Genetic Programming

Outline

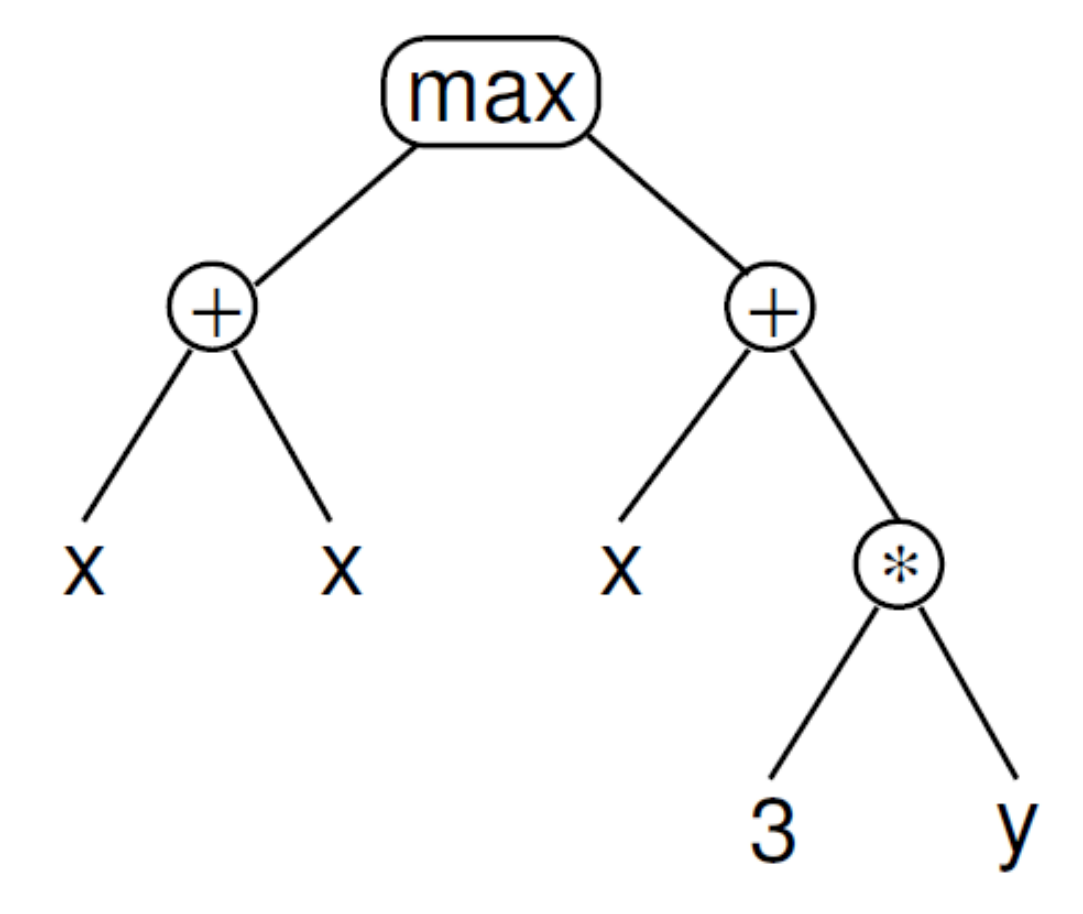
- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Representation of Software in Classical GP

- In the form of tree structures: root at the top, leaves (terminals) at the bottom, nodes represent functions with different “arities”

