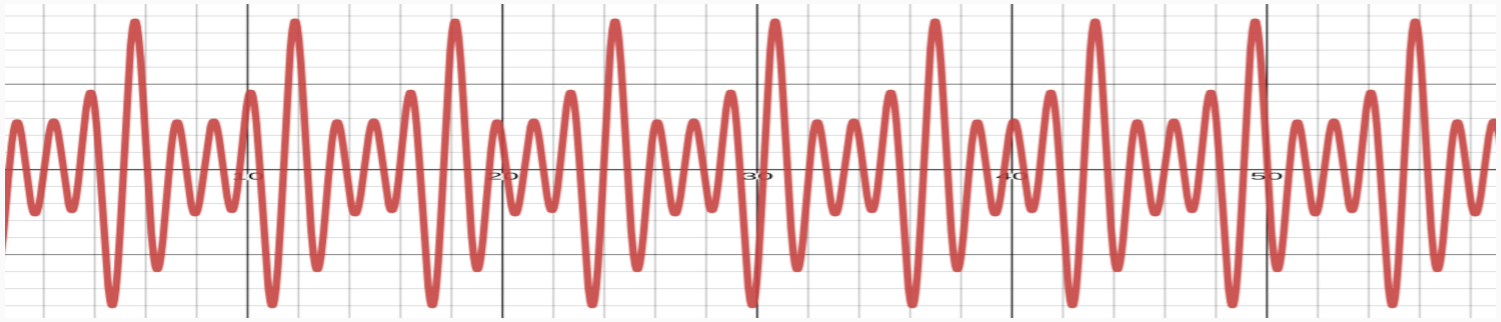
Genetic Programming



Prof. Fabrício Olivetti de França

Federal University of ABC

05 Februrary, 2024

UFABC

Evolution

> The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.

– George Bernard Shaw, *Maxims for Revolutionists*

Search and Optimization

Real-world problems

Some real-world problems are hard to solve:

- There are many possible solutions, we cannot enumerate them all
- We cannot formalize the problem description, thus requiring simplifications
- The evaluation criteria can be noisy, or change with time
- There are many constraints associated with the main problem

Search and Optimization Problems

A problem can be formalized as either a **search problem** or an **optimization problem**.

Search Problem

A **search problem** is when given a set of candidate solutions S and a property $P : S \to \{T, F\}$, we have to find s such that $s \in S, P(s)$.

SAT

The Boolean Satisfability problem (SAT) is described as:

> Given a boolean function $f(x)$, assign **true** or **false** values to each x_i such that the function evaluates to **true**.

SAT

For example:

$$f(x) = (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3)$$

SAT

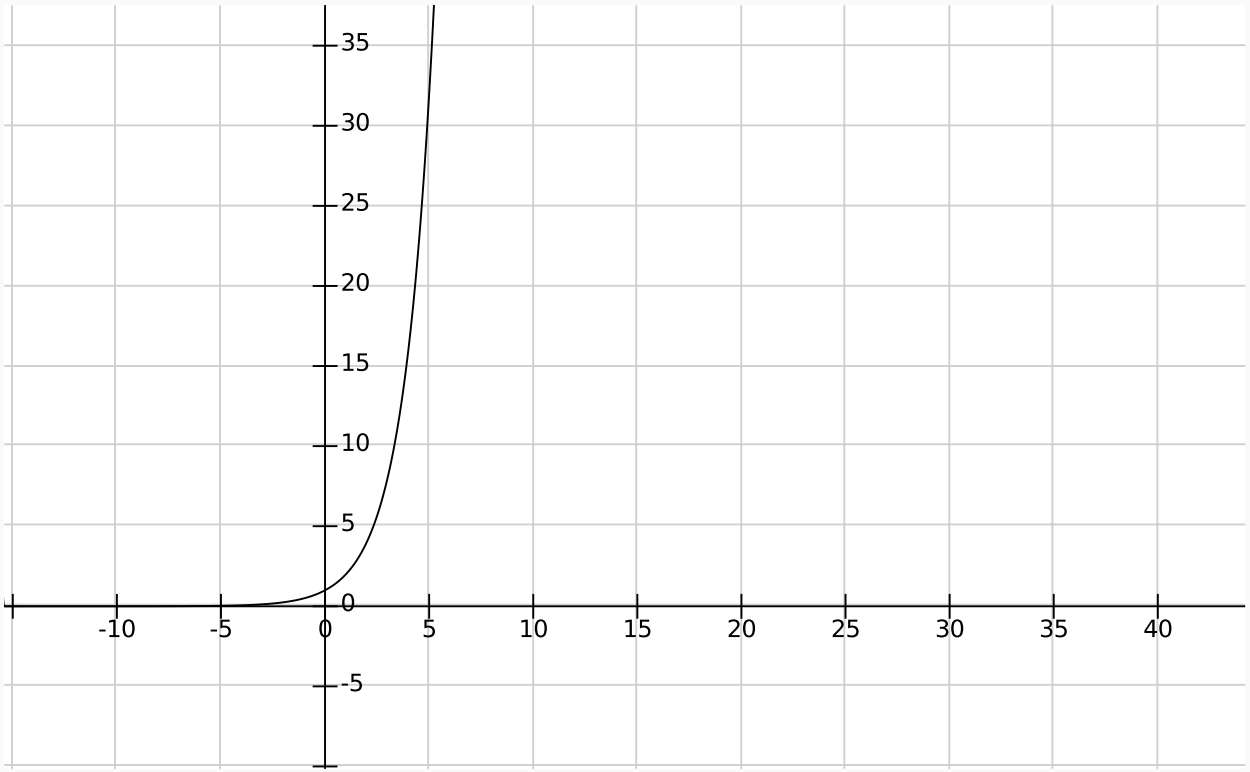
To make things simpler, we can use 0 and 1 to represent false and true, respectivelly.

With 4 variables, our set S has $|S| = 2^4 = 16$ candidate solutions.

x_1	x_2	x_3	x_4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1

...

As the number of variables grow, the cardinality of S grows quite fast:



SAT

n	2^n
1	2
5	32
10	1024
100	1.2e30
1000	1.1e301

Optimization problem

An **optimization problem** is when you have a set of candidate solutions X and a criteria function $F : X \rightarrow \mathbb{R}$ and you want to find $x \in X, x \in argmax_{y \in X} f(y)$, for maximization problems and $x \in X, x \in argmin_{y \in X} f(y)$ for minimization problems.

Optimization problem

maximization problems have the objective of finding one of possibly many solutions that has a maximum value for the criteria function.

minimization problems have the objective of finding one of possibly many solutions that has a minimum value for the criteria function.

$$G2(x) = \left| \frac{\sum_{i=1}^{n} cos^4(x_i) - \prod_{i=1}^{n} cos^2(x_i)}{\sqrt{\sum_{i=1}^{n} i x_i^2}} \right|$$

subject to

$$\prod_{i=1}^{n} x_i \geq 0.75$$

$$\sum_{i=1}^{n} x_i \leq 7.5n$$

$$0 \leq x_i \leq 10, 1 \leq i \leq n$$

Nonlinear Optimization

Conceptually there are an unlimited number of candidate solutions. But, when solving using floating point numbers in a computer, we have a (large but) finite number of solutions.

Assuming we can represent up till 6 decimal places, we would have $10.000.000$ distinct values for each variable x_i.

So we have $10.000.000^n = 10^{7n}$ candidate solutions.

Search and Optimization Problems

The main difference between search and optimization problem is that when you find a solution s, $P(s)$, we can stop the search and return the solution.

With an optimization problem, we have to check all possible solutions to be sure it is the best solution.

Basic Concepts

Search Space

search space is the set of all candidate solutions.

Solution representation

solution representation is how we conveniently represent a solution to our problem.

Objective-function

an **objective-function** is a function that maps a candidate solution to a (usually real) value measuring the quality of that solution. It can be a maximization or minimization function.

Neighborhood

the **neighborhood** of a solution s is the set of all solutions close to s.

Neighborhood

The neighborhood can be defined using a distance measure or a function that projects a solution to a power set.

Neighborhood

For example, for the nonlinear optimization problem, the representation is a vector $x \in \mathbb{R}^d$ and the neighborhood can be defined through a distance measure d:

$$d(x^{(1)}, x^{(2)}) = \sqrt{\sum_{i=1}^{d} (x_i^{(1)} - x_i^{(2)})^2}$$

$$\mathcal{N}(X) = \{y \in S \mid d(x, y) \leq \epsilon\}$$

Neighborhood

For the SAT problem, we can represent the solution as a binary vector and the neighborhood set is the function $\mathcal{N} : S \rightarrow 2^S$ that enumerates all set of solutions that swaps the value of two elements of the current solution.

Neighborhood

Fundamentally, a function $f : S \to 2^S$ is equivalent to $f : S \to S \to 2$.

That, uncurrying, is the same as $f : S \times S \to 2$.

In other words, it is a function that takes two solutions and returns 0 or 1 depending whether they are neighbors or not.

Local search

Local search algorithms start from an initial solution and iteratively walks through the neighborhood until it reaches a point without a better neighbor.

This point is called **local optimum**.

