Solving Aristotle's Puzzle

Evolutionary Algorithms for Your Everyday Task

Marco Neumann

BlueYonder
a jda. company

BlueYonder
Best decisions, delivered daily

”Aristotle’s Puzzle” is officially referred to as “Magic Hexagon”!

BlueYonder
a jda. company

jda.
Plan to deliver™

Selected Customers

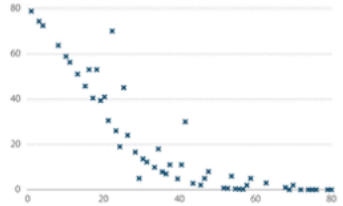
BlueYonder
a jda. company

Dynamic Price Optimization

BlueYonder
a jda. company

What will happen?

Measure of price elasticities



- Continuous calculation of price elasticities of demand (per store, product, time)
- Identification of relevant prediction features
- Data-driven learning of price-demand relations

What should I do?

Price Optimization algorithm

- Apply current price rules
- Consider competitor prices, location-individual stock levels and storage costs
- Apply margin and revenue expectations, optimizing them along price-demand relations

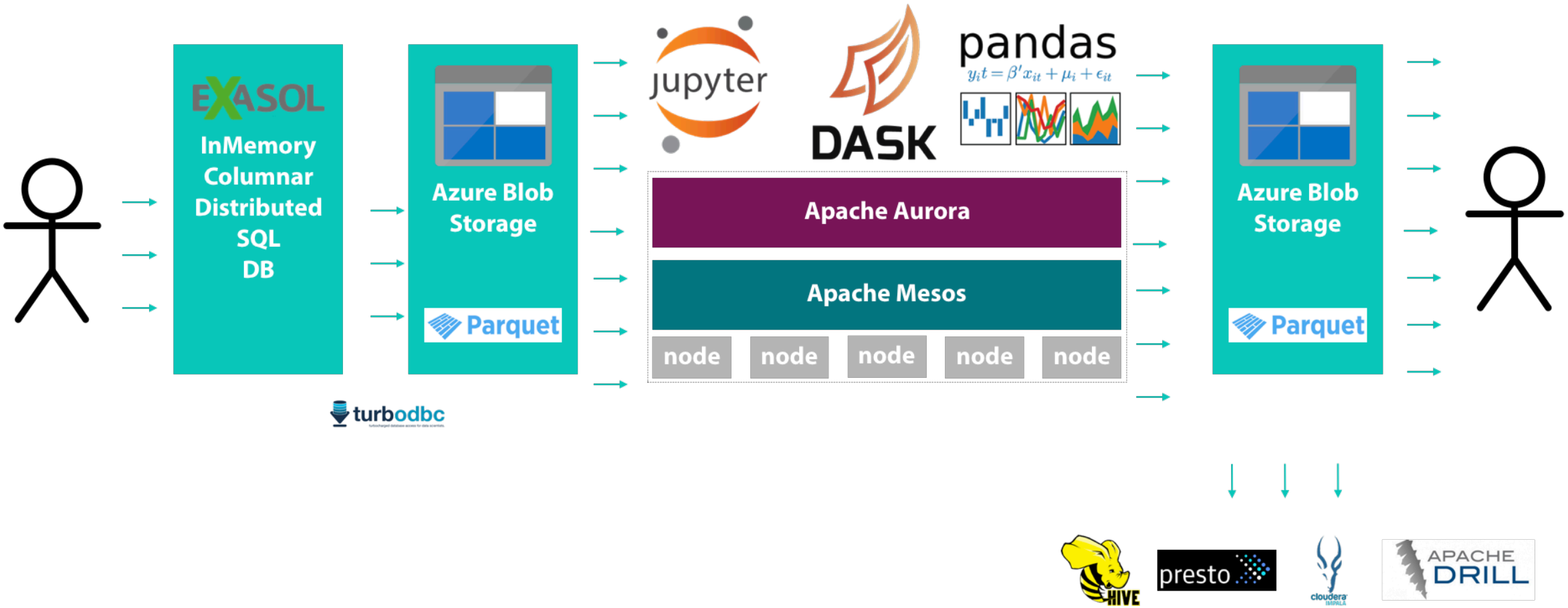
Automated decisions

Optimized price suggestions

- Optimized prices
  - Per channel
  - Per product
- Price calculation according to company strategy

Data Flow for Machine Learning

BlueYonder
a jda. company



Want to join?

BlueYonder
a jda. company

We are looking for open-minded candidates for our development teams:

Data Scientist

Software Engineer

Data Engineer

Site Reliability Engineer

Application via E-Mail:

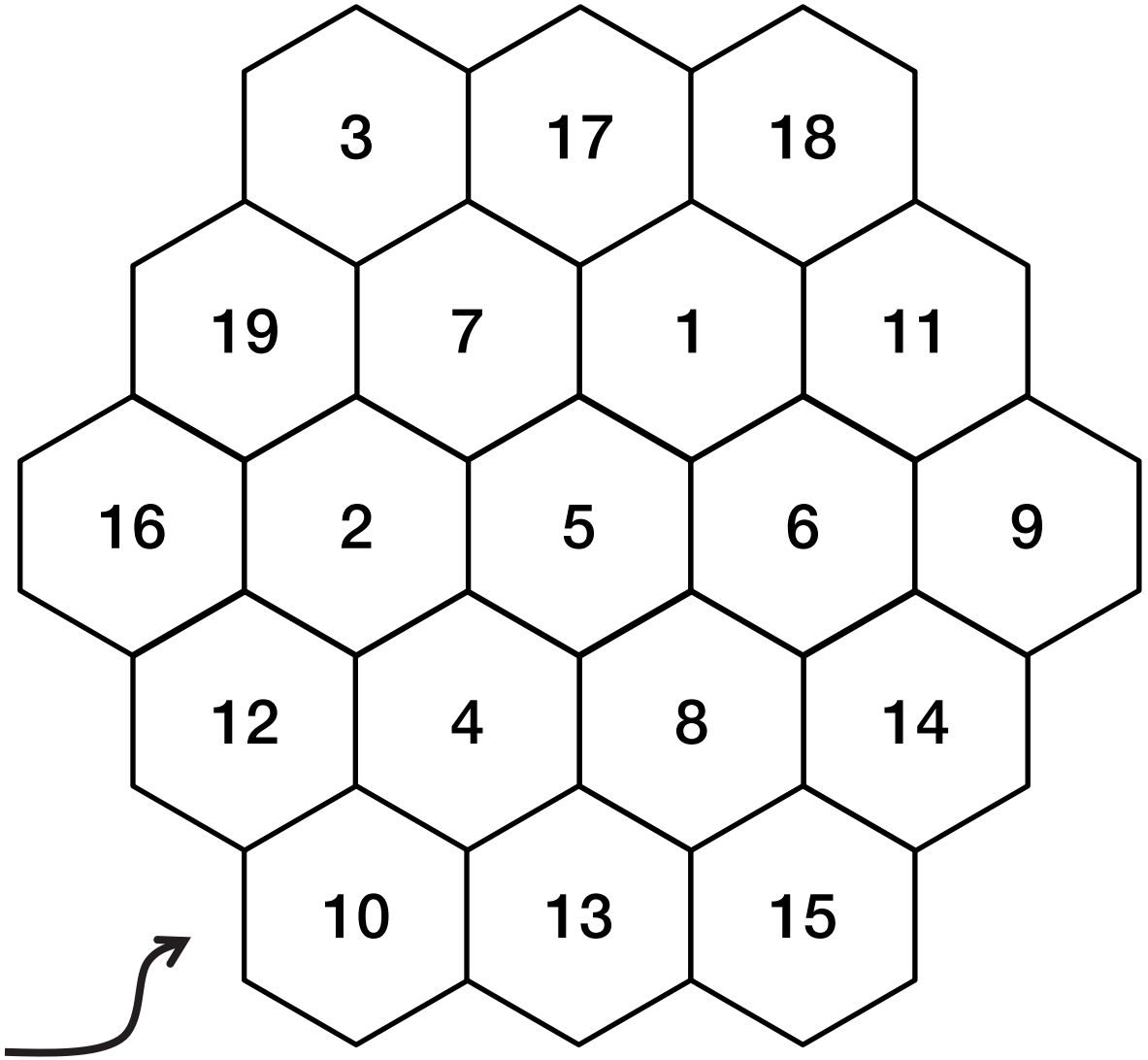
Contact Person: Inga Binder
Subject Line should contain “Meetup”
PD-Candidates-GER@jda.com