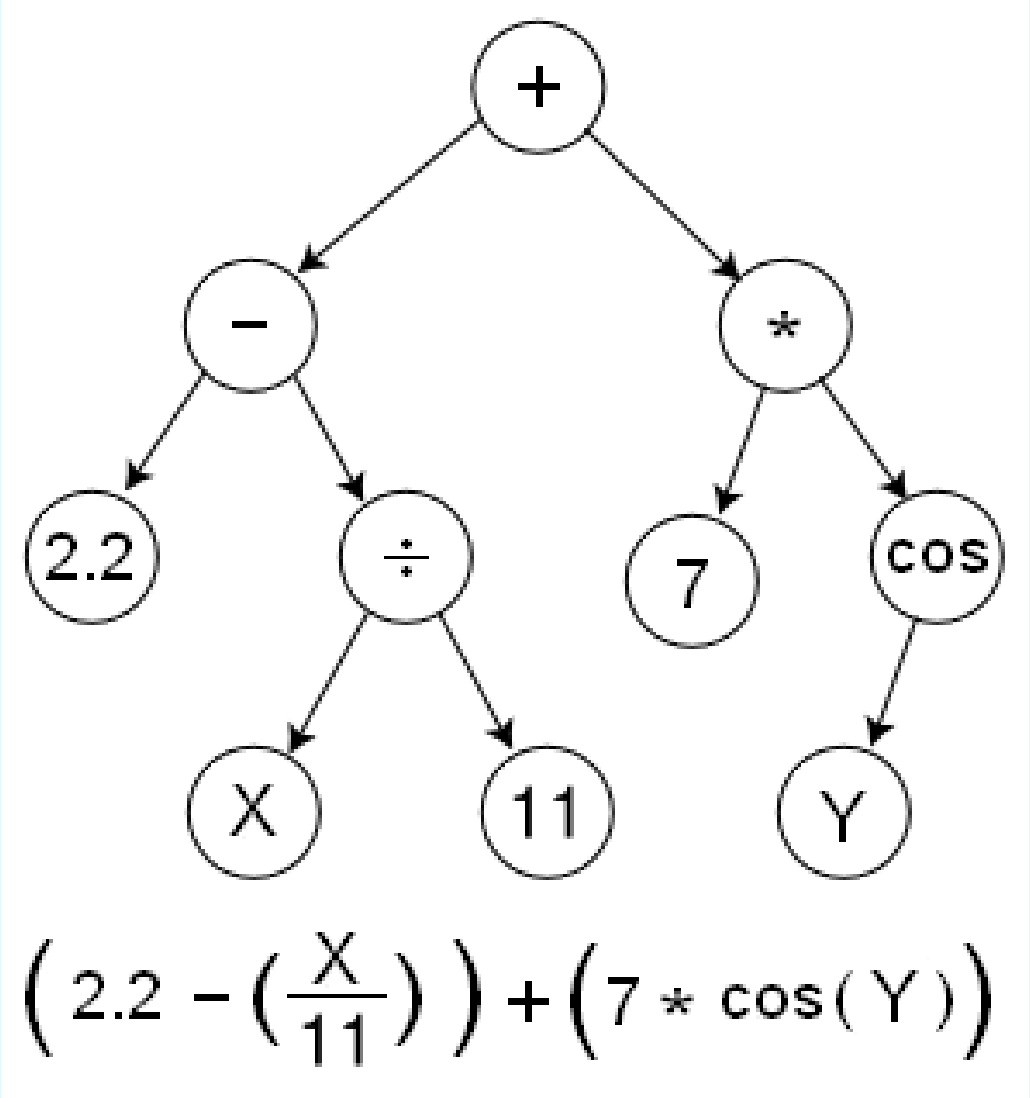


representation of max(x + x, x + 3 * y)

- This and other figs. & tables from Poli, Langdon and McPhee (2008)