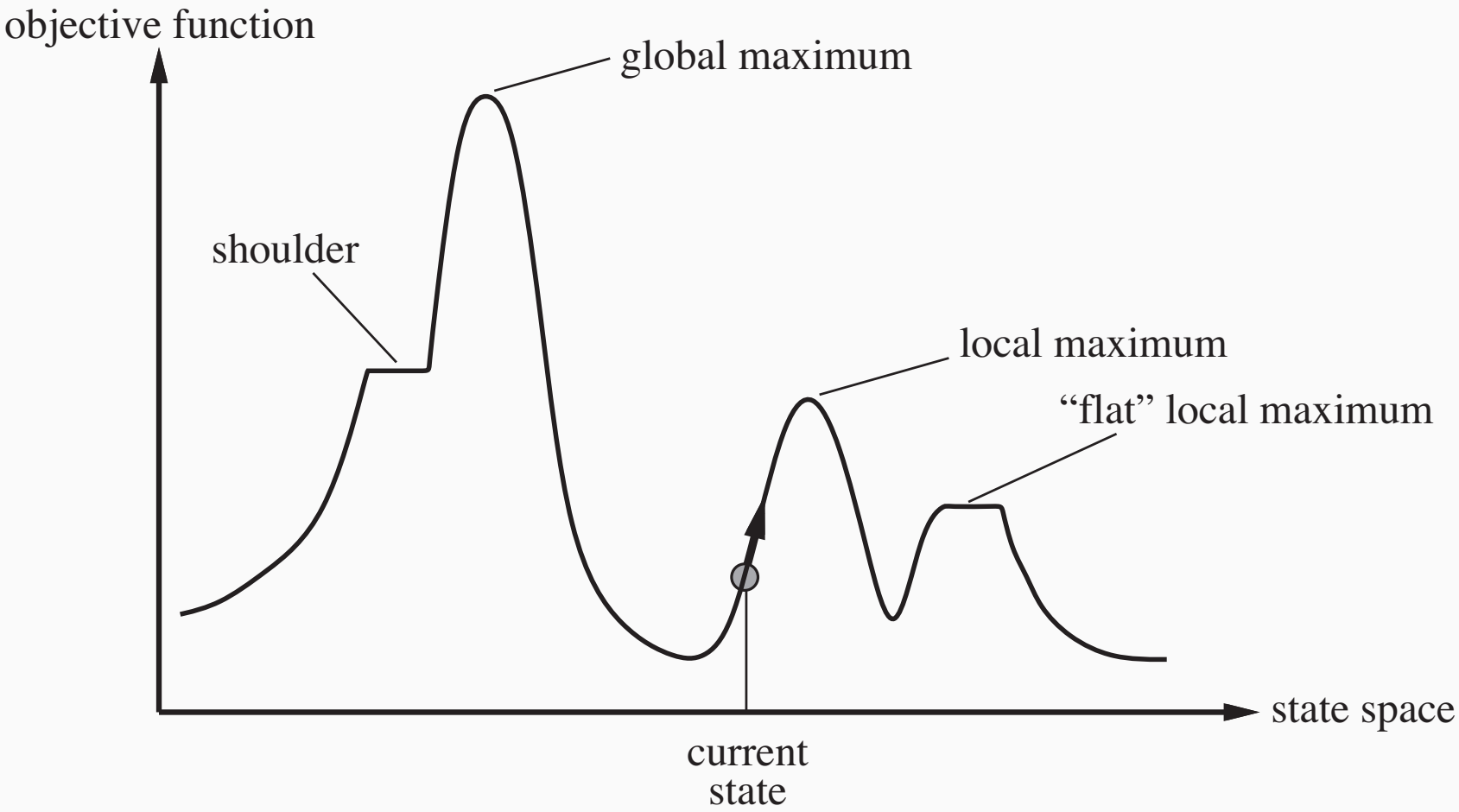
Local search

In some situations, we can guarantee that the **local optimum** is the best solution (or, **global optimum**).

For example, when maximizing $f(x) = -x^2$, iteratively walking through the neighborhood will always reach the global optima (under certain conditions).

Local search

But in many problems, depending on the amount of local optima, this procedure can get stuck at a subpar solution.



Hill-climbing

The **hill-climbing** algorithm or **steepest ascent**, repeats iteratively:

```
1  hill-climbing solution =
2    best-neighbor = best (neighbors solution)
3    if best-neighbor `better-than` solution
4      then hill-climbing best-neighbor
5      else return solution
```

Random Search

Consider now an algorithm that keeps generating and evaluating random solutions.

This is called a **random search**:

```
1  random-search =
2    solution = random-solution
3    return solution : random-search
4
5  get-best random-search
```

Complete and Optimal searches

A search algorithm is called **complete** if it guarantees to return a feasible solution (i.e., any s, $P(s)$).

A search algorithm is called **optimal** if it always return the global optimum.

Complete and Optimal searches

The Hill-climbing is neither complete nor optimal since it cannot guarantee any of these properties.

A random search is both complete and optimal since, given an infinite amount of time, it will eventually find the best solution.

Simulated Annealing

Since we don’t want to wait an infinite amount of time, we may try something in between Hill-climbing and random search.

Simulated annealing is one of such algorithm. It is essentialy a hill-climbing algorithm but the decision of whether to replace the current solution with one of the neighbors is not deterministic.

Simulated Annealing

```
1  simulated-annealing solution T =
2   | T <= eps = return solution
3   | otherwise =
4      neighbor = random-neighbor solution
5      next-solution =
6      if neighbor `isBetterThan` solution
7        || random <= exp (eval neighbor - eval solution)/T
8        then neighbor
9        else solution
10     return simulated-annealing next-solution (shrink T)
```

Simulated Annealing

The main idea of SA is to pick a random neighbor s' and accept it (i.e., replace current solution) if:

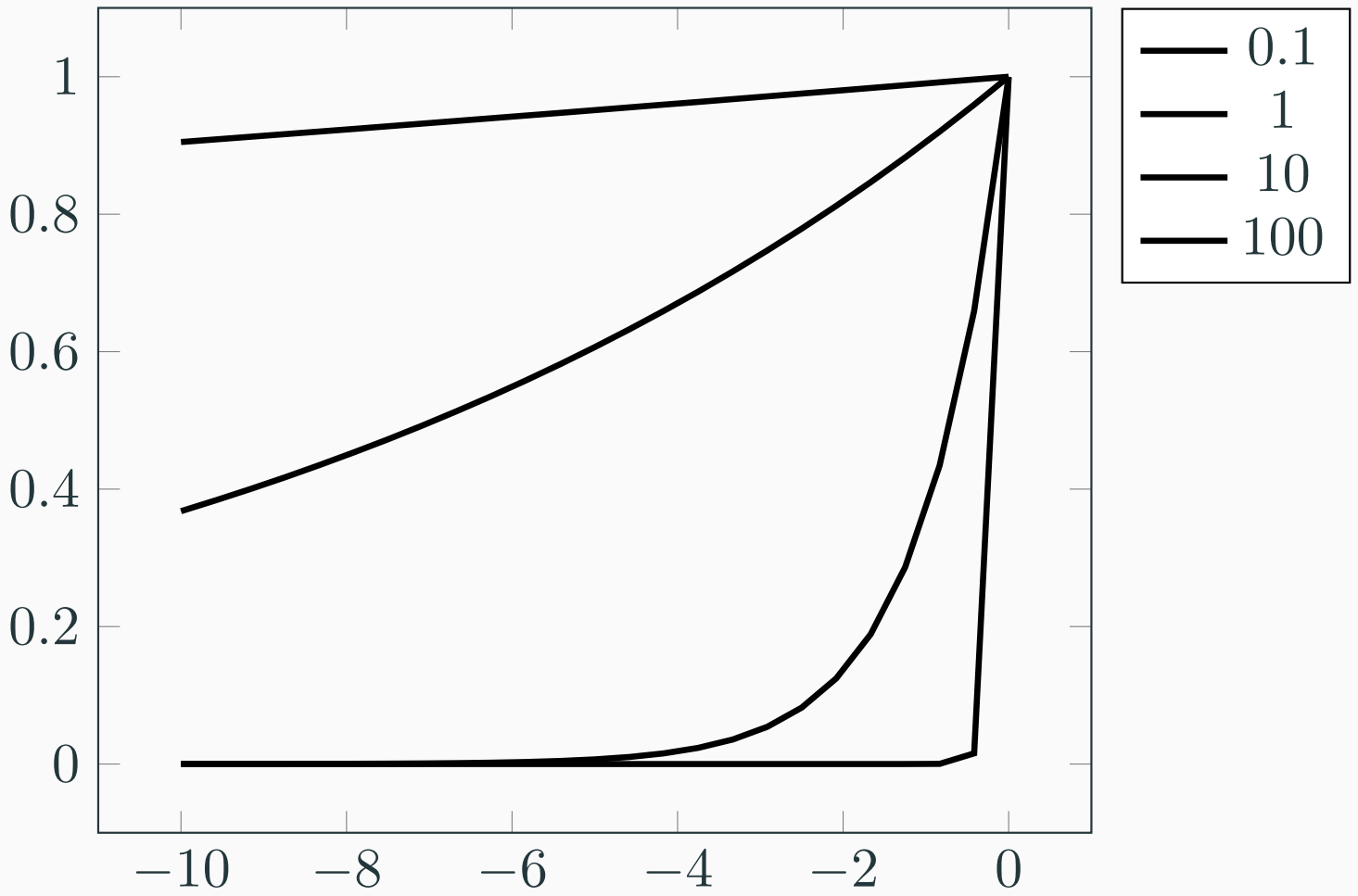
- This neighbor is better than the current solution or
- Accept with a probability of $e^{\frac{f(s')-f(s)}{T}}$

Simulated Annealing

Assuming maximization, if $f(s') = f(s)$ it will always replace as it has the same objective-function value.

If $f(s') < f(s)$ it will make the probability goes toward zero the higher the difference between both solutions. The value of T will determine how much worse we are willing to accept.

Simulated Annealing



Simulated Annealing

Starting with a large value of T, SA will behave similarly to a random search, accepting almost any solution.

As the value of T is reduced at every iteration, it will start to behave more like a hill climbing, as it will be less likely to accept a worse soluion.

Heuristic

Heuristic

Heuristic, from the greek *to find* or *discover*, are tecniques developed to find a solution to a problem without any guarantees.

Heuristic

The main goal of a heuristic method is to find an approximate solution as efficiently as possible.

Heuristic

George Pólya[1] enumerates some tips to create a heuristic to solve a problem:

- If you cannot understand the problem, draw a diagram representing it.
- If you cannot reach a solution from an initial state, try reaching the initial state from the solution.
- If the problem is abstract, try creating a concrete example.
- Try to solve a less restrictive problem first.

[1] How to Solve it, 1945

Greedy Heuristic

A **greedy heuristic** tries to iteratively build a single solution by maximizing the instant reward.

