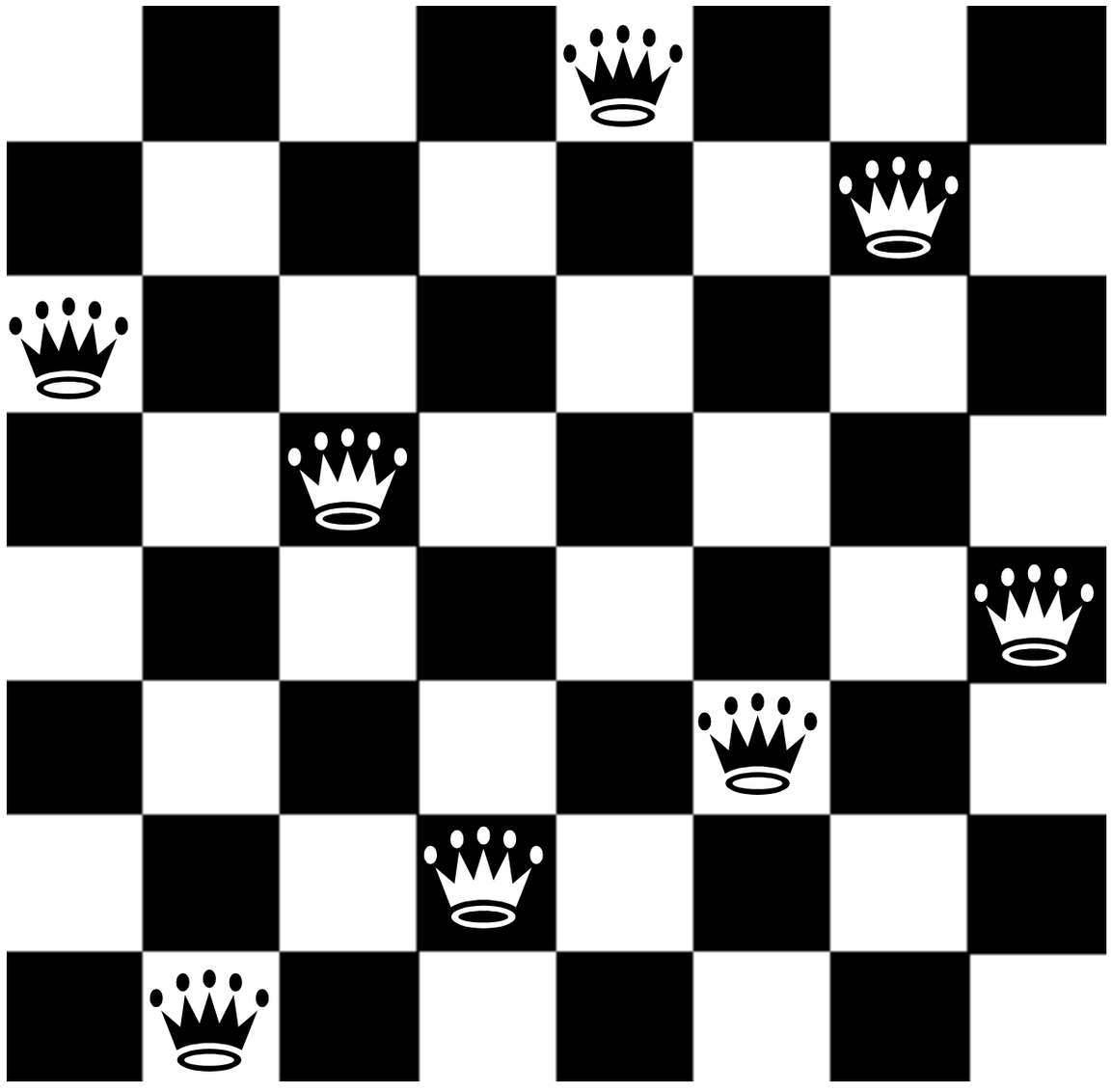
Perfect solution: score 0

Eight Queens Problem

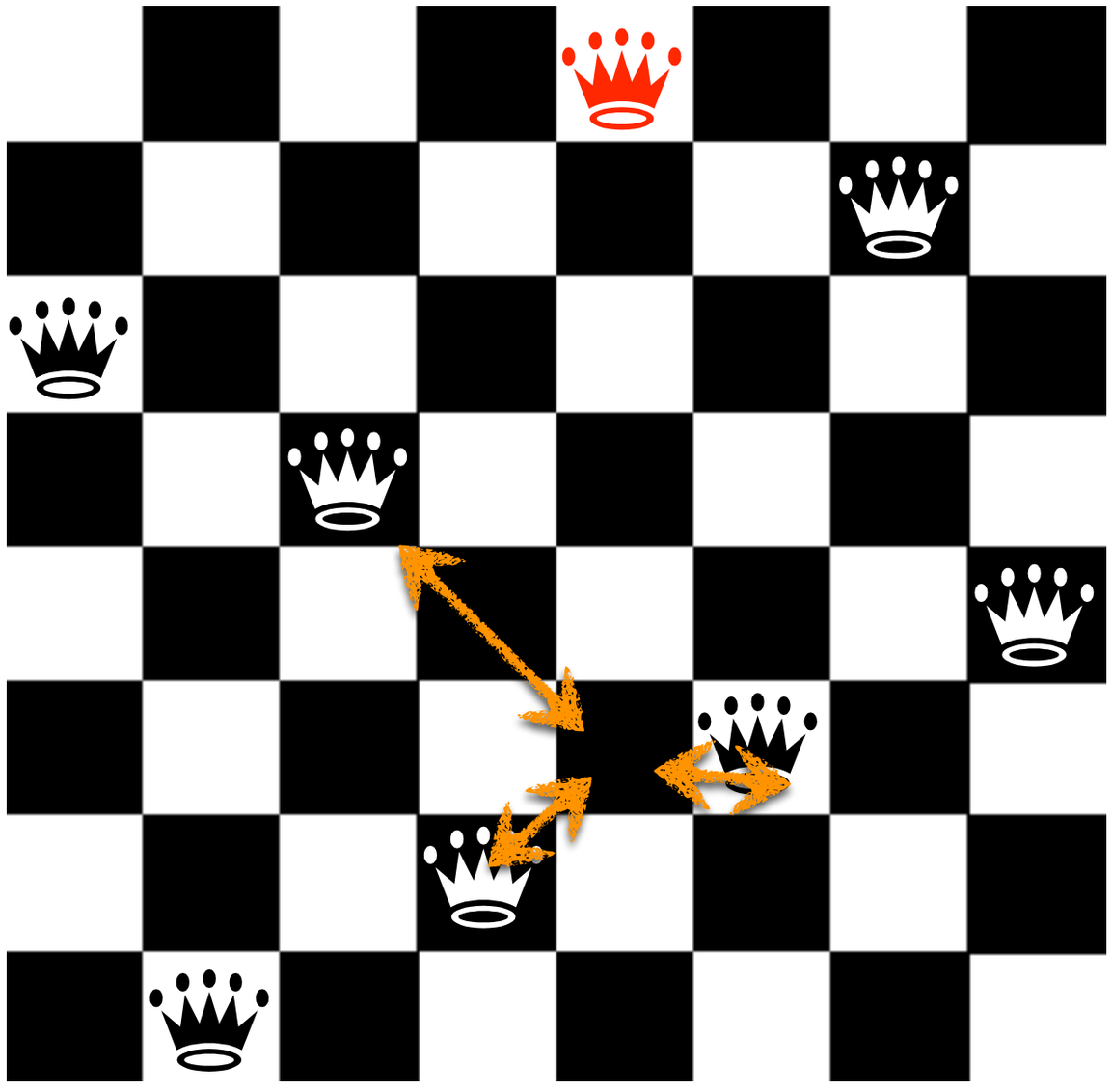


Two attacks: score -2

Eight Queens Problem