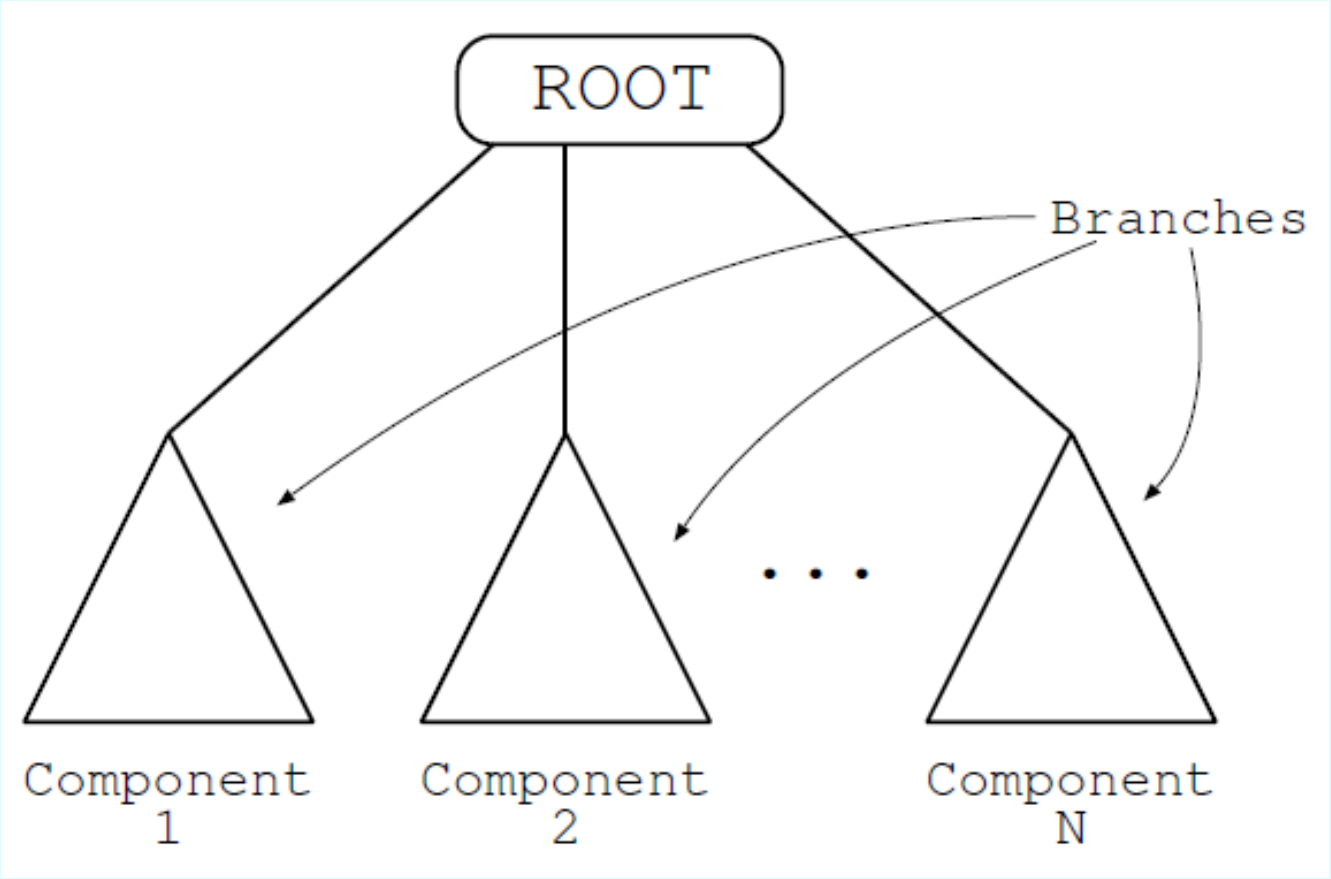
Representation of Software in Classical GP

- In the form of tree structures:



Representation of Software in Classical GP

- Can be expressed hierarchically from higher to lower levels:



- Note this is “Classical” GP, there are other forms of GP which do not necessarily use tree structures.

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

The Primitive Sets – the Function Set

- The first task in trying to create a program represented as a tree is to define the function set (nodes) and the terminal set (leaves)
- For automatic synthesis of a program, these are the ingredients that the automation will play with in different permutations and combinations, creating different structures
- A node combines sub-trees – which can be anything from a simple leaf to a sub-routine or another program itself
- Each node has an arity – that of the function it represents – could be one, two or more (max or min, an IF-THEN-ELSE construct)
- A node can be viewed as the root of a sub-tree
- As stated, a node represents a function, and the set of all functions that can be used for synthesizing the desired program needs to be provided at start.

The Primitive Sets – the Terminal Set

- The terminal set is composed of the variables of the program as well as constants that will be used
- they are the leaves of the tree (it will be clear by now that we are actually talking of an inverted tree)
- Functions that do not take any arguments, e.g. rand(), can also compose the terminal set – note these are 0-arity functions
- The values of some of the constants can also be subject to automated synthesis.