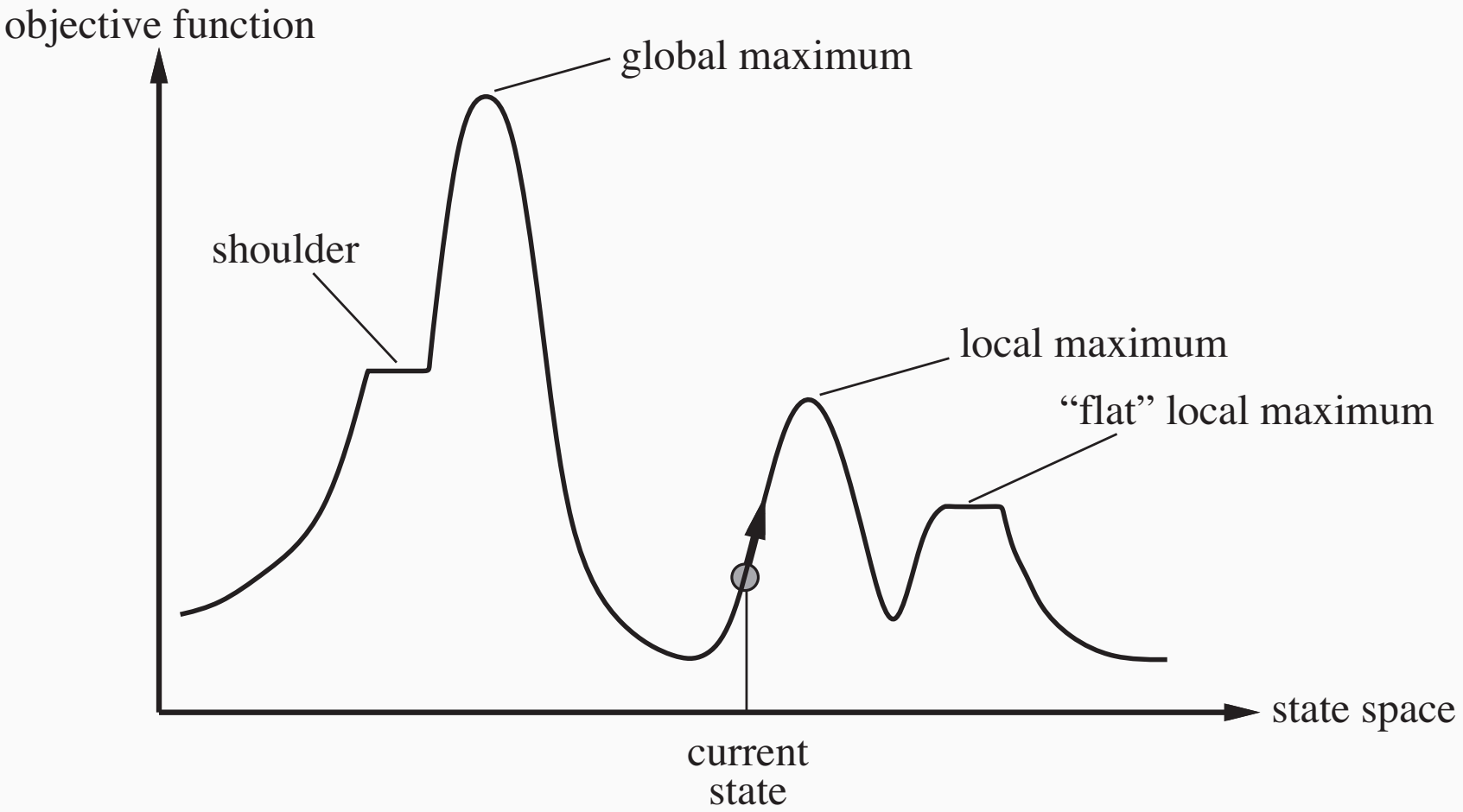
Greedy Heuristic

A possible greedy heuristic for the optimization problem is to fix every variable to a starting value, and optimize a single variable at every iteration.

Populational Heuristic

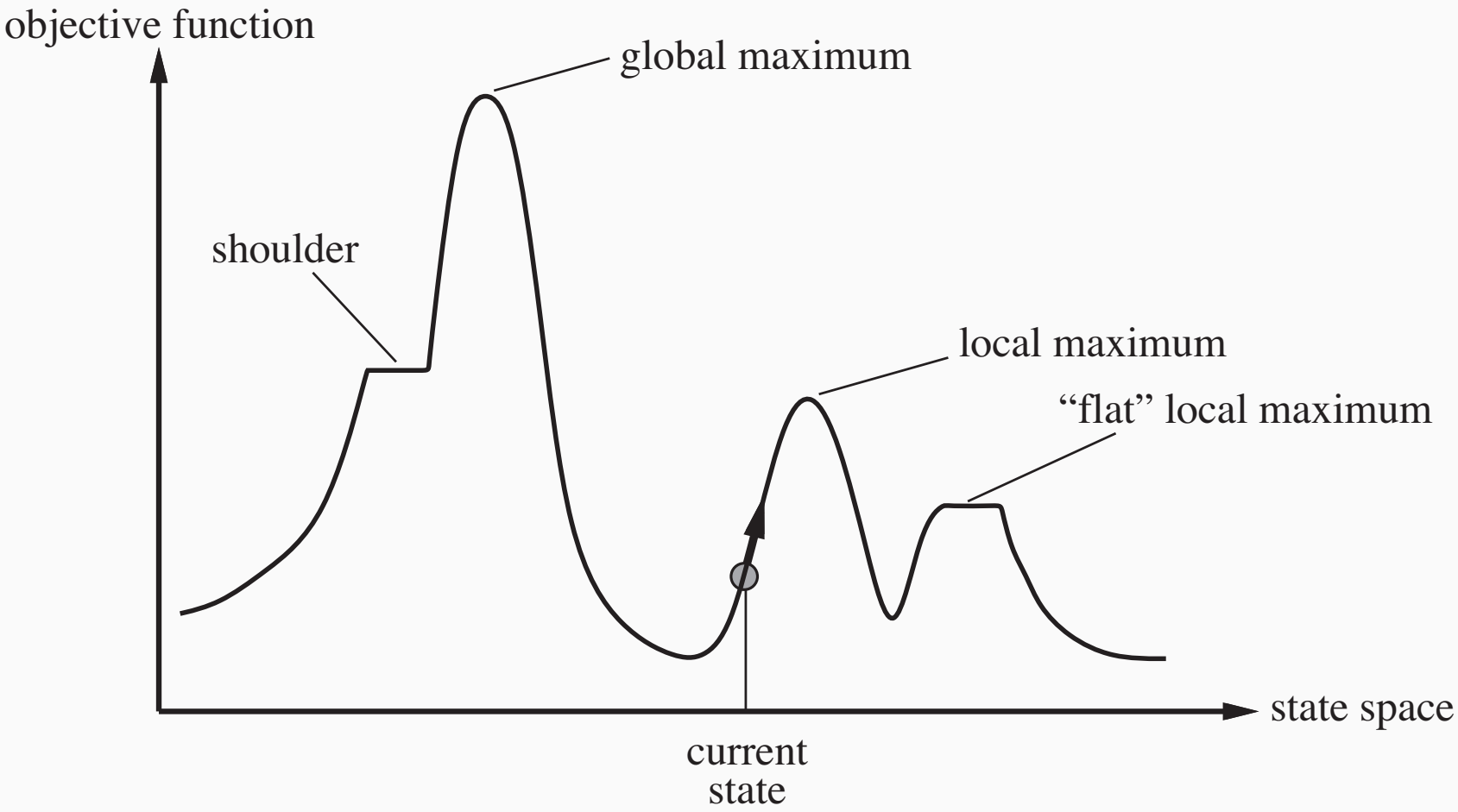
The local search and greedy heuristics so far worked with a single solution.

Their main limitation was to reach the closest optima to the starting point.



Populational Heuristic

What if we start at multiple points in parallel?



Populational Heuristic

By doing so we can reach multiple local optima at once, increasing our chances of finding the global optima.

We can explore this idea even further by creating a competition among the different solutions and focus the search only on the promising regions.

Populational Heuristic

Also, we can try to combine parts of different good solutions with the expectation of creating a better solution.

Populational Heuristic

The main idea of a populational heuristic is to find a balance between **exploration** and **exploitation**.

Populational Heuristic

exploration is the act of exploring the whole search space. Trying to find new promising regions.

Populational Heuristic

exploitation is the act of exploring only the local neighborhood to find the best solution in this smallr region.

Populational Heuristic

Both are equally important as we need to find promising regions while extracting the best from the current best regions.

Populational Heuristic

Let us suppose we create 30 random solutions for the SAT problem.

We can do that by sampling random bits with a probability $p = 0.5$ of sampling a 1.

Populational Heuristic

Next, we randomly select with replacement 30 solutions from this population with a probability proportional to the objective-function.

Populational Heuristic

From this selection, we will apply a random perturbation and repeat these procedures with this new population.

Populational Heuristic

For a SAT problem with 91 terms and 20 variables, in 30 executions with 200 solutions and a maximum of 1000 iterations, this simple procedure will

- Reach the goal in 24 of the executions
- Reach a solution on average after 272 iterations

Populational Heuristic

An abstract description of this algorithm is:

```
1  population-heurisitc =
2    pop = random-population
3    f   = eval pop
4    while not done
5      selection = selectWithReplacement pop f
6      pop       = perturb selection
7    return pop
```

One example of populational heuristic is the **evolutionary algorithm**.

Evolutionary Algorithms

Evolution

Evolution is a natural process in which a species adapts itself successively with the objective of:

- survival
- ecological balance
- diversity

Natural Selection

Natural Selection was proposed by Charles Darwin to explain why some characteristics become very common while others disappear.

Natural Selection

1. More offsprings are produced than necessary
2. Characteristics of an individual defines its probability of survival
3. Characteristics are hereditary

Evolution

Let’s illustrate with the constant fight for survivor of rabbits in a territory filled with foxes.

Evolution

In a population of rabbits, some are more clever and some are faster than others.

Those have more chance of survival and produce **more rabbits**.

Evolution

With fast rabbits reproducing with clever rabbits, new variants of rabbits appear:

- fast and dumb
- slow and clever
- slow and dumb
- fast and clever

Evolution

Nature sometimes throw a *crazy rabbit* in this population with some variation in the genes.

The offsprings are not exact copy of their parents, but random variations (remember the regression to the mean).

Evolution

Throughout the generations, hopefully the population of rabbits become faster and more clever than the initial population.

But the foxes also evolve…

Evolutionary Algorithms

```
1  p = initial-population
2  while not done
3    parents    = select-from p
4    offsprings = recombine-from parents
5    xmen       = mutate offsprings
6    p          = replace-from (p + xmen)
7  return p
```

Population

Each individual of the population represents a solution represented as seen fit.

In evolutionary algorithms this representation is called **chromosome**.