Job Listing:

https://stackoverflow.com/jobs/companies/blue-yonder-gmbh

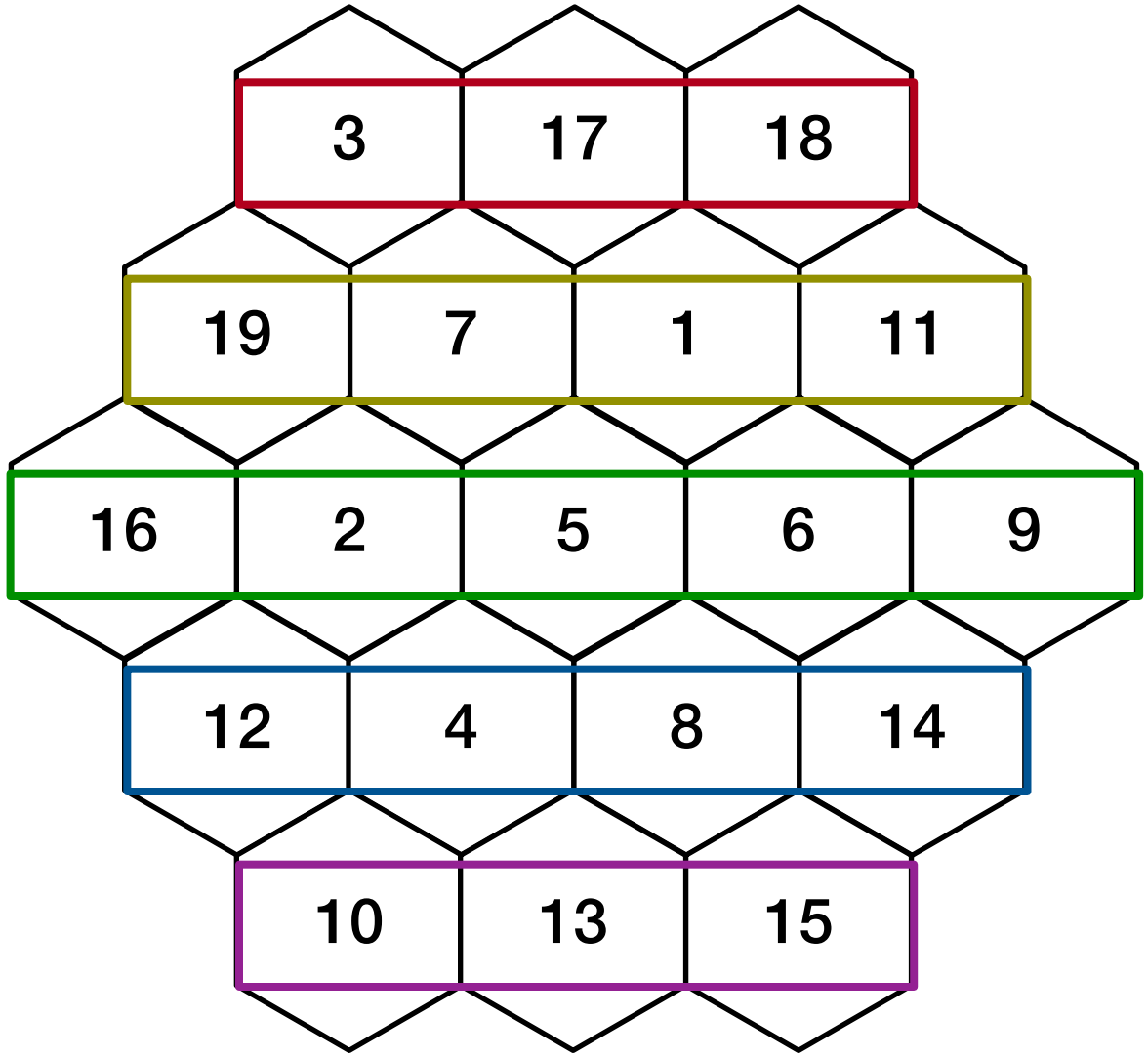
The Problem

19 Tokens

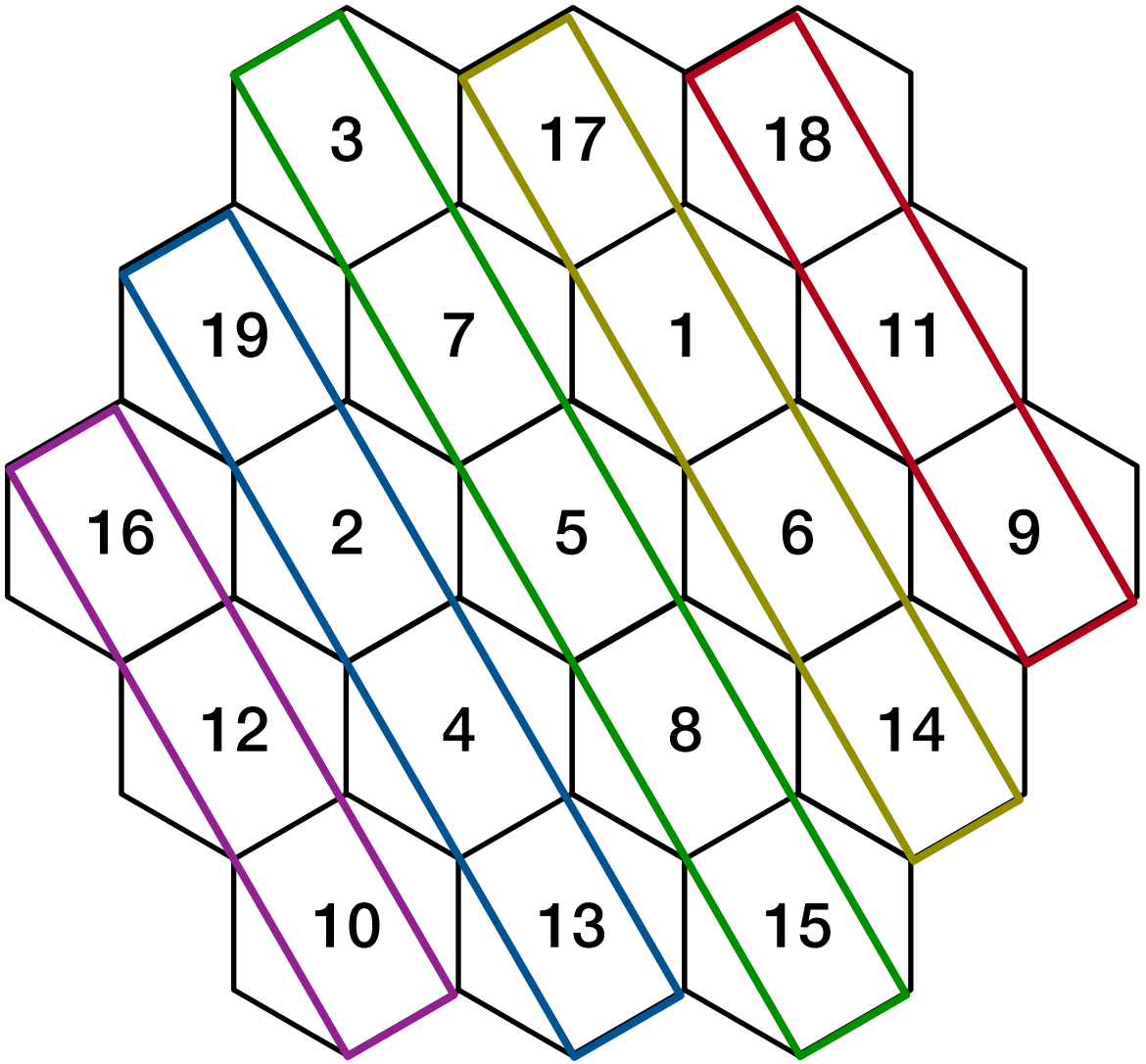


That's actually the solution

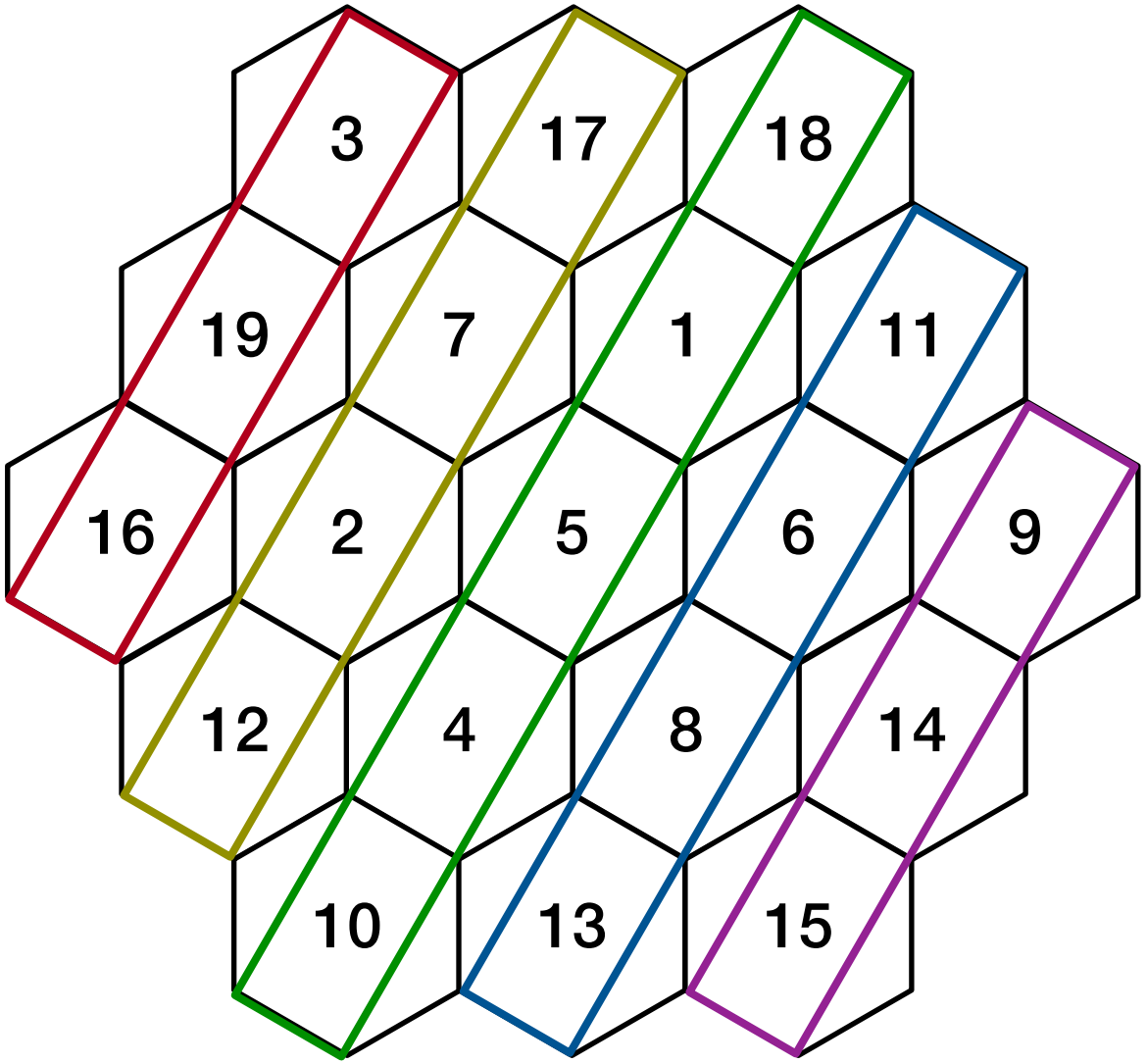
$$\sum_{i \in R} T_i = 38$$



$$\sum_{i \in R} T_i = 38$$



$$\sum_{i \in R} T_i = 38$$



The Solution

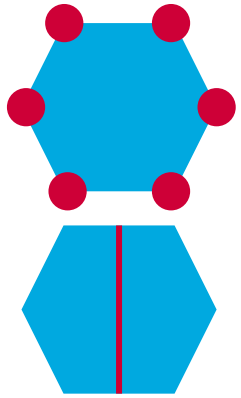
Brute force

BlueYonder
a jda. company