The Primitive Sets – the Terminal Set

- Examples of Function Set and Terminal Set :

Function Set	
Kind of Primitive	*Example(s)*
Arithmetic	`+, *, /`
Mathematical	`sin, cos, exp`
Boolean	`AND, OR, NOT`
Conditional	`IF-THEN-ELSE`
Looping	`FOR, REPEAT`
⋮	⋮

Terminal Set	
Kind of Primitive	*Example(s)*
Variables	`x, y`
Constant values	`3, 0.45`
0-arity functions	`rand, go_left`

“Fitness” of Programs

- In the Automatic Synthesis of programs, we are trying to create the most optimally performing program
- In effect, we are experimenting with different programs and extracting the optimal
- Any discussion on optimization has necessarily to be accompanied with a “fitness function”
- How do we create and evaluate this fitness function?
- The fitness function is created by defining a large number of training samples – input values and corresponding outputs
- A sample program takes these inputs, generates corresponding outputs, and then finds the RMS of error against given outputs
- That is the baseline fitness function, which can be incremented by other components like e.g. soft constraints, size, efficiency, etc.
- e.g. in the second program sample, slide 5, we can take a number of values of x, calculate the outputs, and provide these to the programs for calculating fitness.

“Fitness” of Programs

- Very often GP is used not for synthesizing programs per se, but for designing optimally performing systems
- In this case more formal fitness functions (or objective / cost functions) can be given which minimize or maximize certain performance metrics
- GP is used often – particularly recently – for optimizing existing programs based on various metrics, rather than creating a program from scratch.

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

The major steps

- The most important hallmark of GPs is that it is an Evolutionary Algorithm, with roots firmly in GA
- Thus, it works concurrently on a population of candidate programs
- The population of candidates has to be first *Initialized*
- The search space of solutions is the space of all possible programs, duly constricted by limiting the elements of *function set* and *terminal set*
- New programs are created out of the old across generations by
  - Crossover
  - Mutation
  - Selection – with fitness evaluation as an important component
- Termination when some pre-defined criteria is met.

Outline

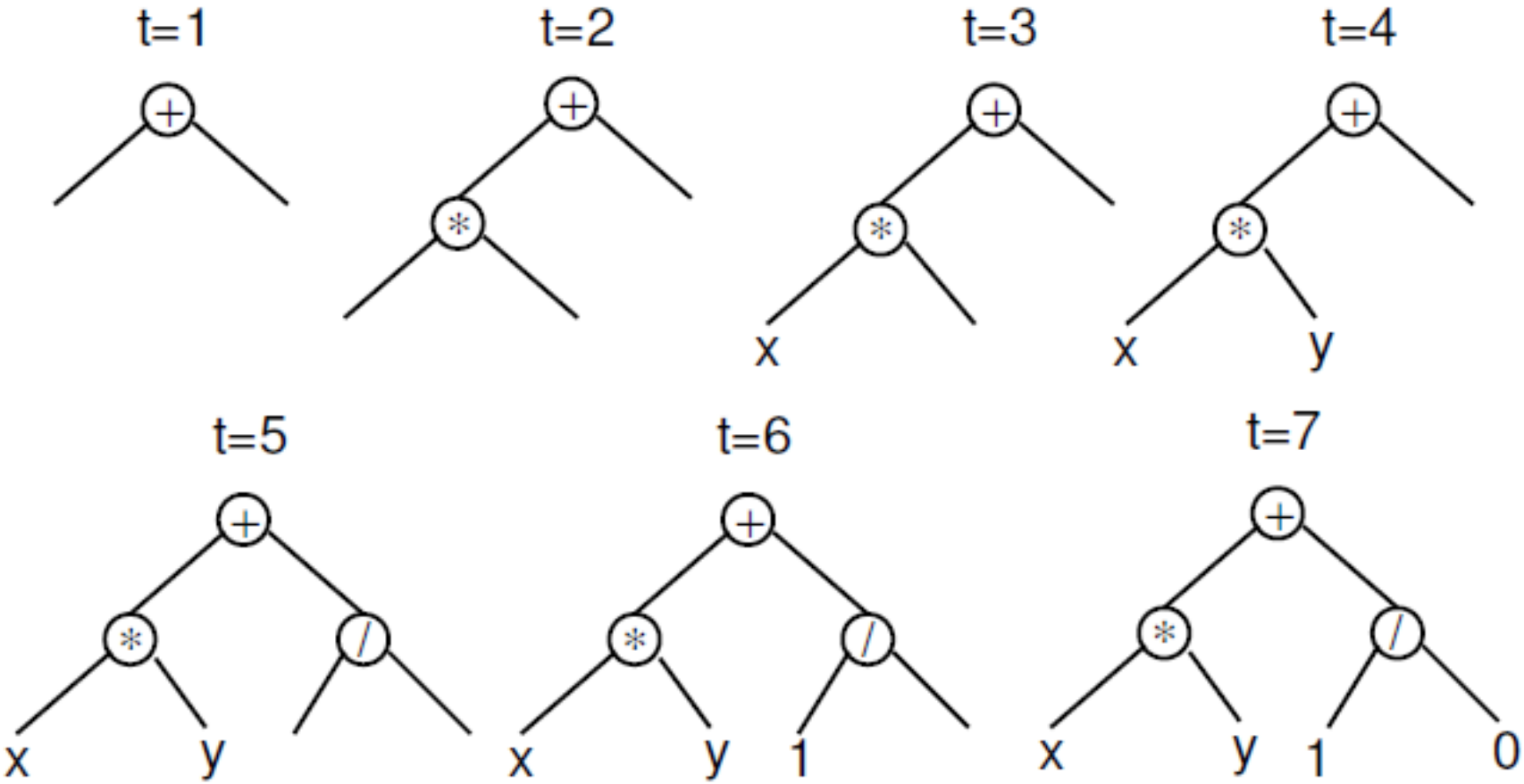
- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Initialization