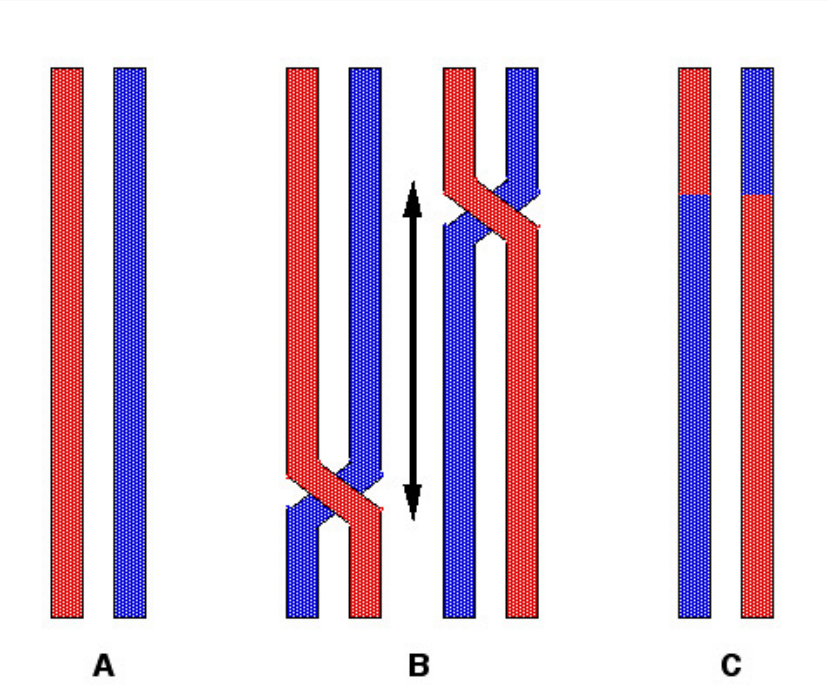
Selection

At this stage, the soon-to-be parents are selected from the population with a probability proportional to their fitness.

The idea is that the fittest have a higher chance of reproducing and passing their characteristics to their offsprings.

Recombination

Recombination simulates the reproduction of individuals of the same species where the exchange of genetic material happens.



Recombination

Assuming a vectorial representation:

```
parent1 = [3 0 2 | 5 2 1]
parent2 = [5 1 2 | 5 0 3]

child1 = [3 0 2 5 0 3]
child2 = [5 1 2 5 2 1]
```

Mutation

Mutation promotes novelty in our current population.

It prevents that every individual is the same.

Mutation

Mutation in this algorithm is a random perturbation in the solution representation:

```
child   = [3 0 *2* 5 2 1]
xman = [3 0 *4* 5 2 1]
```

Replacement

Once we have a population of offsprings, we will replace the current population with either the offsprings or a mix of the current population and their offsprings.

Replacement

Whenever we need to make a decision based on how fit an individual is, we use the objective-function or an adaptation of that to calculate the **fitness** of an individual.

Notice that conceptually the fitness is a maximization objective, but it is sometimes used as minimization for some problems.

Taxonomy

- **Genetic Algorithms** : proposed by Holland with the main goal of studying the adaptation phenomena.
- **Evolution Strategies**: introduced by Rechenberg to solve parameter optimization of nonlinear functions.
- **Evolutionary Programming**: represents a program as a finite state machine, proposed by Fogel at al.
- **Genetic Programming**: main goal of evolving computer programs, proposed by Koza.

Evolutionary Algorithms

Evolutionary Algorithms were proposed by different researchers in different forms, but with some common properties, they all

- Work with the idea of population of solutions
- Use selective pressure, with a likelihood of survival proportional to the fitness
- Measure the quality of each solution with a maximization objective-function
- Combine set of solutions thus expanding the local neighborhood
- Perturb selected solutions at random

This generates two fundamental forces in this class of algorithms:

- **Variation Operators:** promote diversity and search for new solutions (exploration).
- **Selective Pressure:** promote maintenance of good quality solutions (exploitation).

Terminology

- **Individual:** a solution represented by a *chromosome*.
- **Chromosome:** the computational representation of a solution, also known as **genotype**.
- **Phenotype:** the decoded representation.
- **Fitness:** a maximization objective-function that measures the quality of an individual.
- **Population:** a bag of individuals.
- **Crossover:** or **recombination**, a function that combines the information of two or more individuals.
- **Mutation:** a function that randomly changes one individual.

Selection Strategies

Selection proportional to fitness

In this strategy, assuming f_i the fitness of the i-th individual, the probability of selecting it as a parent is (assuming non-negative fitness):

$$p_i = \frac{f_i}{\sum_j f_j}$$

So the probability is proportional to the absolute value of the fitness.

Selection proportional to fitness

This strategy has some drawbacks:

- Individuals with much higher fitness may end up dominating the selection causing a **premature convergence**.
- When the fitness values are too close to each other, there is no selective pressure and the choice is almost uniformly at random.
- The probability of choosing an individual may be sensitive to small variations in fitness.

Selection proportional to fitness

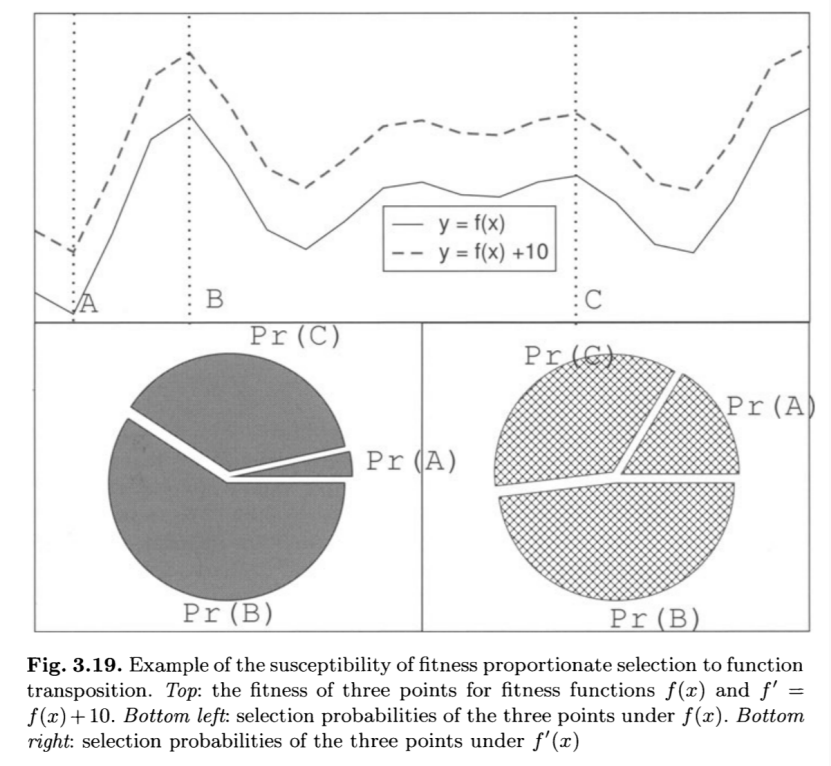


Fig. 3.19. Example of the susceptibility of fitness proportionate selection to function transposition. *Top*: the fitness of three points for fitness functions $f(x)$ and $f' = f(x) + 10$. *Bottom left*: selection probabilities of the three points under $f(x)$. *Bottom right*: selection probabilities of the three points under $f'(x)$

Figure 1: Eiben, Agoston E., and James E. Smith. Introduction to evolutionary computing. Springer-Verlag Berlin Heidelberg, 2015.

Selection proportional to fitness

We can alleviate some of these problems with the **sigma scaling** proposed by Goldberg:

$$f'_i = max(f_i - (\bar{f} - c\sigma_f), 0.0)$$

$$p_i = \frac{f'_i}{\sum_j f'_j}$$

where $\bar{f}, \sigma_f$ are the mean and standard deviation fitness of the population and c is a scaling constant, usually set to 2.

Rank Selection

Another strategy is to use the **rank** of the solutions instead of the fitness.

For example, the fitness:

$$[0.1, 19.3, 1.4, 0.05]$$

Would be mapped to:

$$[2, 4, 3, 1]$$

Rank Selection

To calculate the selection probability based on rank, we can use a linear or exponential scale:

$$rank_{linear}(i) = \frac{2-s}{\mu} + \frac{2(i-1)(s-1)}{\mu(\mu-1)}$$

$$rank_{exp}(i) = \frac{1-e^{-i}}{\sum_j 1-e^{-j}}$$

where i is the rank value, $1 < s < 2$ the scaling factor, μ is the highest observed rank.

Roulette Wheel Selection

The roulette selection assigns a region of a wheel to each individual, pick a random value, and returns the individual at that spot of the wheel.

$$P(x_i) = \frac{\text{fitness}(x_i)}{\sum_j \text{fitness}(x_j)}$$

```
1  spin fs = do
2    r     = random (0, 1)
3    wheel = cumsum(probability fs)
4    return (firstOf (>r) wheel)
```

Roulette Wheel Selection

If the fitness of our individuals are $[324, 1, 100, 289]$

The probability for each individual is $[0.454, 0.001, 0.140, 0.405]$.

Roulette Wheel Selection

Next we calculate the cummulative sum of these probabiilities, defining the slice of the wheel:

$$[0.454, 0.455, 0.595, 1.000]$$

Roulette Wheel Selection

Now we choose a random value $0 \leq r \leq 1$ and check which slice this value belongs to. The individual within this slice is chosen as one of the parents for reproduction.

$[0.454, 0.455, 0.595, 1.000]$

For $r = 0.3$ we choose the first individual.

For $r = 0.4541$ we choose the second individual.

SUS

The main problem with this approach is that by spinning the wheel multiple times we may not generate a representative sample of our population.

We can expand this idea by spinning the wheel with multiple arrows and sample multiple individuals at once. This is known as **stochastic universal sampling** (SUS)

SUS

```
1  sus n fs = do
2    r       = random(0, 1/n)
3    wheel   = cumsum(probability(fs))
4    choices = for [1..n] (\i -> firstOf (>(r*i)) wheel)
5    return choices
```

This strategy guarantees that the number of times the i-th individual is selected is at least the integral part of $n \cdot P(i)$.

Tournament Selection

The tournament selection simply samples k individuals of the population and returns the best among them.

This is a local selection strategy, since it only uses the knowledge about the k sampled individuals as opposed to the roulette wheel that requires knowing the fitness of the entire population.

Tournament Selection

An important characteristic of this strategy is that it is invariant to translation and transposition. The choice of fitness does not affect the results of this method.

Tournament Selection

The probability that an individual is selected depends of:

- Its rank in the population
- The size k of the tournament (the higher the size, the higher the bias to above average individuals)
- If the individuals are selected with or without replacement (without replacement, the $k - 1$ worse will never be selected)

Tournament Selection

```
1  tournament k pop = do
2    competitors = sampleWithReplacement(k, pop)
3    return (best competitors)
```

Lexicase Selection

Most selection strategies so far compare individuals using an aggregated score (i.e., the fitness).