- $19! \approx 1.2 \times 10^{17}$ combinations

Age of the solar system: $1.44 \times 10^{17} s$

- Assume:
  - a processor core that can generate + check 1×10^{9} combinations per second
  - 100 cores
- Checking all takes $\approx 1.2 \times 10^{6} s \approx 338 h \approx 14 d$
- Assume 1 real solution with:
  - 6 rotations
  - 2 mirror images



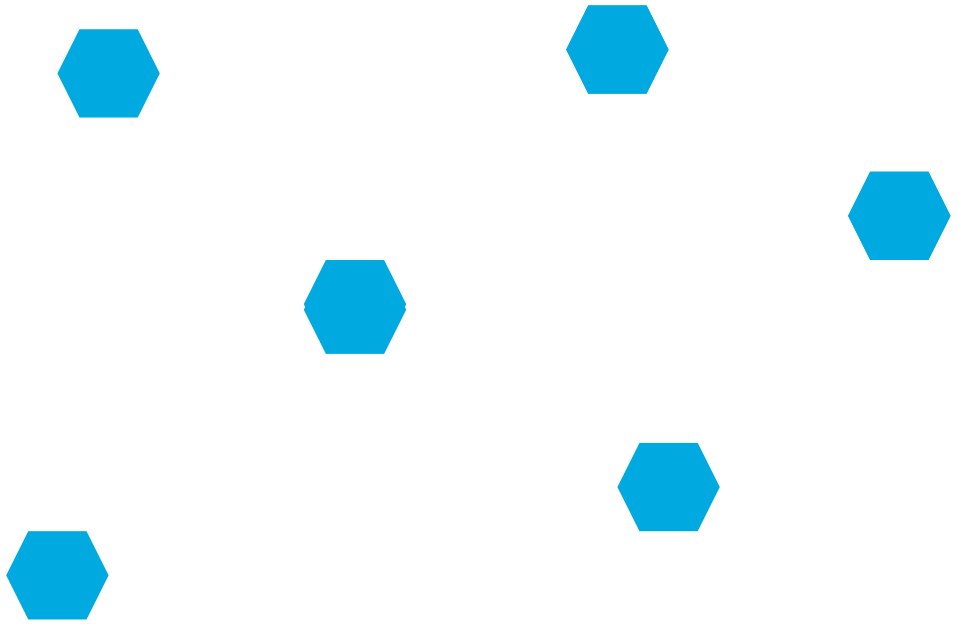
- So on average we expect $\approx \frac{14d}{6 \times 2 \times 2} \approx 14h$