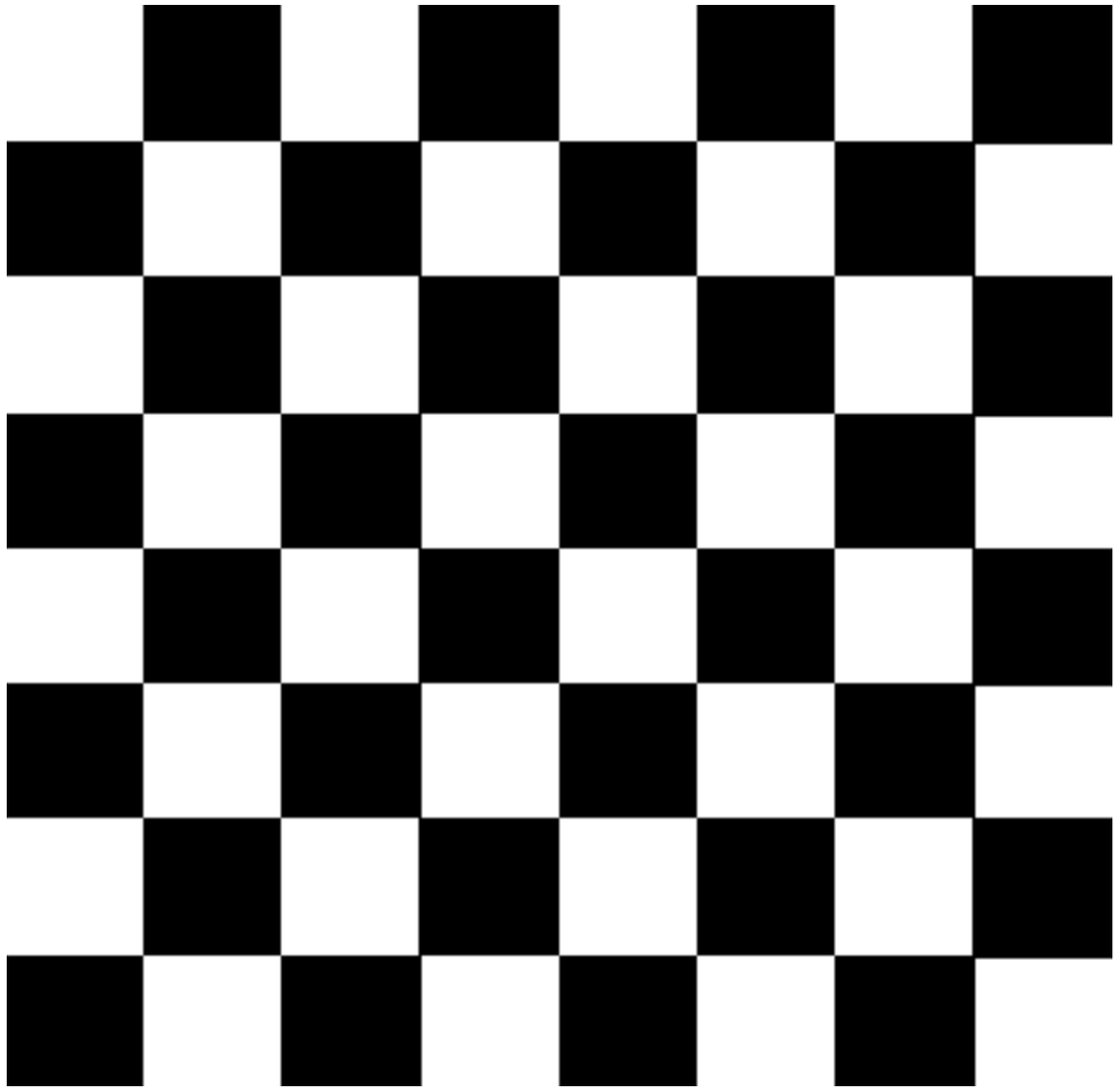
Eight Queens Problem



Three attacks: score -3

Two Approaches

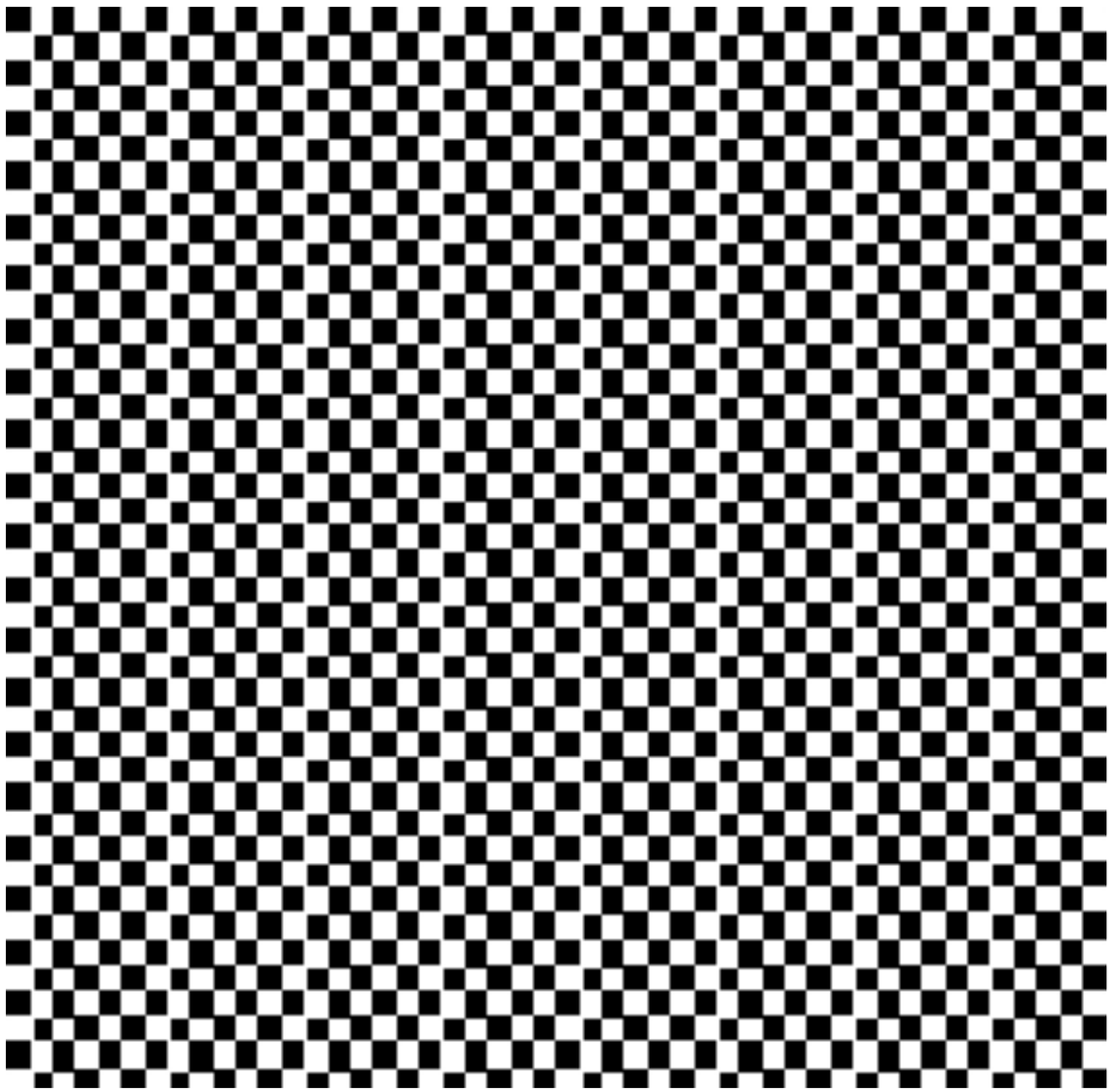
Build an algorithm that produces a solution to the problem by placing one piece at a time



Build an algorithm that compares two solutions to the problem; try different solutions, keep the better one, until you solve the problem

Does it scale?

44 Queens Problem



Place 44 Queens on the board with no attack.

10^{12} Queens Problem

Place 10^{12} Queens on the board with no attack.