In some situations, we can evaluate the individual locally.

For example, in regression we can calculate the squared residue for a single point or the sum of squared residues for a selection of the sample.

Lexicase Selection

Evaluating the aggregated value we stimulate **generalists**. But during the evolution process we may want to combine two or more **specialists**.

Lexicase Selection

Assuming maximization (fitness) and xy is the list of tuples (x, y) from our sample:

```
1  lexicase pop xy =
2    pool  = pop
3    cases = shuffle xy
4    return (while-loop pool cases)
5
6  while-loop pool (case:cases)
7    | sizeOf pool  == 1 = return (head pool)
8    | sizeOf cases == 0 = return (random pool)
9    | otherwise =
10     best  = max (partial-fit case) pool
11     pool' = filter (==best . partial-fit case) pool
12     while-loop pool' cases
```

Lexicase Selection

When our fitness is continuous and we want some tolerance between two very similar individuals, we can use the ϵ-lexicase selection.

In this version, we simply replace the comparison with the best value with a comparison to whether the difference between the best value and the current value is within an ϵ radius.

Replacement

Evolutionary Algorithms often works with one of two popular models: **generational** and **stationary**.

Generational

Generational model in a population of size μ we choose μ parents and generate $\lambda = \mu$ offsprings that entirely replace the parent population.

Stationary

In the stationary model, we generate $\lambda < \mu$ offsprings to replace some of the current solutions in the population. A special case is when $\lambda = 1$.

Replacement

When using the stationary model, we have to select which individuals will survive and which will be replaced.

Fitness based replacement

In the fitness based replacement, we use one of the selection strategies to select the next generation.

Replace the worst

In this strategy we pick n worse individuals of the current population and replace with the offsprings.

Elitism

In this strategy, we keep the best n individuals in the population and use another strategy to select the remainders.

Multi-objective

Sometimes we want to optimize more than one objective and they are conflicting with each other (if we improve one, we make the other worse).

For these situations we can apply a Multi-objective version of the evolutionary algorithms.

The main difference is in the selection and reproduction operators in which the comparison between individuals are made through the dominance operator instead of equality.

Multi-objective

In multi-objective optimization we evaluate the individuals using $m > 1$ objectives. As such, we have to redefine how we compare two solutions.

Instead of saying one solution is better than another, we say that one solution dominates another (denoted by $\mathbf{f}(\mathbf{x_1}) \prec \mathbf{f}(\mathbf{x_2})$) if both:

$$\forall i \in \{1 \dots m\} : f_i(\mathbf{x_1}) \leq f_i(\mathbf{x_2}),$$
$$\exists i \in \{1 \dots m\} : f_i(\mathbf{x_1}) \neq f_i(\mathbf{x_2}),$$

are true considering a minimization problem.

Multi-objective

In short, this means that a certain solution dominates the other if it is equal or better in every objective and better in at least one of them.

The Pareto optimal set is defined as the set of optimal solutions that are not dominated by any other:

$$\mathbb{P} = \{\mathbf{x}^* \in \Omega \mid \nexists \mathbf{x} \in \Omega : \mathbf{f}(\mathbf{x}) \prec \mathbf{f}(\mathbf{x}^*)\}.$$

And the Pareto front is represented as the image of this set:

$$\mathcal{F} = \{\mathbf{f}(\mathbf{x}) : \mathbf{x} \in \mathbb{P}\}.$$

Multi-objective

A Multi-objective approach for evolutionary algorithms is the Fast Non-dominated sorting algorithm (NSGA-II)[2] together with the Crowding Distance.

The only changes in the algorithm is the replacement step that takes the dominance into consideration

[2] Deb, Kalyanmoy, et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." IEEE transactions on evolutionary computation 6.2 (2002): 182-197.

Multi-objective

```
1  replacement pop =
2    S = [filter (dominatedBy i) pop | i <- pop]
3    n = [lengthIf (dominates i) pop | i <- pop]
4    pop' = [i | i, ni <- enumerate n, ni == 0]
5    case length pop'
6      == length pop -> return pop'
7      >  length pop -> crowding pop' (length pop)
8      <  length pop -> return (pop' + replacement (pop / pop'))
```

Multi-objective

```
crowding pop nPop =
  dists = array 0 `ofSize` nPop
  for j <- [0 .. nObjs]
    ixs = argsortByObj j pop
    for (prev, i, next) <- window ixs
      dists[i] = dists[i] + (dists[next] - dists[prev])
    dists[first ix] = inf
    dists[last ixs] = inf
  ixs = argsort (-dists)
  return (take nPop ixs)
```

Genetic Programming

Evolving Programs

Is it possible to evolve a computer program using just a sample of inputs and outputs?

What is a program?

A program can be thought as a function that gets one or more input arguments and return a value of a certain type.

What is a program?

This function can be decomposed in elementa functions that belong to our grammar:

```
countElements [1, 2, 1, 3, 2, 1] = [(1,3), (2,2), (3,1)]
```

```
countElements = sort => group => map (head, length)
```

What is a program?

Depending of our context, the grammar can be reduced!

A classification problem has the following structure:

```
if predicate
  then class1
  else class2
```

What is a program?

And the predicate is simply a boolean function:

```
x1 > 3 && x2 <= 5
```

What is a program?

We only need to evolve the predicate function! We have `>`, `>=`, `<`, `<=`, `==`, `!=`, `number`, `var`, `&&`, `||`, `not` as part of our grammar.

What is a program?

A mathmatical expression is also a program:

```
f(x) = x[1]^2 - x[2]*x[3]*cos(pi*x[1])
```

Evolving a program

To evolve a program we have first to define the set of functions and terminals:

Name	Set
Function	$\{+, -, *, /, ^2\}$
Terminal	$\mathbb{R} \cup \{x1, x2, x3\}$

Evolving a program

Each function requires a number of input arguments, this number is called **arity**.

The function $+$ has arity 2, the partially applied function ^{2} has arity 1.

The function `if-then-else` has arity 3.

Evolving a program

These sets must obey the formal language rules:

- Every element of the terminal set T must be a valid and correct expression.
- If $f \in F$ is a function with arity n and $e_1, e_2, \ldots, e_n$ are valid and correct expressions, then $f\,e_1 \ldots e_n$ is also a valid and correct expression.
- There are no other correct form besides these.

Evolving a program

It is also possible that the expression contains the type information.

A typed function must get the arguments of the correct type.

The expression $e_1 \vee e_2$ requires that e_1, e_2 are booleans.

Genetic Programming

The main evolutionary algorithm that evolves program is called **Genetic Programming** (GP).

Genetic Programming

```
1  gp =
2    pop = randomPopulation
3    until convergence do
4      children = empty
5      until length(children)==lambda do
6        mut? = random(0,1)
7        if mut?
8          then ix       = random(0, n)
9               child    = mutate(pop[ix])
10              children = children <> child
11         else (p1, p2) = randomParents(pop)
12              (c1, c2) = combine(p1, p2)
13              children = children <> [c1, c2]
14     pop = replace pop children
15 return pop
```

Evaluating program

The fitness of a program is proportional to the amount of the test cases that passes within the training data.

If the output is a continuous value, we can measure the absolute or square difference between generated and expected outputs.

Partial Functions

If the representation allows invalid programs, there is a need to apply some treatments to either fix the program or protect the output.

For example, it is common to use the analytical quotient instead of division:

$$aq(a, b) = \frac{a}{\sqrt{1 + b^2}}$$

Representação

There are many ways to represent a program:

- Linear
- Tree
- Direct Acyclic Graph

Linear Representation

With the linear representation, a program is reprenseted in its imperative form with state changes. In other words, a source code similar to an assembly language.

Linear Representation

This representation allows the programs to be coded as *bytecodes* allowing us to use the common mutation and recombination of binary representation.

Linear Representation

The Gene Expression Programming[3] algorithm represents a program as an array of pre-fixed size:

$$Q * - + abcd$$

[3] Ferreira, Candida. "Gene expression programming: a new adaptive algorithm for solving problems." arXiv preprint cs/0102027 (2001).

Polish Notation

We can also represent a function in polish notation:

$$*+x2y = (x+2)*y$$

Graph representation

The algorithm *Cartesian Genetic Programming*[4] represents a program as a graph:

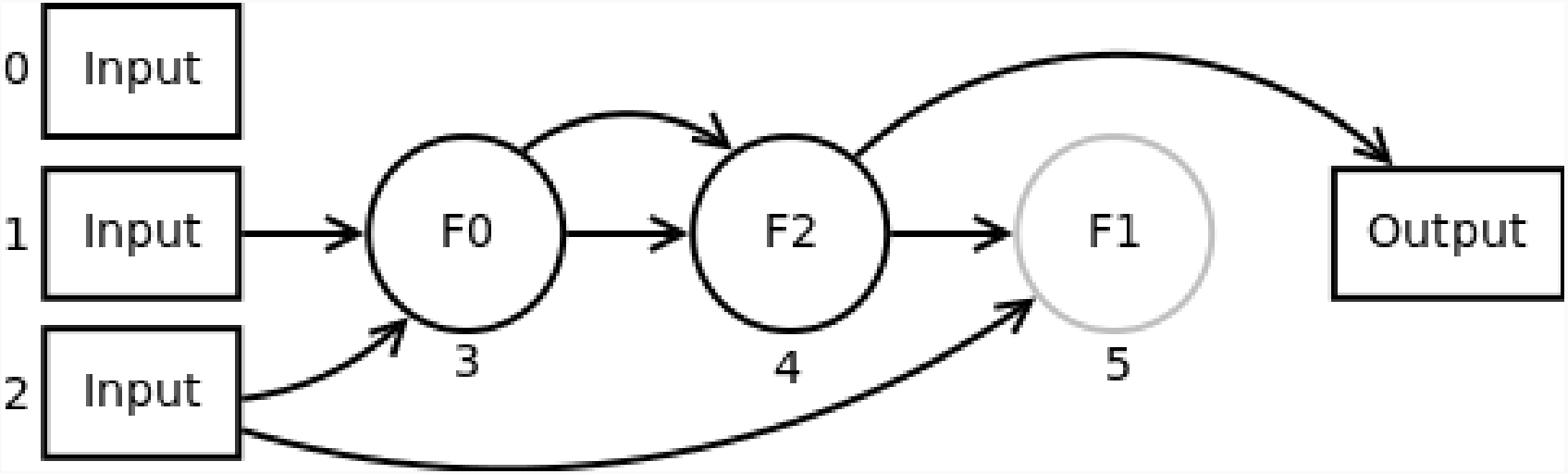


Figure 2: FONTE: http://www.cgplibrary.co.uk/files2/CartesianGeneticProgramming-txt.html

[4]Miller, Julian Francis, and Simon L. Harding. "Cartesian genetic programming." Proceedings of the 10th annual conference companion on Genetic and evolutionary computation. 2008.