- In classical GP there are two broad approaches to initialization – the full and grow methods
- In both cases, the user has to predefine the function and terminal sets, AND the *maximum depth* – defined as the number of links from the root to the deepest leaf
- Very often a part – about half – of the population is initialized using the *full* method, and the other half using *grow*
- This is called *ramped half-and-half*.

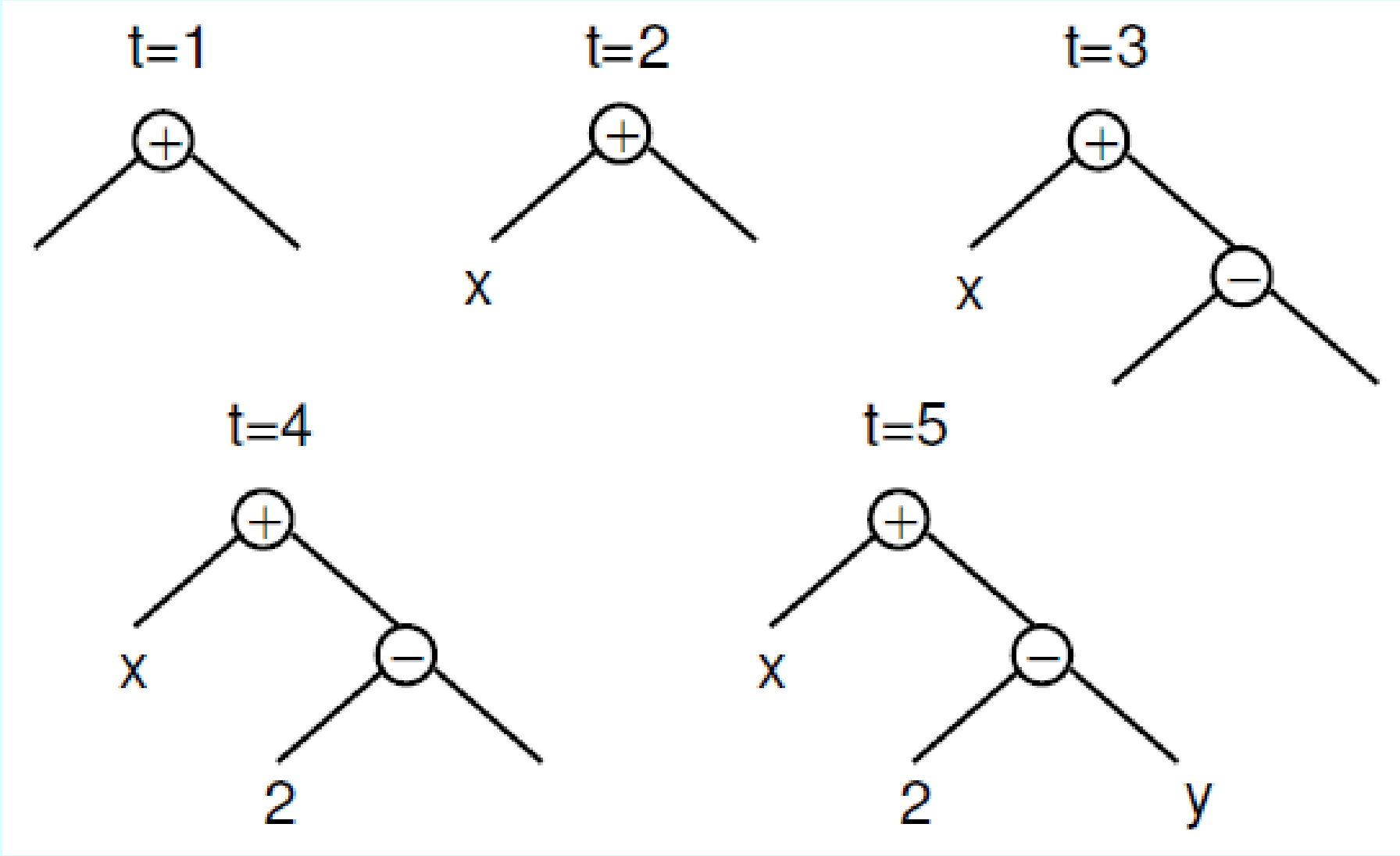
Initialization: the Full Method

- Creation to a full depth of 2 in seven steps, only arithmetic operators as function set



Initialization: the Grow Method

- Creation to a max depth of 2 in five steps, only arithmetic operators as function set



Outline

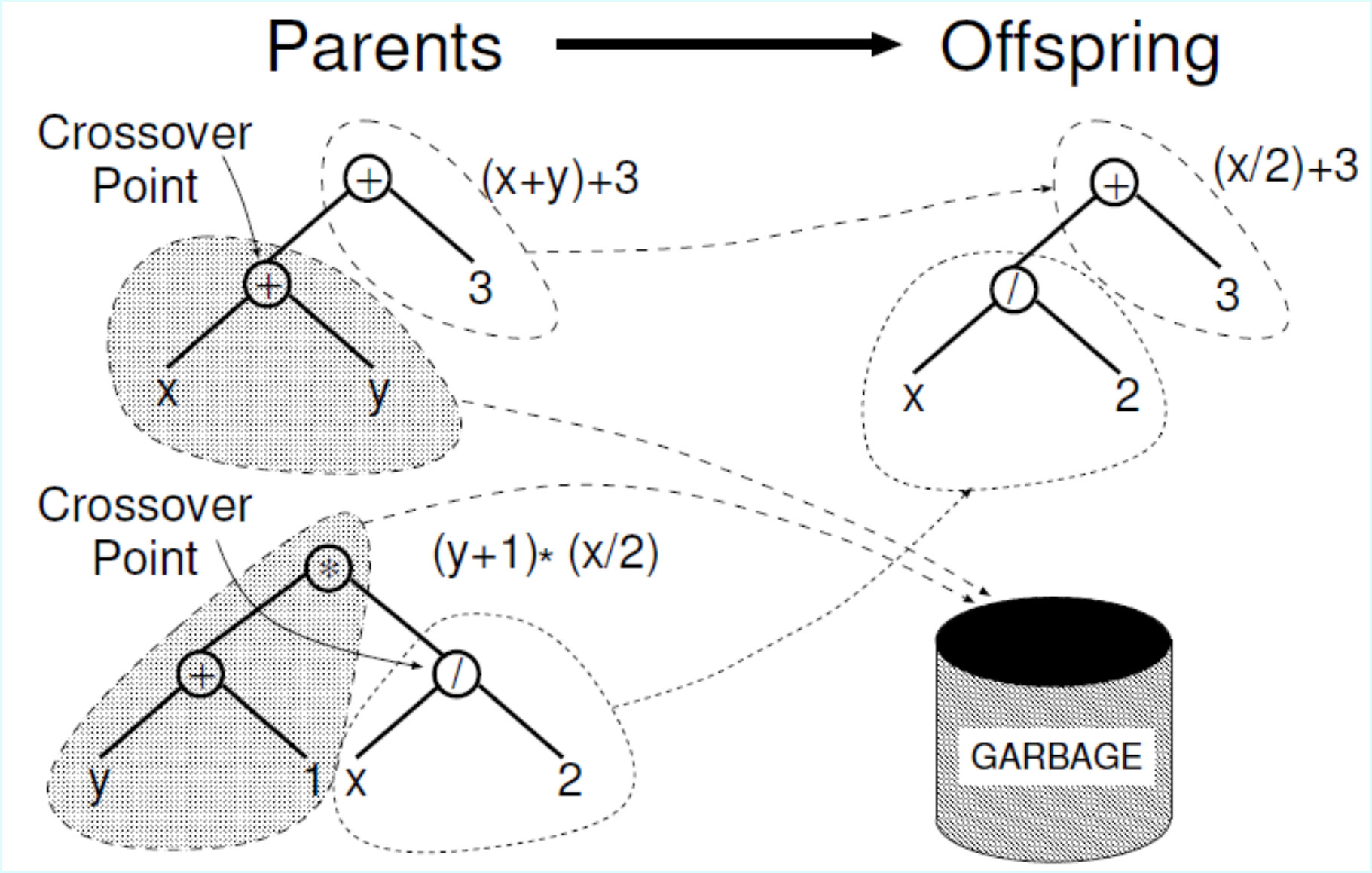
- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Crossover