Trial and Error

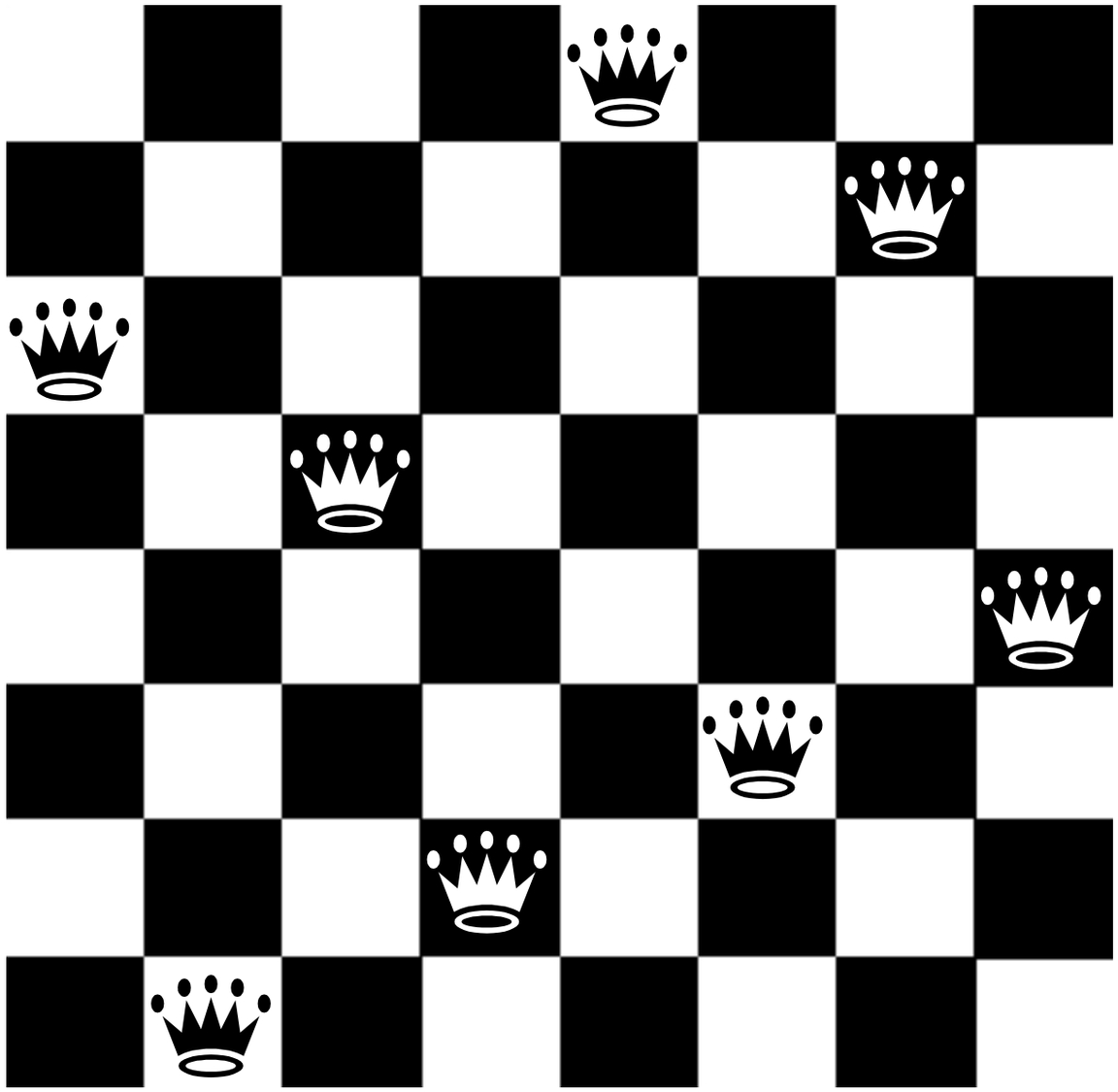
- Abundance of computational resources means many domains are adopting (knowingly or not) a similar approach.
  - Corpus-based NLP
  - Go (the only competitive AI players are based on Monte-Carlo Method)
  - Many application of machine learning

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

8 Queens Problem

- Representation: 8 by 8 matrix
- Operators: generate one valid board position from the current position, following the rule about Queen's movement
- Fitness function: number of attacks (to be minimised)