Tree Representation

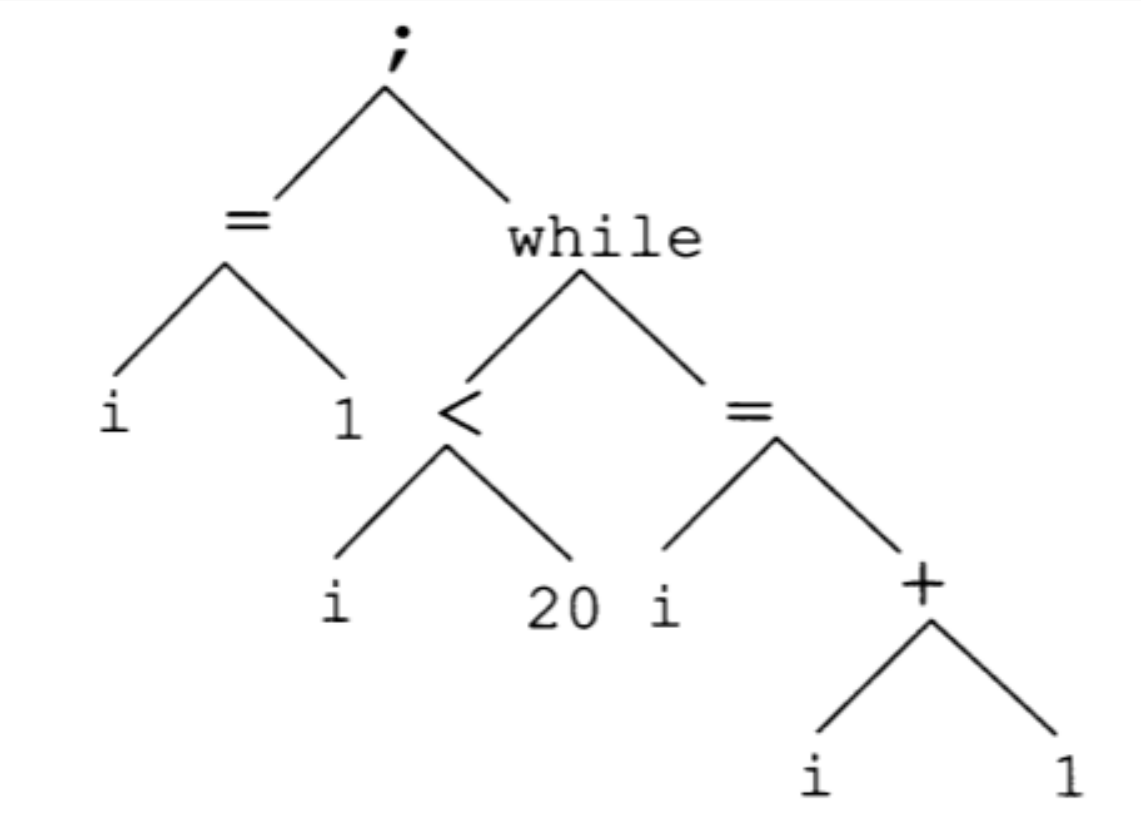


Figure 3: Fonte: Eiben, Agoston E., and James E. Smith. Introduction to evolutionary computing. Springer-Verlag Berlin Heidelberg, 2015.

Logic Tree

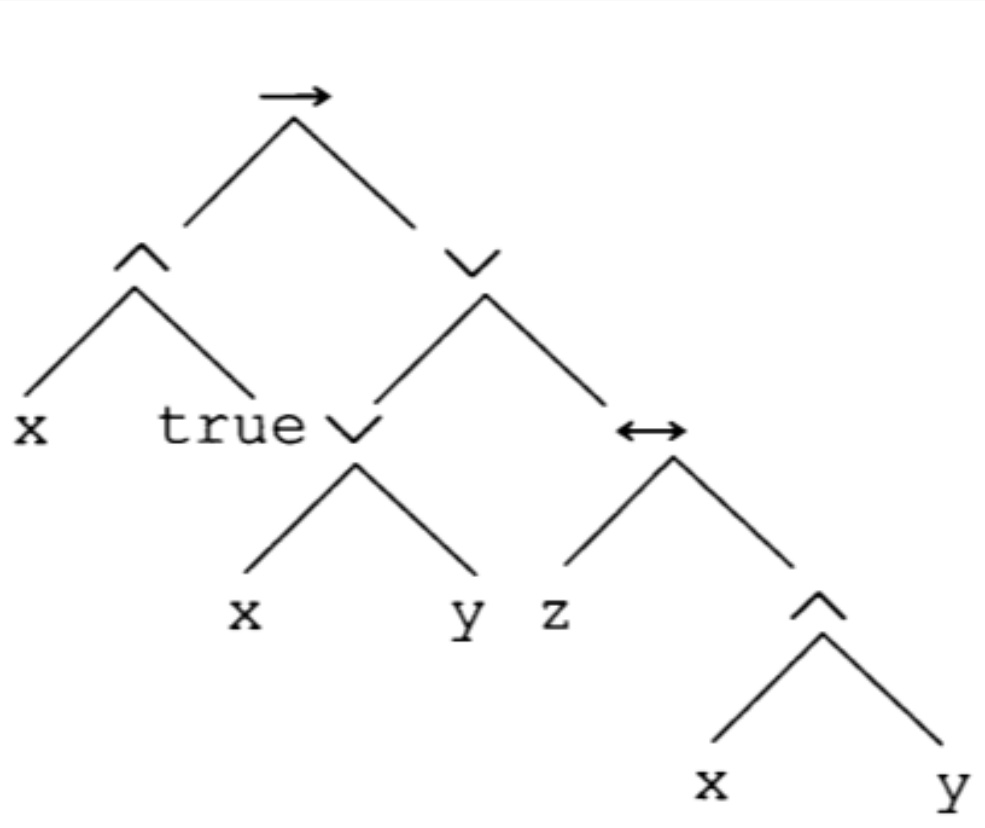


Figure 4: Fonte: Eiben, Agoston E., and James E. Smith. Introduction to evolutionary computing. Springer-Verlag Berlin Heidelberg, 2015.

Expression Tree

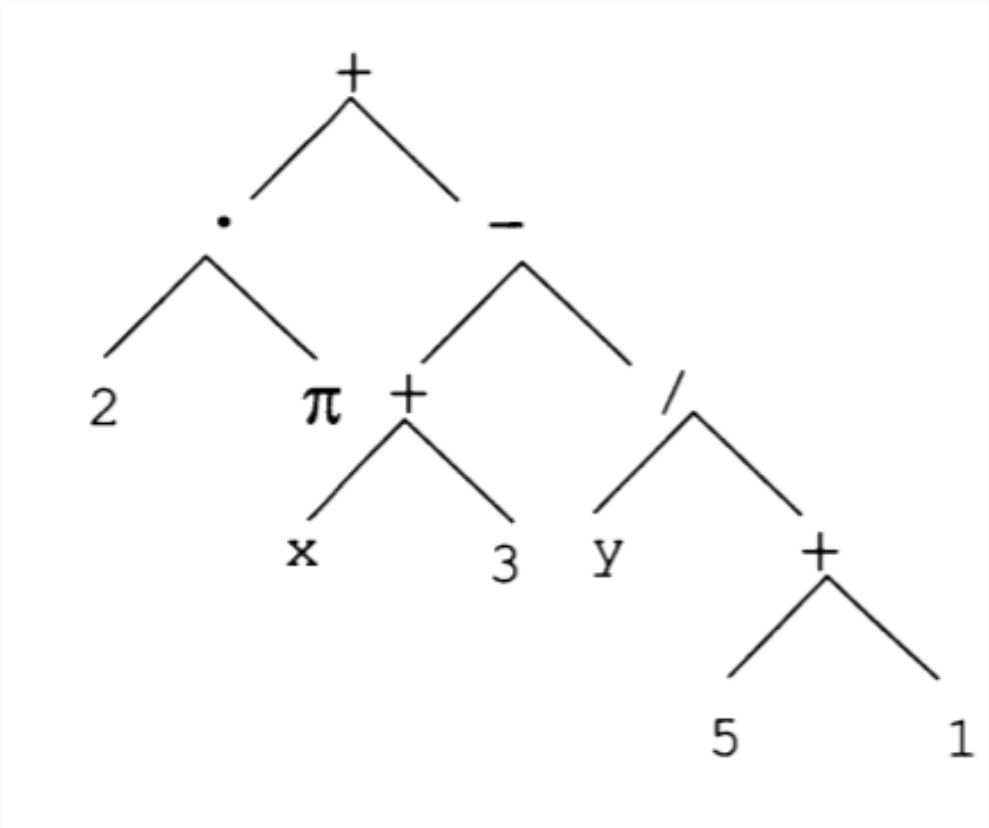


Figure 5: Fonte: Eiben, Agoston E., and James E. Smith. Introduction to evolutionary computing. Springer-Verlag Berlin Heidelberg, 2015.

Initialization of a Tree

When creating an initial solution, we must be careful that we create a correct expression (i.e., the leaves are terminals) and that the program is not too large.

Full method

In the **full** method, we create a complete tree with depth d. In other words, all of the branches of the tree must have the same depth.

Método Full

```haskell
full max-depth =
  node = if max-depth == 0
           then sampleTerm
           else sampleNonTerm
  children = [full (max-depth - 1) | _ <- [1 .. arity(node)]]
  return (Node node children)
```

Grow Method

The **grow** method freely generates a tree up to a maximum depth d, where it will sample only terminals.

Before reaching the maximum depth, the sample of a node is biased toward non-terminals to avoid short programs.

Método Grow

```
1  grow max-depth =
2    ratio = n_terms / n_symbols
3    r     = random(0,1)
4    node  = if max-depth == 0 or r < ratio
5              then sampleTerm
6              else sampleNonTerm
7    children = [grow (max-depth - 1) | _ <- [1 .. arity(node)]]
8    return (Node node children)
```

Ramped Half-and-Half

In **Ramped Half-and-Half** we create a population of initial solutions by varying the application of the previous methods with different values of maximum depths.