Evolutionary Algorithm

BlueYonder
a jda. company

Generate

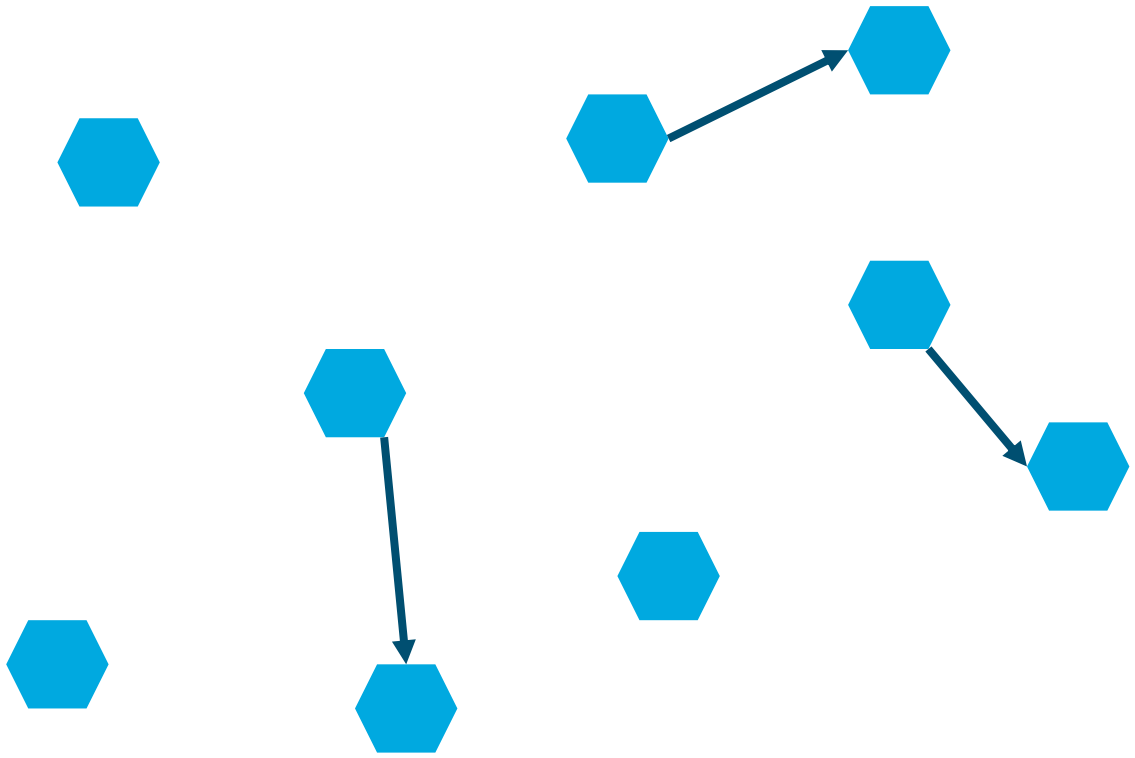
Evolutionary Algorithm



Generate

Mutate

Evolutionary Algorithm

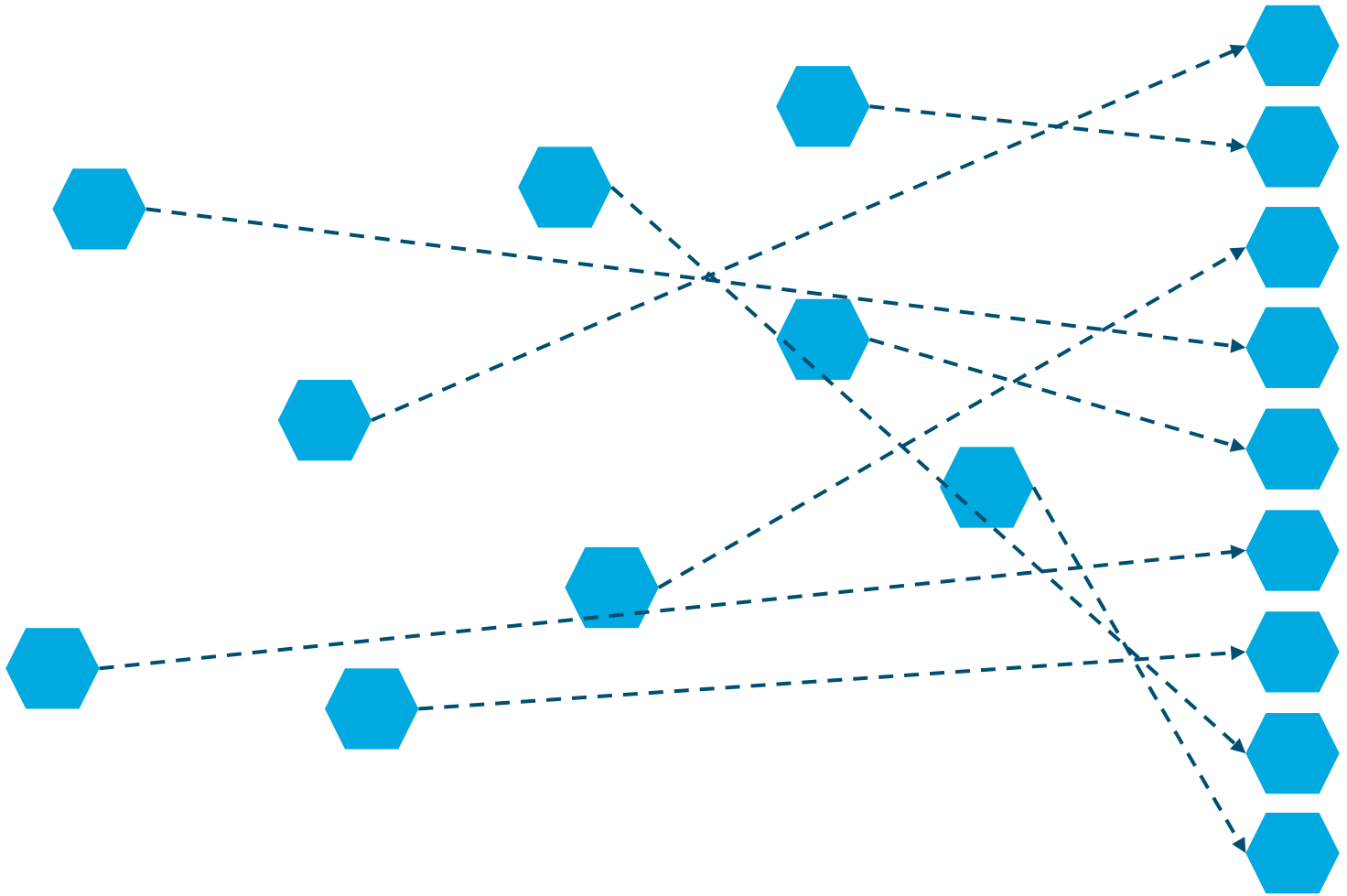


Generate

Mutate

Evolutionary Algorithm

BlueYonder
a jda. company



Generate

Mutate

Order