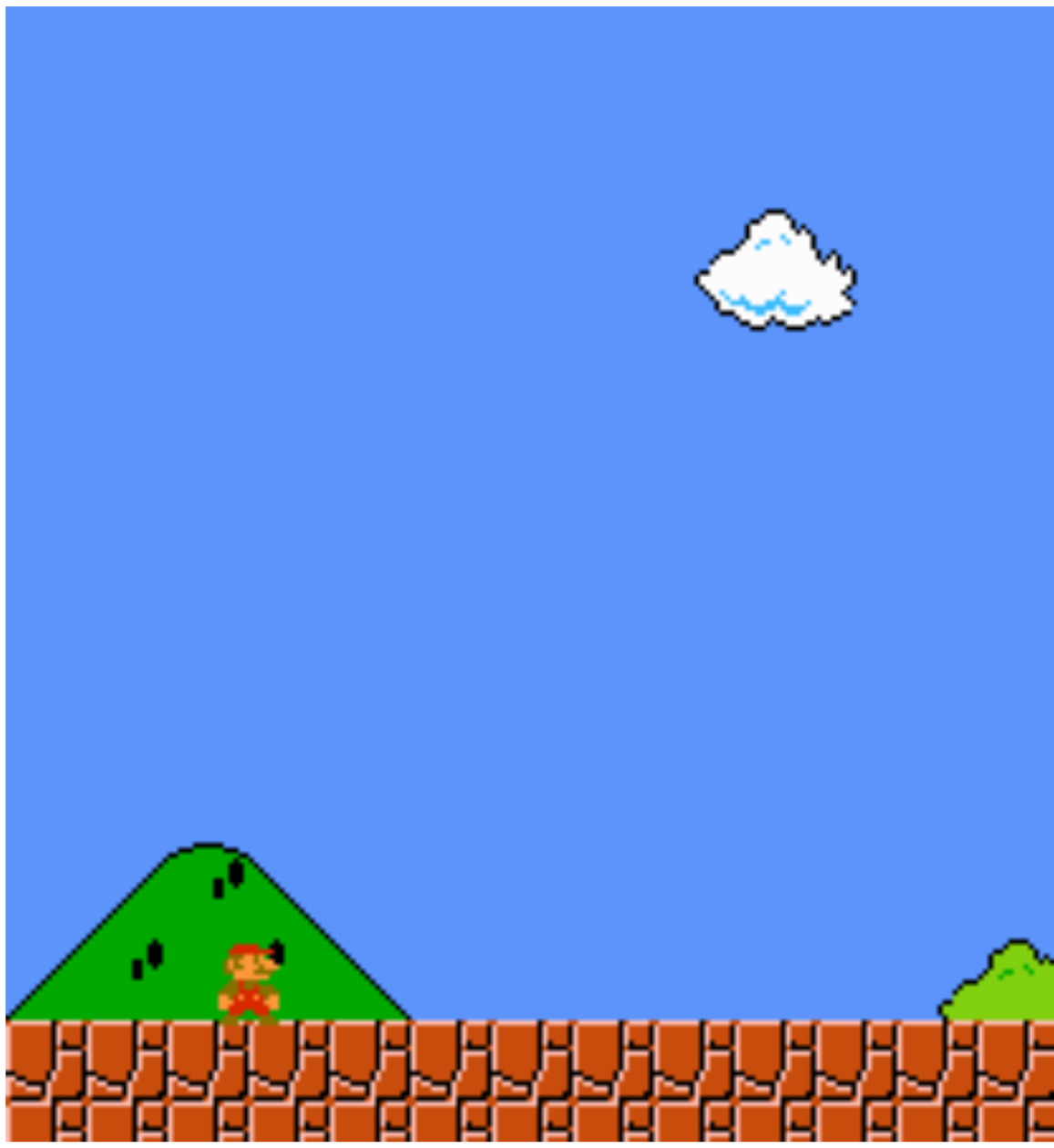


Super Mario Bros.

- 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



Universal Viewpoint

- There are many algorithms in computational intelligence; you do need to learn individual algorithms in detail.
- However, I also want to communicate a frame of thinking, not only individual algorithms.
- The tuple of (representation, operators, fitness function) can be a universal platform to understand different classes of algorithms.
- We will revisit individual algorithms, using this tuples.

Design dictates solution

- Incorrect representation: what happens if we use `(button, pressed_time)` instead?

- Using wrong operators: what happens if we decrease/ increase **`start_time`** and **`end_time`** by 5 seconds?

- Missing constraints: what happens if we swap the order of two tuples?

- Measuring the wrong fitness: what happens if we use the time elapsed until death? Or the final score?

Exploitation vs. Exploration

- Exploitation: if a candidate solution looks promising, optimisation should focus on that particular direction. However,

- Exploration: unexplored solution space may contain something *much better*.

- How to balance these two is critical to all learning/ optimisation algorithms.

Machines are Dumb and Lazy

- Like human, they will do the minimum work that passes your criteria, i.e. design of the optimisation problem.

- Not because of their work ethic, but because of the fact that, usually, minimum work is the easiest to find solution.

Case Study: GenProg

- GenProg uses stochastic optimisation to modify existing faulty software code, until it passes all tests.
- We can only tell it to try until it passes all tests, not until the program is correct.

Anecdotes borrowed from Wes Weimer's SSBSE 2013 Keynote: https://www.cs.virginia.edu/~weimer/p/weimer-ssbse2013.pdf

Things GenProg Did…

- **`nullhttpd`**: test case for POST function failed; GenProg removed the entire functionality.

- **`sort`**: test required output to be sorted; GenProg's fix was to always output an empty set.

- Tests compared **`output.txt`** to **`correct_output.txt`**; GenProg deleted **`correct_output.txt`** and printed nothing.

Current State of the Art

- Low hanging fruits almost gone: “I applied algorithm X to problem Y” no longer counts.