If we want to create n individuals, we use full in half of them and grow to the other half. For each method, we choose a maximum depth from a range $[min\_depth, max\_depth]$ uniformly distributed.

Ramped Half-and-Half

```
ramped min-depth max-depth n-pop =
  range     = max-depth - min-depth + 1
  n         = n-pop / 2  -- divisão inteira
  (q, r)    = (n / 2, n % 2)
  treesFull = [full min-depth | _ <-[1..q]]
  treesGrow = [grow min-depth | _ <-[1..q+r]]
  trees     = ramped(min-depth+1, max-depth, n-pop - n)
  return (treesFull + treesGrow + trees)
```

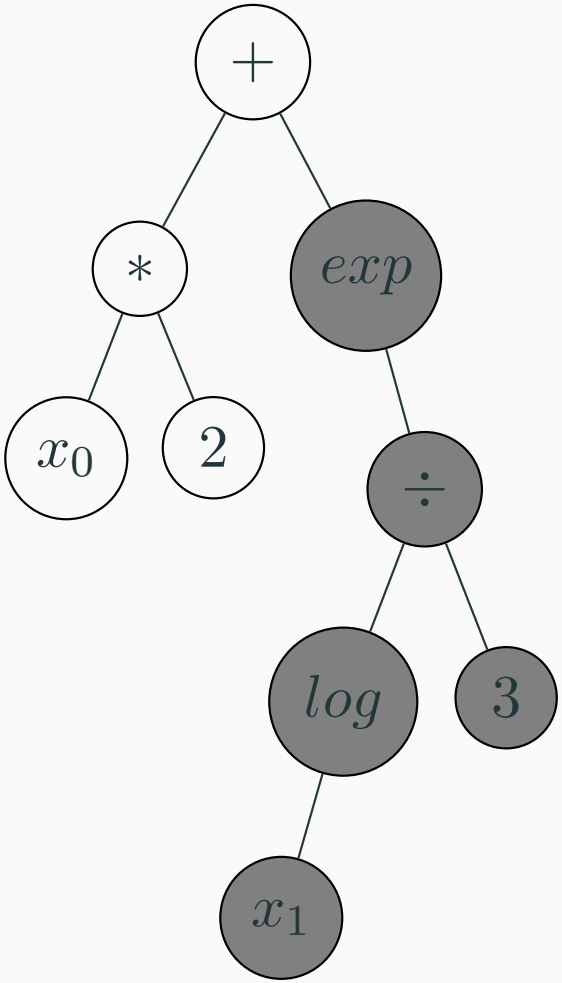
Mutation

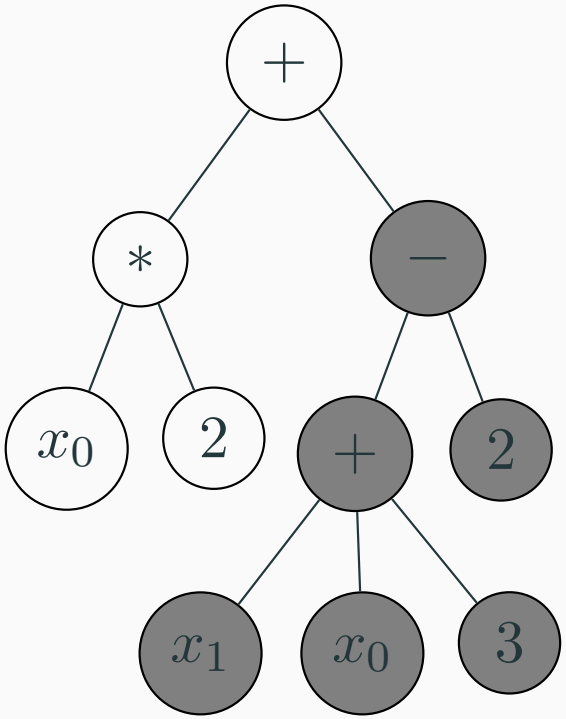
Subtree Replacement

In this mutation operator we choose a subtree and replace it by a new randomly generated tree.

We can use either the grow or full method to create this random subtree.

Subtree Replacement

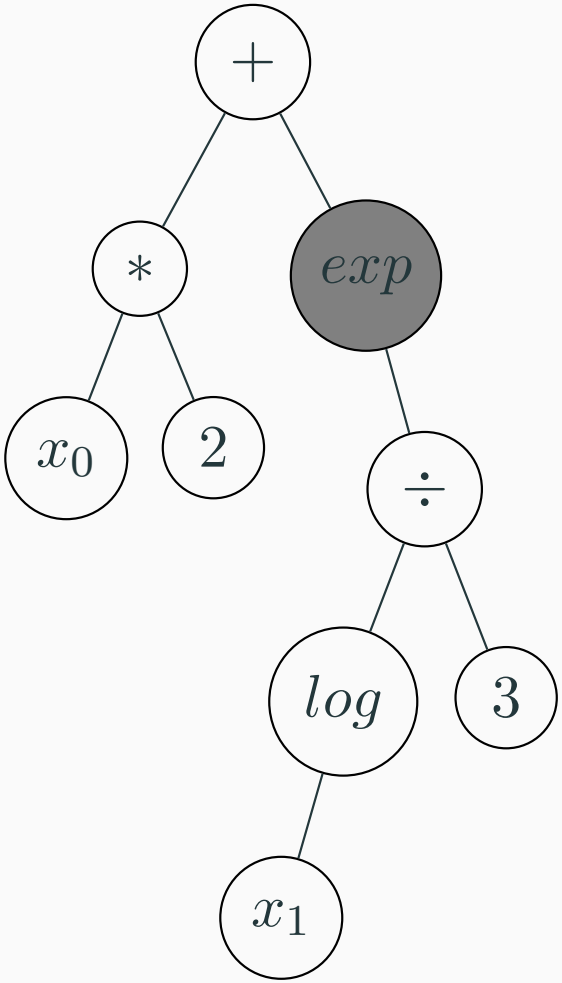


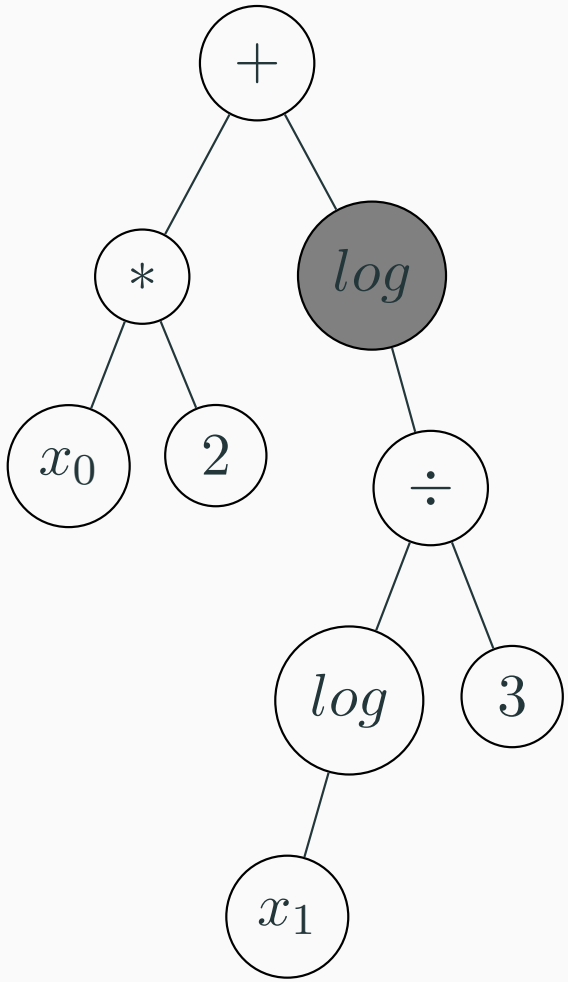


Node Mutation

With node mutation, we choose a node and change it with another token of the same arity.

Node Mutation

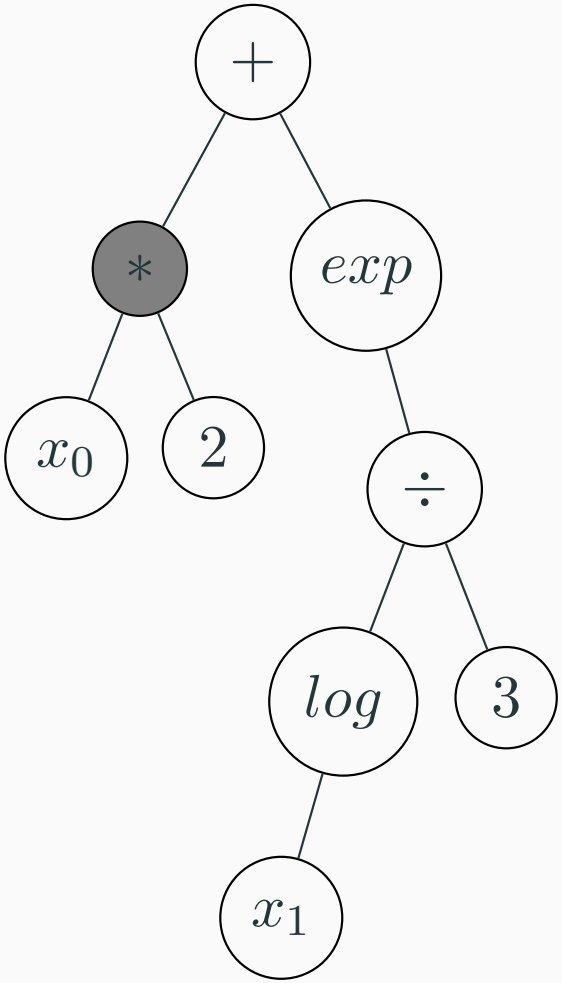


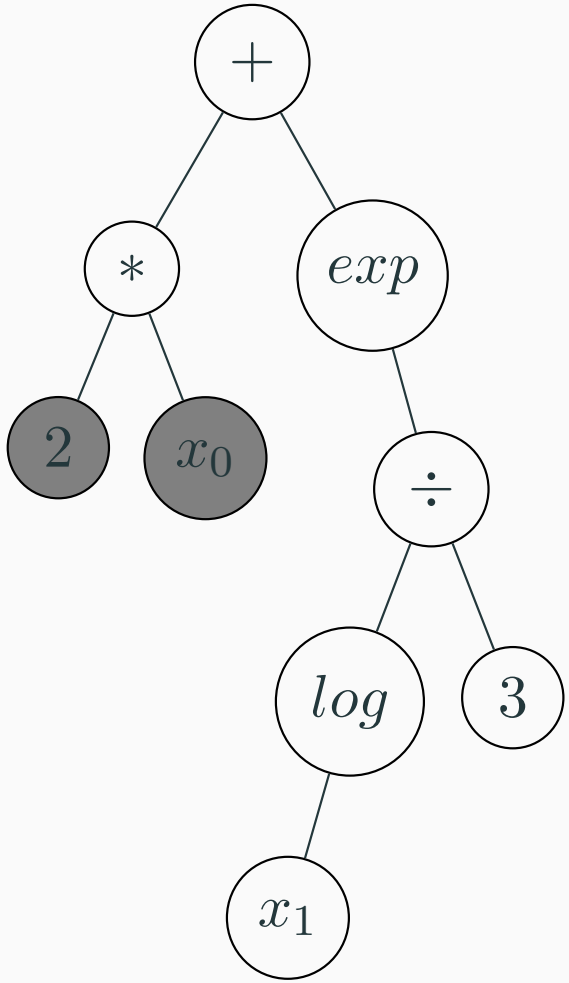


Swap Mutation

In swap mutation, a non-terminal node is chosen and its children are swapped (if they are of the same type).