Evolutionary Algorithm

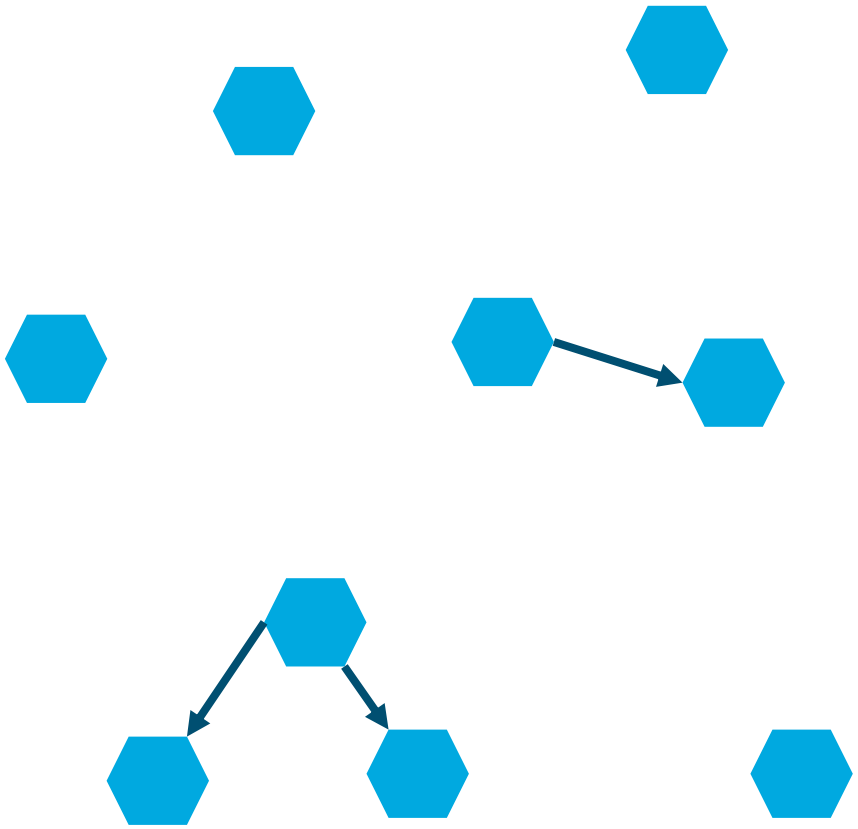
Generate

Mutate

Order

Select

Evolutionary Algorithm



Generate

Mutate

Order

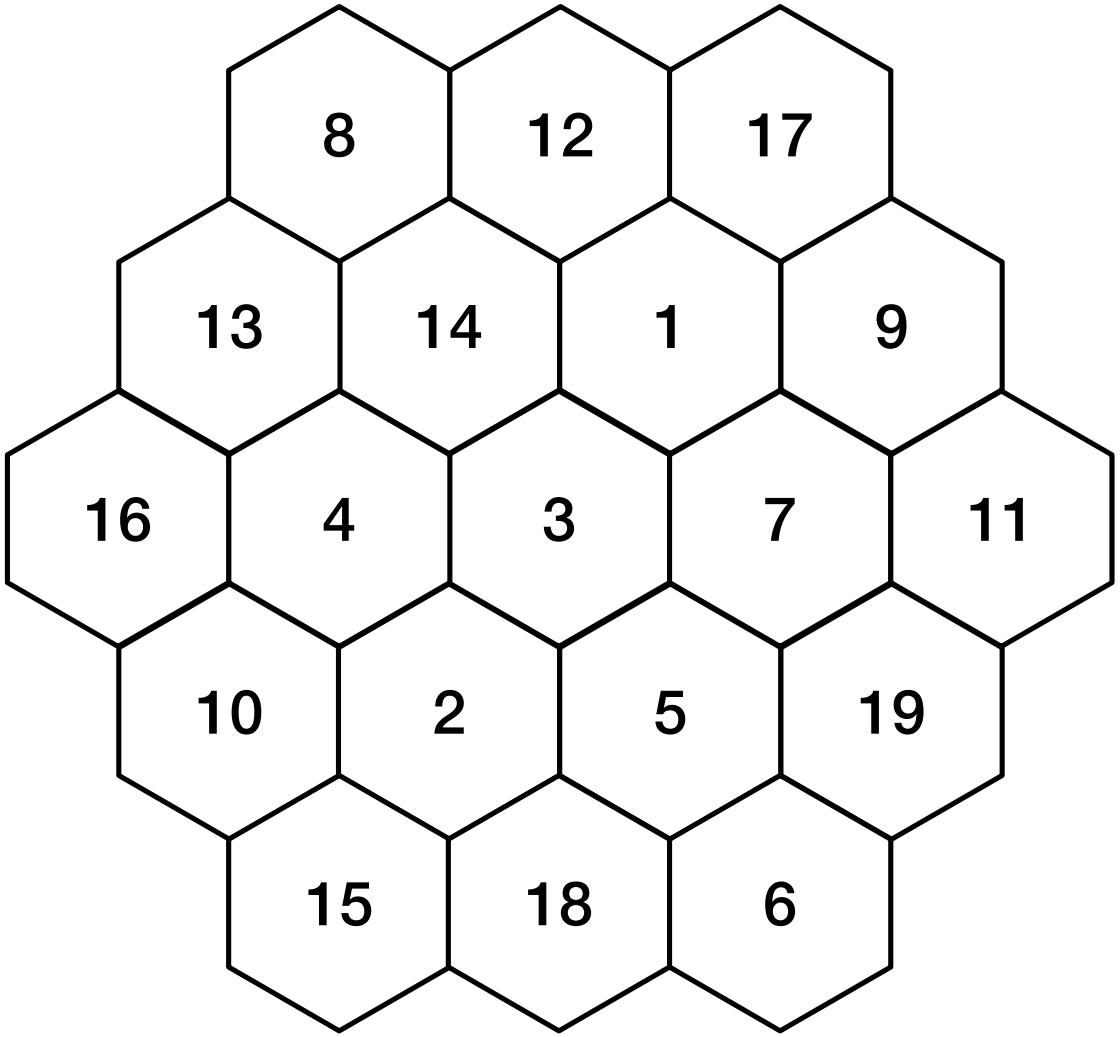
Select

Mutate

Candidate Generation

BlueYonder
a jda. company

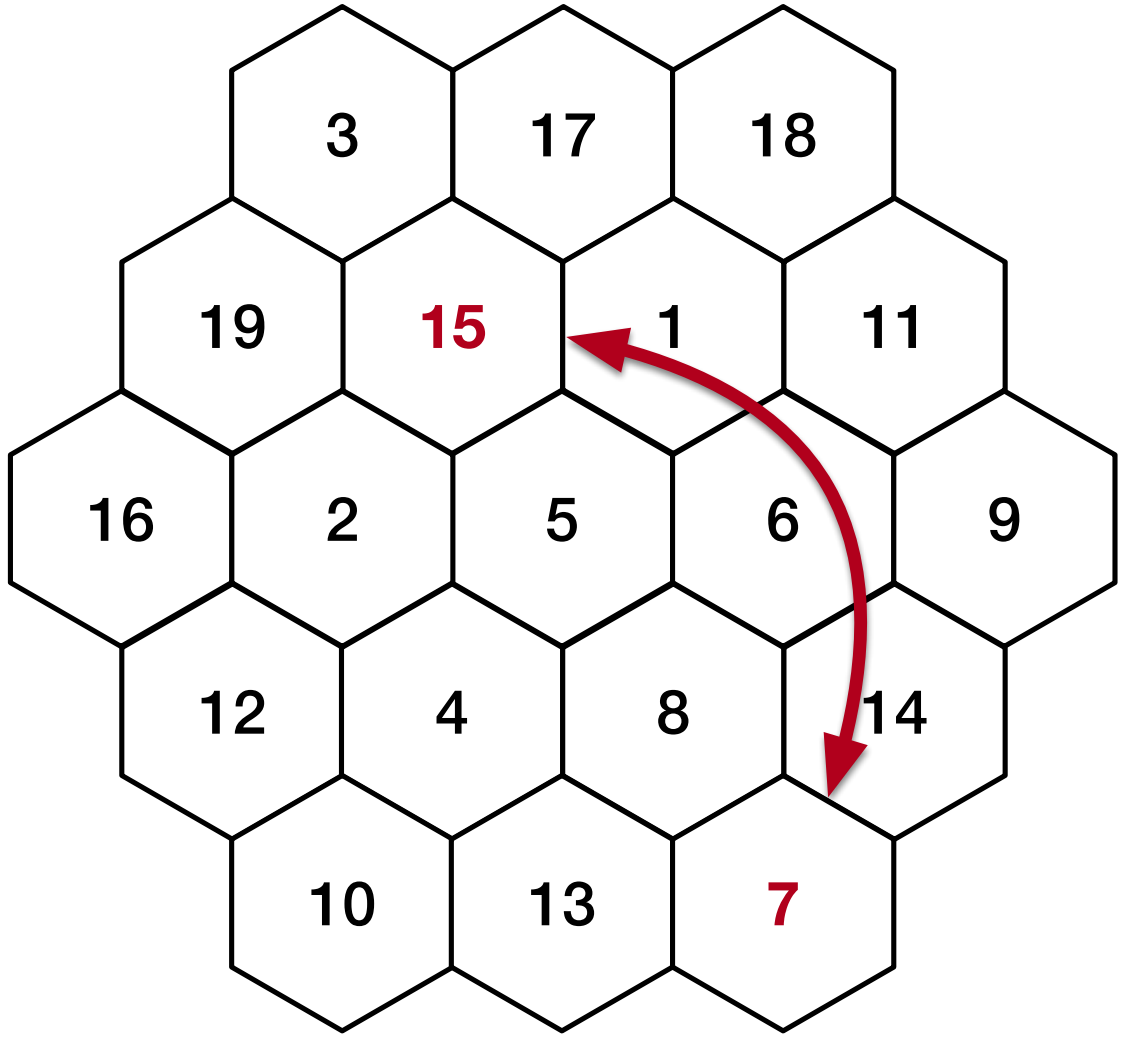
Choose random permutation