- Two random population members are selected (often instead of purely random, stronger members of population are considered more frequently)
- Copies are made of both candidates and the operations are performed on the copies, leaving originals intact
- In either, a random node is identified as a crossover point
- The sub-tree attached to the crossover point of one is extracted and replaces the sub-tree attached to the crossover point of the alternate member
- In this way one new member of population is created upon crossover, leaving the original incumbents unchanged which can continue to perform crossover. Note this is variant from classical GA.
- There are many variants of crossover.

Crossover

- Figure illustrates the sub-tree crossover mechanism.



Outline

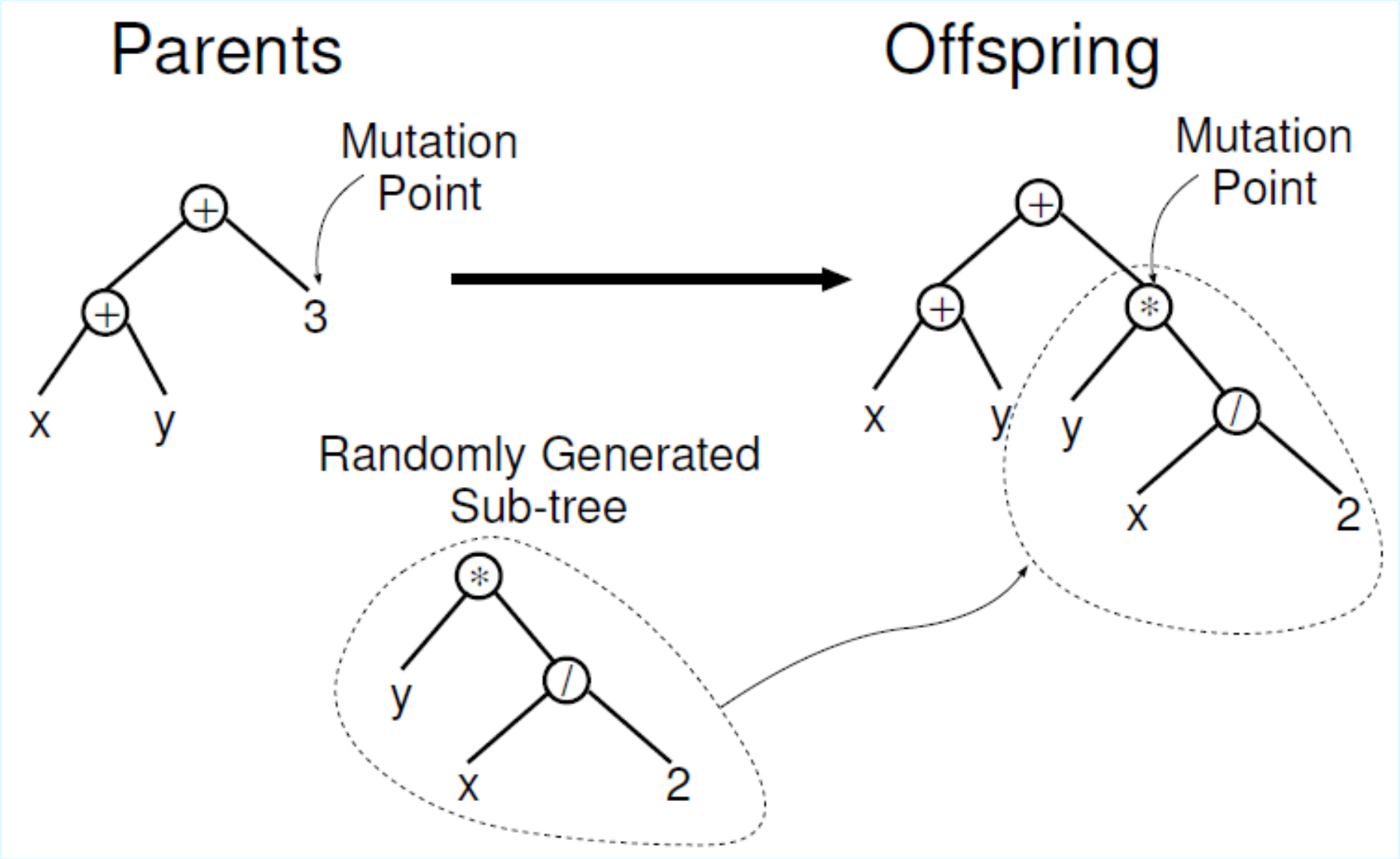
- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Mutation