Swap Mutation

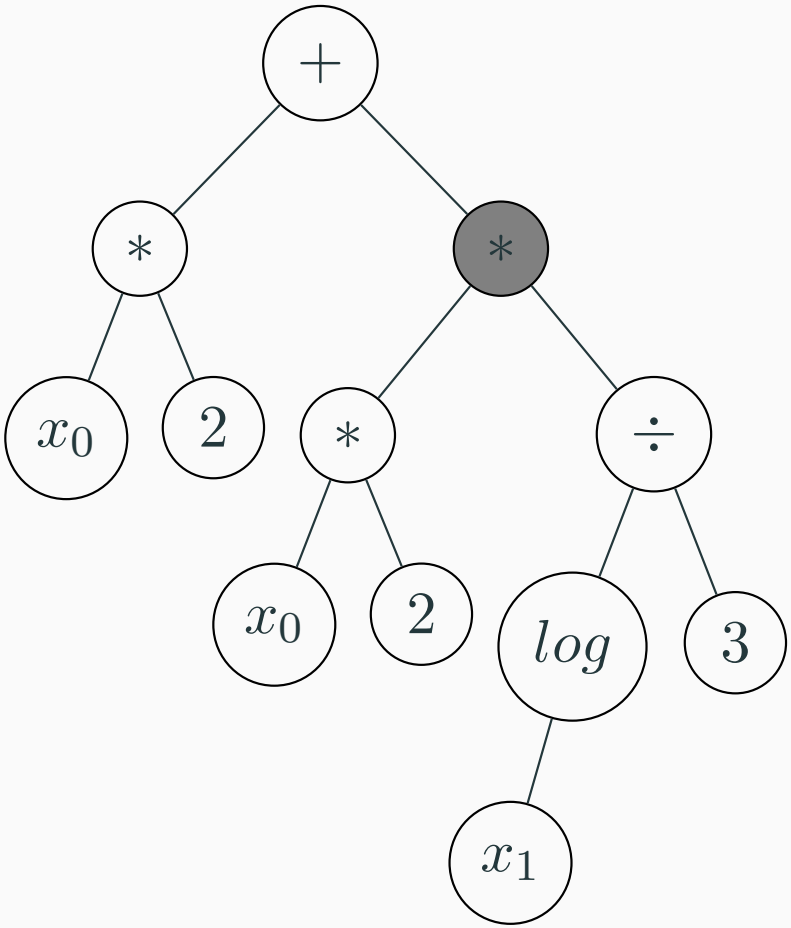


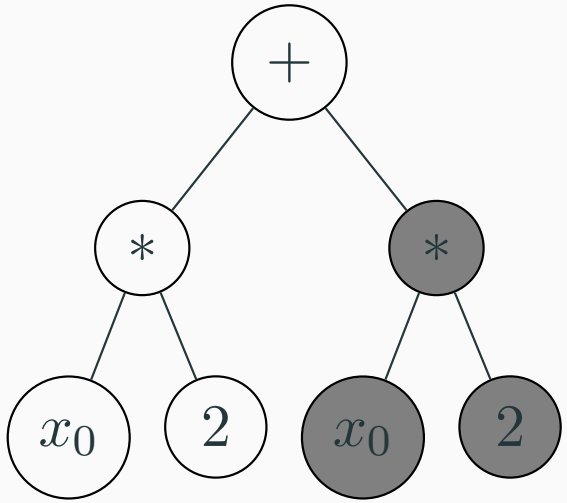


Shrink Mutation

In shrink mutation, a random nonterminal node is replaced by one of its children.

Shrink Mutation

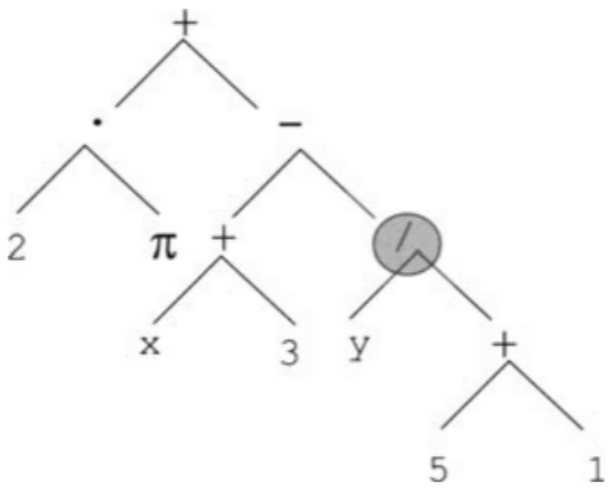




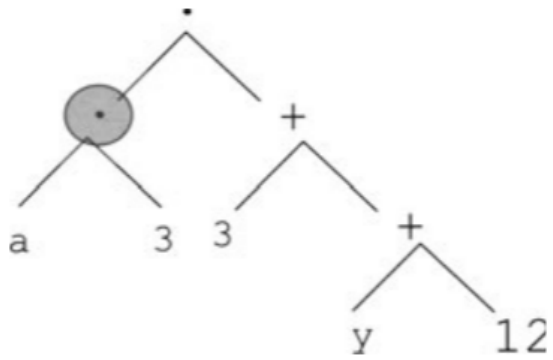
Recombination

The recombination operator simply chooses a subtree of each parent and swap them.

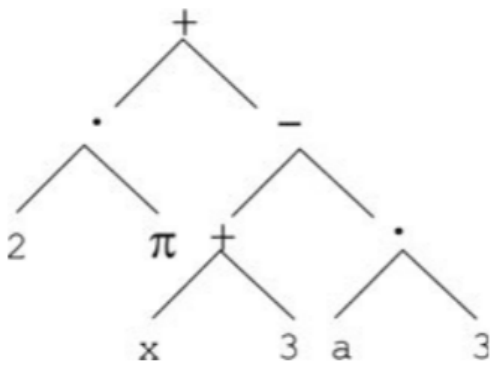
Recombination



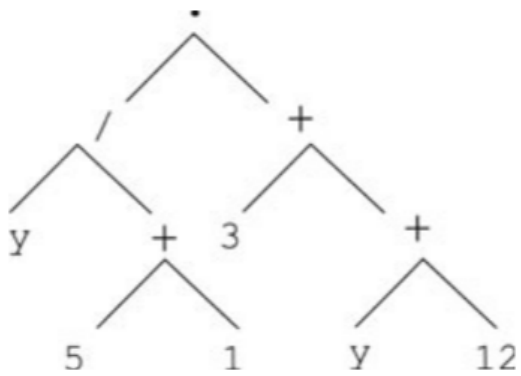
parent 1



parent 2



child 1



child 2

Bloat

Bloat

In the initial generations, it is common to observe the increase of the average fitness of the population together with the increase of the size of the tree.

At a certain point, it is possible to observe the increase in size without any increase in the average fitness.

Bloat

This phenomenon is known as **bloat** and it may be a problem since the computational cost to evaluate a larger program is higher.

Not only that, but the program loses its interpretability potential.

Bloat

A hypothesis for this is the **replication of accuracy theory** which says the ability to generate a child solution that is functionally similar to its parents favors its replication in the population.

Bloated expressions favors this property.

Bloat

In the **removal bias theory**, we notice that a tree can have inactive codes. When we apply reproduction and mutation in those inactive subtrees, there is a chance of increasing the size without any benefit to the fitness.

Bloat

The **programs search space theory** says that after a certain size, the average fitness will not change with size.

The reproduction of large programs tend to create even larger children which propagates throughout the generation favoring a population of large programs.

Discarding bloat

One way to deal with bloat is to disallow the generation of programs of a certain size.

If a children is larger than the maximum allowed size, it is discarded.

Discarding bloat

The main problem of this solution is that those trees close to the maximum allowed size will have many copies in the population as their children will likely be discarded.

Discarding bloat

Another solution is to return the children that violates such restrictions with a very small fitness such that it will be naturally discarded during replacement.

Terminology learned today

- **search problem:** find a valid solution from a set.
- **optimization problem:** find the best solution from a set.
- **maximization problem:** the best solution has the maximum value.
- **minimization problem:** the best solution has the minimum value.
- **search space:** set of all candidate solutions.
- **solution representation:** convenient representation of a solution.
- **objective-function:** functiona that maps a candidate to a value measuring the quality of the solution.
- **neighborhood:** set of candidate solutions close to a solution s.

Terminology learned today

- **local optima:** the best solution inside the neighborhood.
- **global optima:** the best solution of the search space.
- **local search:** searches for the nearest local optima.
- **heuristic:** technique to efficiently find a solution without any guarantees.
- **exploration:** act of exploring the search space.
- **exploitation:** act of exploring a promising neighborhood.

Further reading

- Field Guide - GP
- Livro - Koza
- Livro 2 - Koza
- Livro 3 - Koza
- Livro 4 - Koza

Further reading

- Genetic Programming The Movie Part 1
- Genetic Programming The Movie Part 2
- Genetic Programming III: Human Competitive Machine Intelligence
- Genetic Programming IV Video: Human-Competitive Machine Intelligence
- Chapter 4 of Gabriel Kronberger, Bogdan Burlacu, Michael Kommenda, Stephan M. Winkler, Michael Affenzeller. Symbolic Regression. To be plubished.

Next lecture

- Nonlinear evolutionary symbolic regression

Acknowledgments