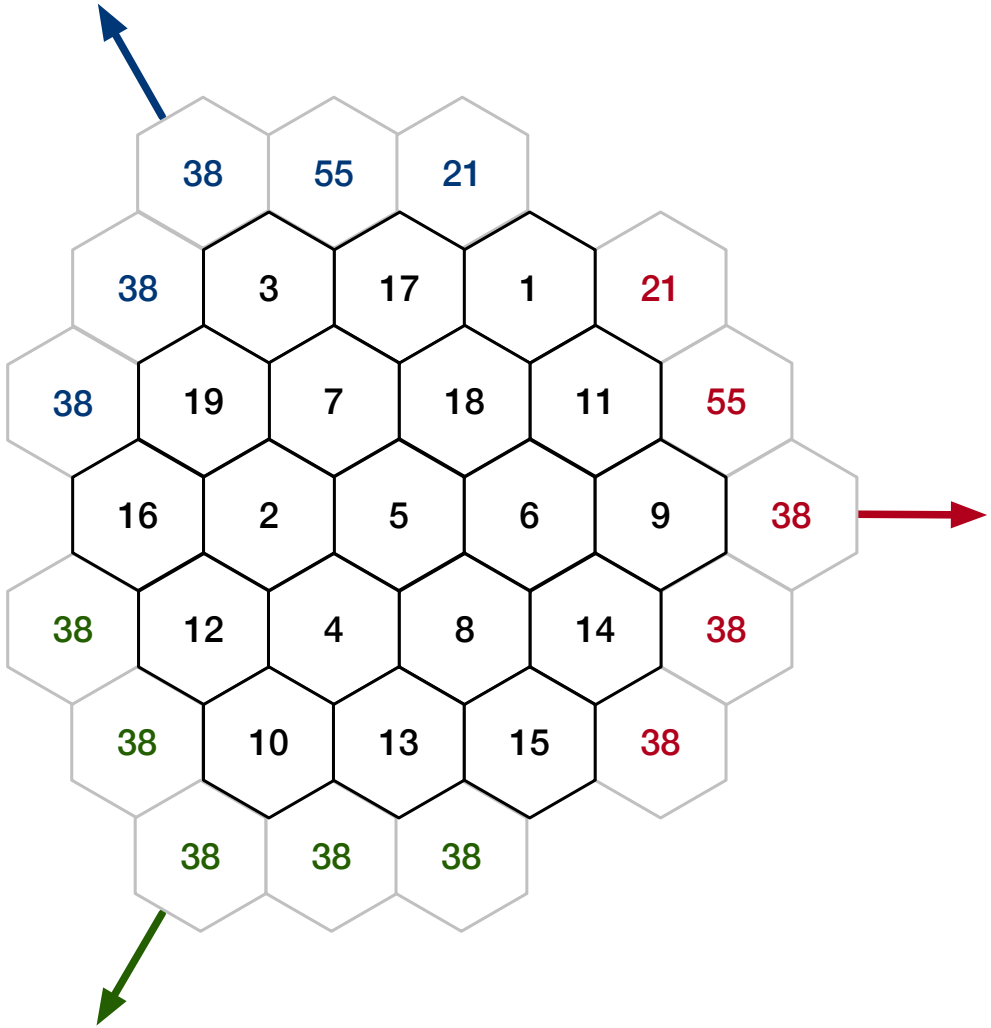
Mutation

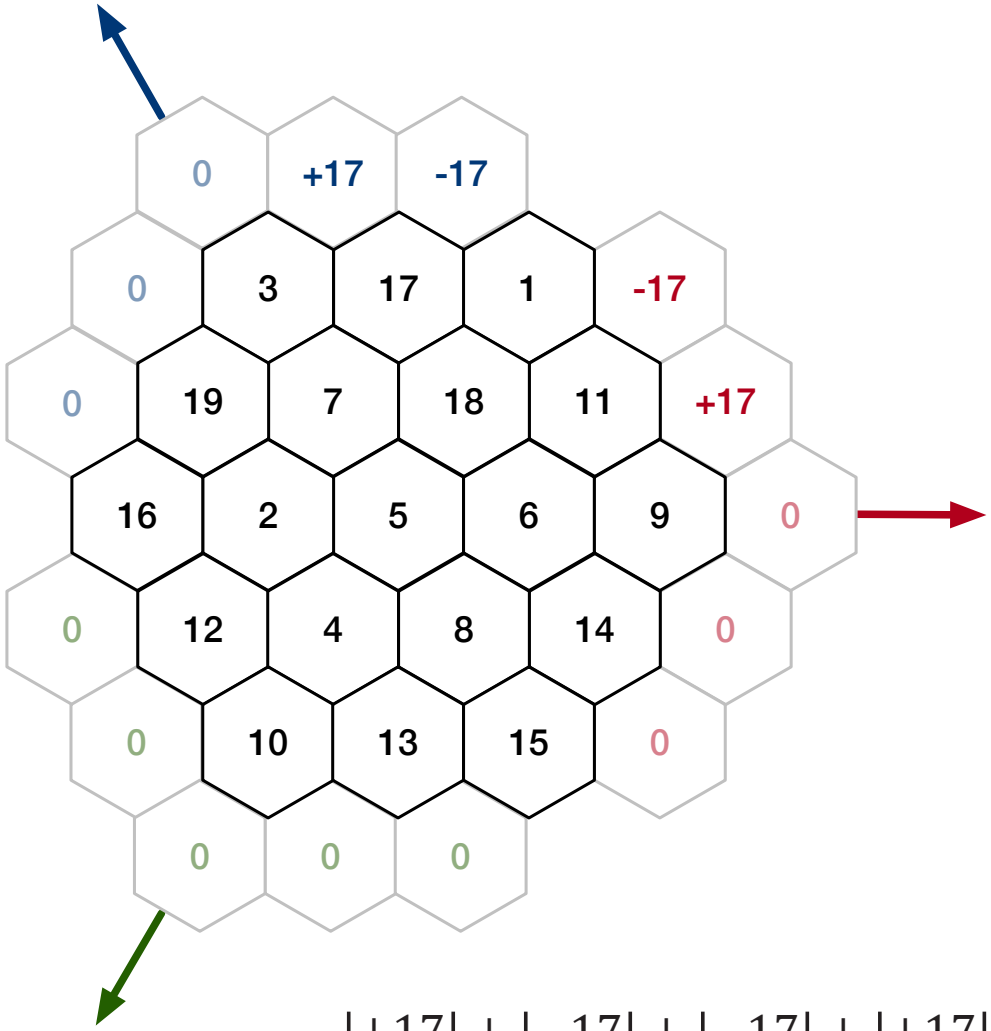
BlueYonder
a jda. company

Swap random tokens



Fitness Function





$$|+17| + |-17| + |-17| + |+17| = 68$$

Improvement #1

BlueYonder
a jda. company

Issue:

Fit candidates might get duplicated, take over population.