- Metric-based optimisation (where fitness equals an existing SE metric) is starting to be criticised. Compare the following two papers:

  - M. Harman and J. Clark. **Metrics are fitness functions too**. In 10th International Software Metrics Symposium (METRICS 2004), pages 58–69, Los Alamitos, California, USA, Sept. **2004**. IEEE Computer Society Press.

  - C. Simons, J. Singer, and D. R. White. **Search-based refactoring: Metrics are not enough**. In M. Barros and Y. Labiche, editors, Search-Based Software Engineering, volume 9275 of Lecture Notes in Computer Science, pages 47–61. Springer International Publishing, **2015**.

How to do better?

- Try to learn from human engineers more.
- Eventually, solutions from SBSE should be adopted because humans accept it, **not** because the fitness value is above an arbitrary cut-off point X.
- Turing-test as fitness function!

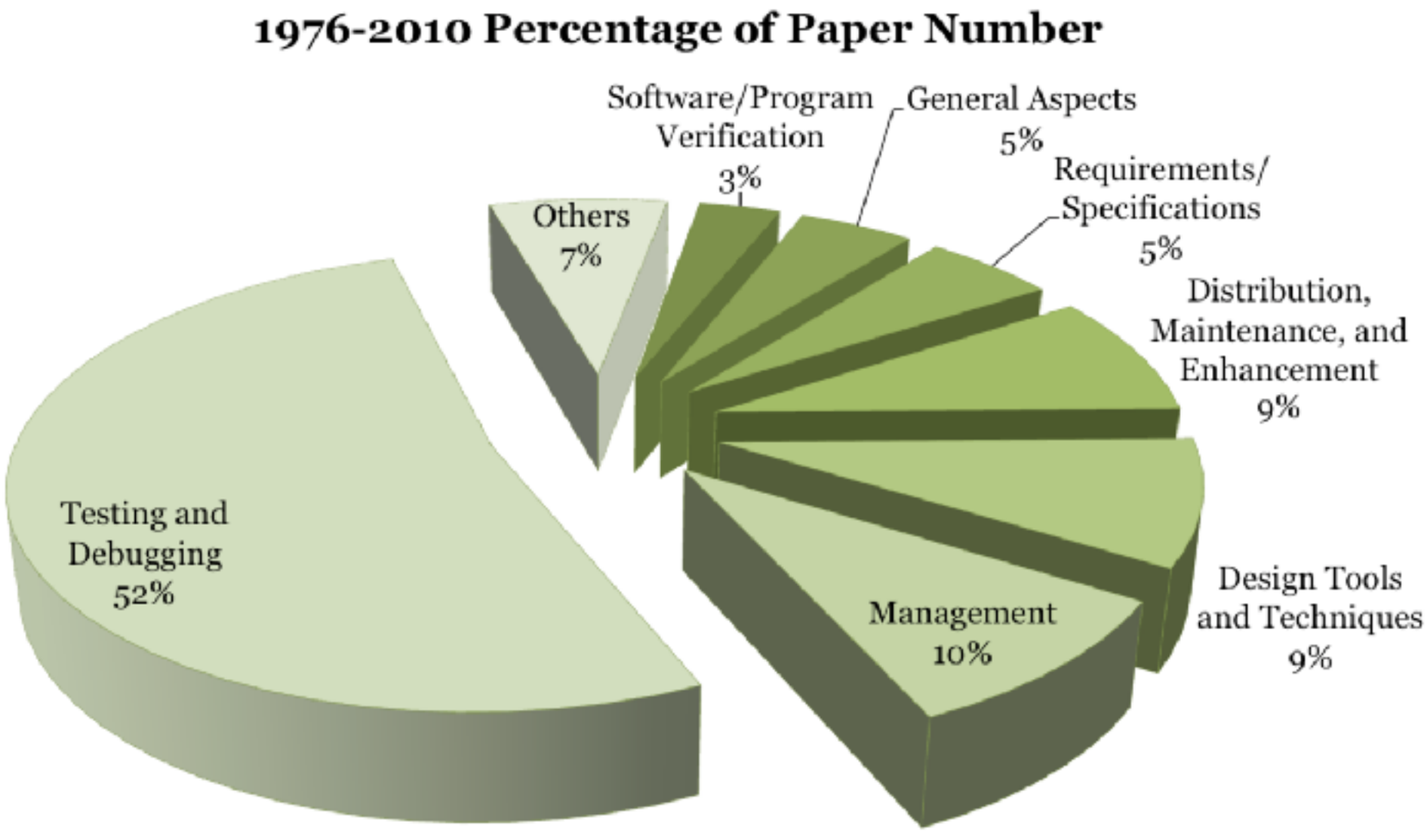
Expected Learning Outcome

- **Understand** basic metaheuristic algorithms; learn how to **implement** and **adapt** one to a given problem.