Solution:

Keep unique candidates (`HashSet`).

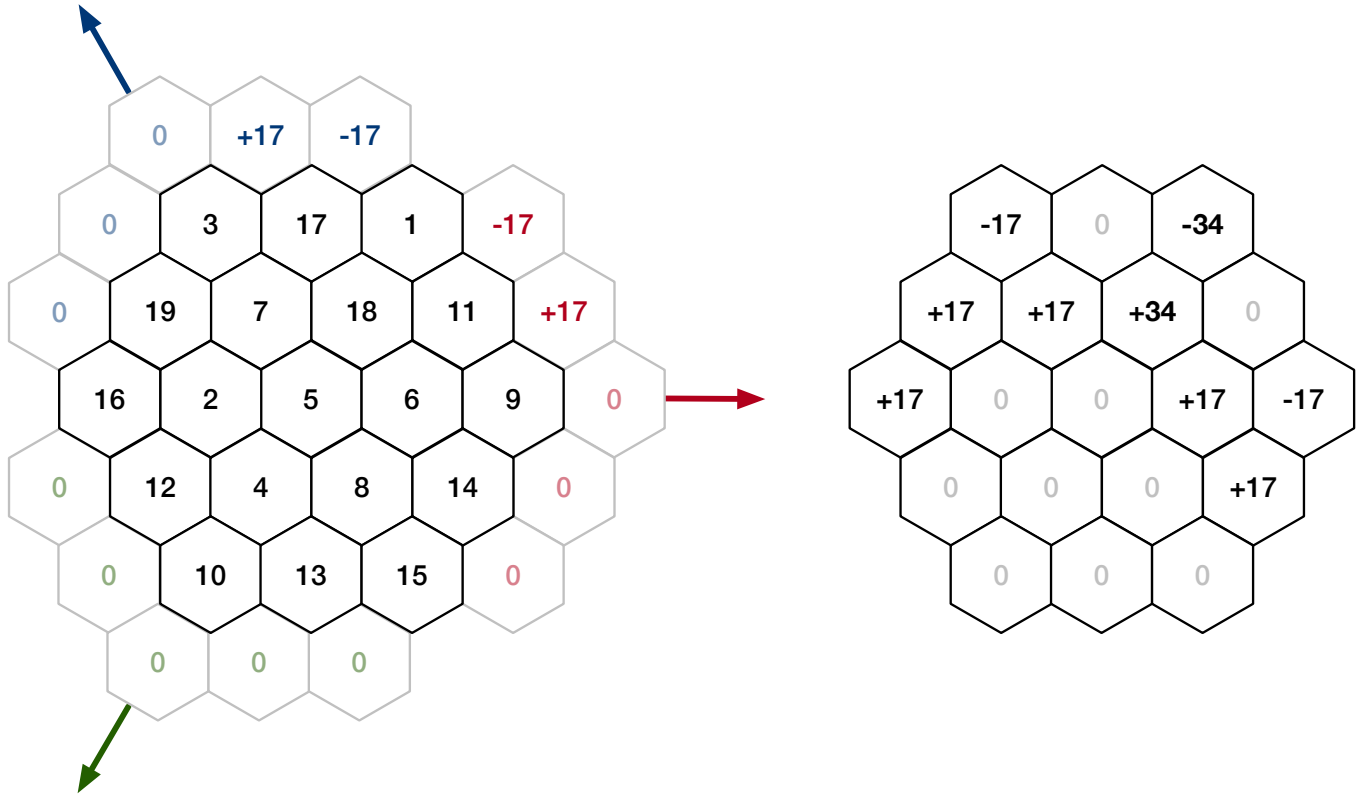
Improvement #2

Issue:

- Fitness function only evaluates equations, not tokens.
- Humans would

Solution:

Add token-based fitness function.



$$|+17| + |-34| + |+17| + |+17| + |+17| + |+34| + |+17| + |+17| + |-17| + |+17| = 204$$

Improvement #3

BlueYonder
a jda. company

Issue:

- Single mutation swap operation converges too slow.
- Multiple swap operations can ”overshoot”.

Solution:

Use random number of swaps.

Improvement #4

BlueYonder
a jda. company

Issue:

Doesn’t converge all the time.

Solution:

Use larger population (now 10k).

Improvement #5

BlueYonder
a jda. company

Issue:

It’s slow.

Solution:

Iterating over `HashSet` is slow, don’t do that.

Improvement #5

BlueYonder
a jda. company

Issue:

It’s still slow.

Solution:

Use `[u8; 19]` instead of `Vec<u8>` for candidates.

Improvement #6

BlueYonder
a jda. company

Issue:

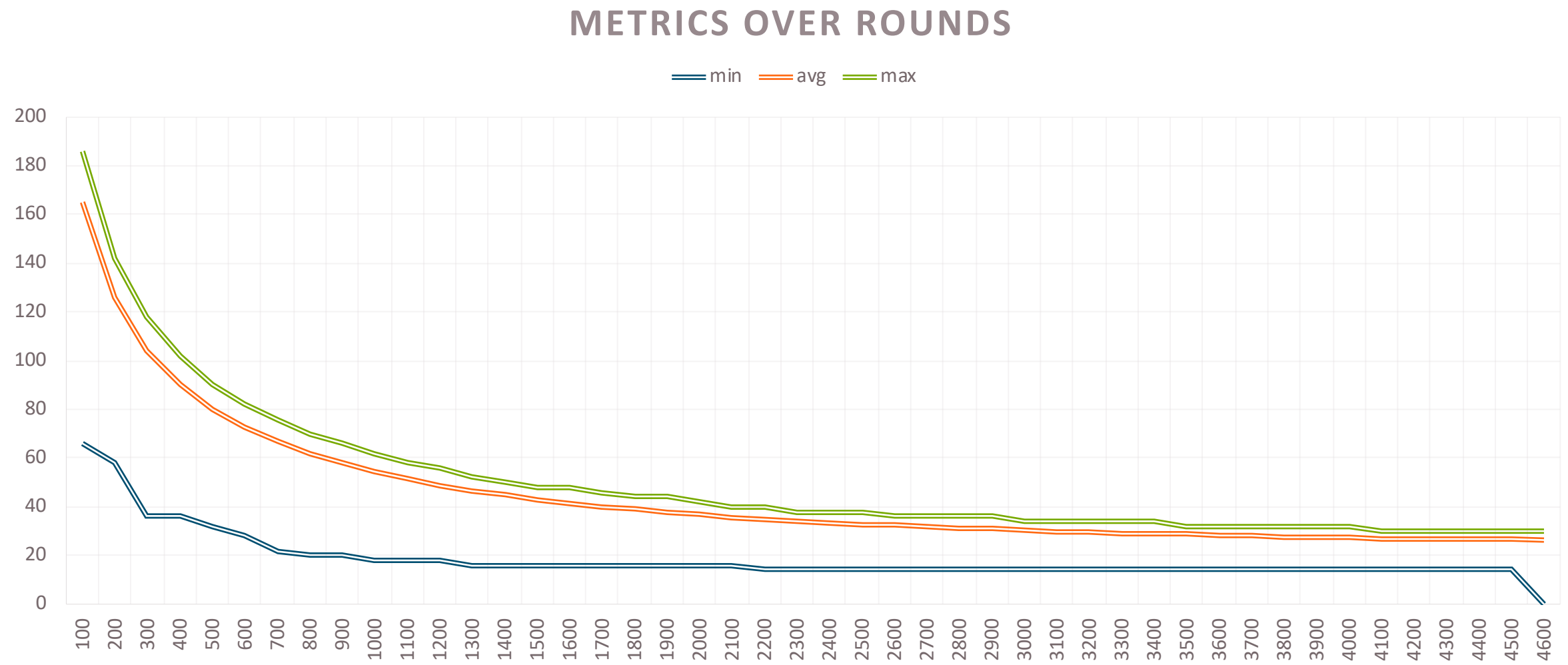
Still, could be faster.