- **Embrace** metaheuristic optimisation as a valid tool for software engineers.

- Gain knowledge of the **literature**; learn **case studies** for various software development lifecycle stages.

Problem Domains



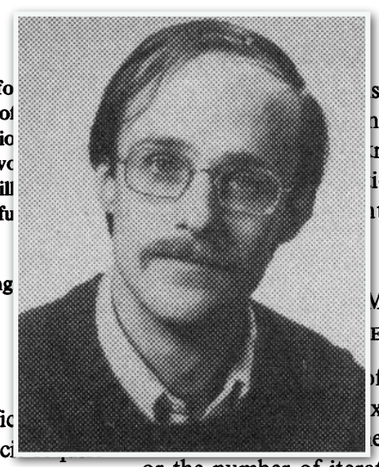
Structural Testing

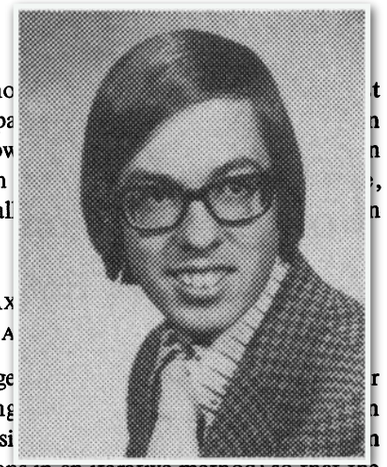
- Intuitively: define th
  achieve structural c
  using optimisation.
  - Symbolic executi
  - Dynamic analysis
- Either way, huge ac
  - Clearly defined fi
    least on achievin

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO. 3, SEPTEMBER 1976

Automatic Generation of Floating-Point Test Data

WEBB MILLER AND DAVID L. SPOONER





Abstract—For numerical programs, or more generally fo
with floating-point data, it may be that large savings o
storage are made possible by using numerical maximizatio
instead of symbolic execution to generate test data. Two
a matrix factorization subroutine and a sorting method, ill
types of data generation problems that can be successfu
with such maximization techniques.

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

INTRODUCTION

RESEARCH in program evaluation and verific
only rarely (e.g., [1]) begun with the explici
ment 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]

s no
n pa
now
ion
itial

Max
ERA

f ge
xing
ensi

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., $c = \text{ABS}(X - Y)$ 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
```

Oracle Problem

- Coverage is not enough: “was the last execution **correct**?”
- Test oracle tells you whether the observed execution was correct or not
- Formal specification can serve as one; manual inspection by human can serve as one. But how do we automatically generate oracles?
- We want to test the code; we automatically generate test from the code; we want to check whether the test passed; we automatically generate test oracle from the co… wait a minute!
- This is a very hard problem; one which the state of the art does not know how to solve.

Testing non-functional properties

- Worst-Case Execution Time Analysis: strictly necessary for certain embedded systems (e.g. airbag controller), very hard to do statically; genetic algorithm has been very successful.

- J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Systems, 15(3): 275 – 298, 1998.

Requirements Engineering

- Next Release Problem: given cost and benefit (expected revenue) for each features, 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.
- Sensitivity Analysis: requirements data are usually estimates; which estimation will have the largest impact on the project, if it is off by X%?

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

Design/architecture/ refactoring

- Cluster software models to achieve certain structural properties (cohesion/coupling).
- Ironically, SBSE has also been used to analyse refactoring metrics: metric A and B both claim that they measure the same concept - optimising for A resulted in worse value of B, and vice versa :)

Genetic Improvement

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

- Genetic Programming has been successfully applied to make genome-sequencing software 70 times faster. 70!

    - W. Langdon and M. Harman. Optimizing existing software with genetic programming. Transactions on Evolutionary Computation, 19(1):118–135, 2015.

- Evolve a specialised version of MiniSAT solver for problem classes.

    - J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In proceedings of the 17th European Conference on Genetic Programming, volume 8599 of LNCS, pages 137–149. Springer, 2014.

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.

Summary

- Key ingredients to SBSE: representation, operators, fitness function.

- Design dictates solutions.

- Applications across all software development lifecycle activities, and beyond.

SBSE Repository

- http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/