Solution:

Tune compilation in `Cargo.toml`:

```
[profile.release]
debug = true
panic = "abort"
lto = true
codegen-units = 1
```

Convergence

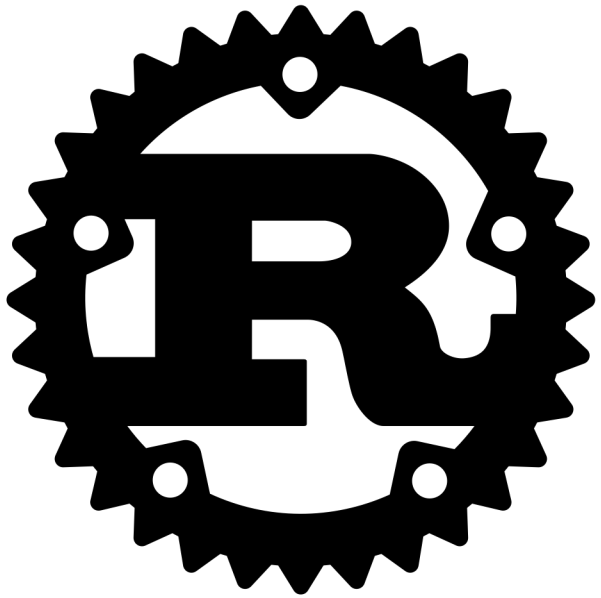


How Rust Helps

BlueYonder
a jda. company

Compact

- 142 lines, well formatted
- No weird performance hacks required



Compiler

It compiles ➔ it runs.

Performance

```
Benchmark #1: cargo run --release
  Time (mean ± σ):     48.387 s ± 23.373 s    [User: 41.296 s, System: 0.858 s]
  Range (min … max):   19.084 s … 104.775 s    10 runs
```

Potential Extensions

BlueYonder
a jda. company

Performance

- Parallel Solution
- Pre-calculate fitness function

Mutation

- Better swap count (e.g. “simulated annealing”)
- Intelligent swap (e.g. use token-based rating)
- Crossover
- Normalize rotation and mirroring

Appendix

Other Applications

BlueYonder
a jda. company

- Other Puzzles like Sudoku
- Conference Scheduling: https://medium.com/@filiph/a-genetic-algorithm-scheduled-our-developer-conference-heres-what-i-learnt-eac0069709f5
- Training Neural Networks: https://openai.com/blog/evolution-strategies/

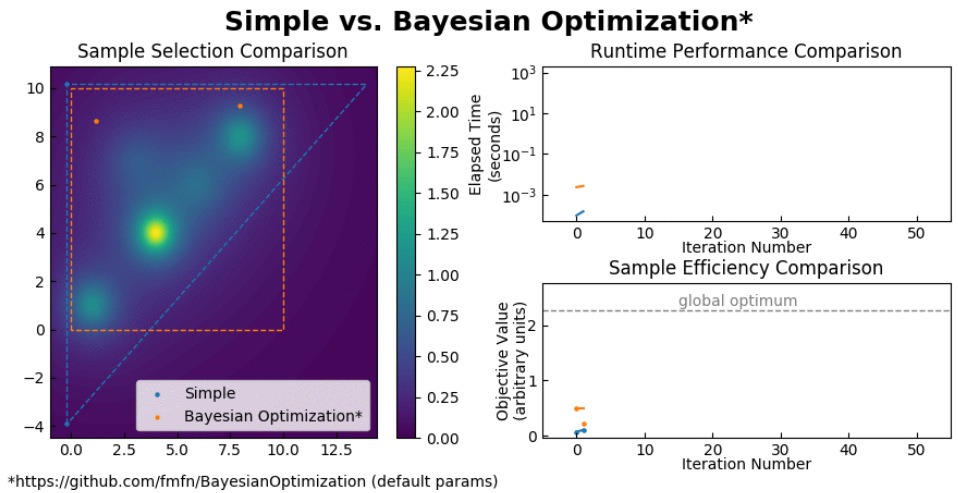
Frameworks #1

BlueYonder
a jda. company

Simple(x) Global Optimization

… to deal with a large number of sample and high dimension efficiently.

https://github.com/nestordemeure/Simplers



*https://github.com/fmfn/BayesianOptimization (default params)

source: https://github.com/chrisstroemel/Simple

argmin{}

Mathematical optimization in pure Rust

http://argmin-rs.org/

```rust
/// First, create a struct for your problem
#[derive(Clone, Default, Serialize, Deserialize)]
struct Rosenbrock {
    a: f64,
    b: f64,
}

/// Implement `ArgminOp` for `Rosenbrock`
impl ArgminOp for Rosenbrock {
    /// Type of the parameter vector
    type Param = Vec<f64>;
    /// Type of the return value computed by the cost function
    type Output = f64;
    /// Type of the Hessian. Can be `()` if not needed.
    type Hessian = Vec<Vec<f64>>;

    /// Apply the cost function to a parameter `p`
    fn apply(&self, p: &Self::Param) -> Result<Self::Output, Error> {
        Ok(rosenbrock_2d(p, self.a, self.b))
    }

    /// Compute the gradient at parameter `p`.
    fn gradient(&self, p: &Self::Param) -> Result<Self::Param, Error> {
        Ok(rosenbrock_2d_derivative(p, self.a, self.b))
    }

    /// Compute the Hessian at parameter `p`.
    fn hessian(&self, p: &Self::Param) -> Result<Self::Hessian, Error> {
        let t = rosenbrock_2d_hessian(p, self.a, self.b);
        Ok(vec![vec![t[0], t[1]], vec![t[2], t[3]]])
    }
}
```

Frameworks #2

BlueYonder
a jda. company

genevo

… provides building blocks to run simulations of optimization and search problems using genetic algorithms

https://github.com/innoave/genevo

```rust
/// The fitness function for `Selection`
impl<'a> FitnessFunction<Selection, i64> for &'a Problem {
    fn fitness_of(&self, selection: &Selection) -> i64 {
        ...
    }


    fn average(&self, values: &[i64]) -> i64 {
        (values.iter().sum::<i64>() as f32 / values.len() as f32 + 0.5).floor() as i64
    }


    fn highest_possible_fitness(&self) -> i64 {
        self.highest_possible_fitness
    }


    fn lowest_possible_fitness(&self) -> i64 {
        0
    }
}
```

oxigen

a parallel genetic algorithm library implemented in Rust

https://github.com/Martin1887/oxigen

```rust
impl Genotype<u8> for QueensBoard {
    type ProblemSize = u8;


    fn iter(&self) -> std::slice::Iter<u8> { ... }
    fn into_iter(self) -> std::vec::IntoIter<u8> { ... }
    fn from_iter<I: Iterator<Item = u8>>(&mut self, genes: I) { ... }


    fn generate(size: &Self::ProblemSize) -> Self { ... }


    fn fitness(&self) -> f64 { ... }


    fn mutate(&mut self, rgen: &mut SmallRng, index: usize) { ... }


    fn is_solution(&self, fitness: f64) -> bool { ... }
}
```

Source Code

BlueYonder
a jda. company

https://github.com/crepererum/aristotles_number_puzzle_genetic

Literature

- “Essentials of Metaheuristics”
  2013
  Sean Luke
  https://cs.gmu.edu/~sean/book/metaheuristics/

- “Learning Bayesian Network Structures by searching for the best ordering with genetic algorithms”
  1996
  Pedro Larrafiaga, Cindy M. H. Kuijpers, Roberto H. Murga, Yosu Yurramendi
  https://www.cse.unr.edu/~sjose/Papers%20Referenced/Learning%20Bayesian%20Network%20Structures%20by%20Searching%20for%20the%20Best%20Ordering%20with%20Genetic%20Algorithm.pdf