- Classically two approaches are followed – *sub-tree mutation* and *point mutation*
- In the former, first a random sub-tree is generated using rules analogous to initialization
- Then a random node is identified in a tree
- The sub-tree associated with that node is replaced by the newly created random sub-tree
- Under point mutation, a random node is identified again
- The function in that node is replaced by a different, randomly selected function *of the same arity* from the function set.

Mutation

- Illustration of *sub-tree mutation*



Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Selection

- Any Evolutionary algorithm is designed to probabilistically reproduce the stronger candidates more than the weaker ones
- *Fitness Proportionate Selection* (like Roulette Wheel) makes it explicit and maps the fitness values to the probabilities of reproduction
- *Tournament Selection* is more benign and just compares fitnesses as higher or lower
- Classical GP mostly follows Tournament Selection
- For every one candidate position to be filled in the new generation, a small sample of candidates are randomly picked (in current gen) and the strongest among them is used to fill that position
- In this way all new candidates are sequentially created
- Importantly, since the parent sample is selected randomly for each new candidate, any candidate in the existing generation has equal probability of being picked for comparison; only the weakest few will never get selected into the new generation.

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

Closure and Sufficiency

- These are important practical considerations that can be considered necessary parts of basic GP understanding
- Since crossover and mutation tend to arbitrarily replace the sub-tree of a node (i.e. function) it is important that the argument type needed by the function does not change, e.g. an arithmetic operation needs numbers while a logical operation needs booleans
- this can be effected by
  - putting constraints on crossover and mutation such that only valid sub-trees are placed in desired positions (or mutation flips to valid functions for a node)
  - using type-casts (restricted) such that a function receives valid values

Closure and Sufficiency

- Further, it is necessary to ensure that the Terminal Set has all ingredients needed to generate the solution for a particular problem
- It is easy to choose a very large terminal set, but that will blow up the search space needing much more computation to arrive at a solution that, in any case, does not need the bulk of the functions and values
- but the restrictions imposed on the terminal set should be sufficient to generate the optimal solution, else only sub-optimal solutions will be created.

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

The Parameter Set

- By this time it would be clear that a good number of meta-parameters are needed for a GP solution. These include
  - The population size
  - The maximum number of generations allowed
  - The crossover and mutation probabilities
  - The maximum allowed depth of tree (to prevent *bloat*)
  - The maximum number of nodes (size of tree) (to prevent *bloat*)
- Also the various other parameters – like the terminal set, the objective function, the fitness evaluation data and function/s, the initialization approach, termination criterion, etc, are needed.
- These are shown as a compact table as an example in the next slide.

The Parameter Set

Objective:	Find program whose output matches $x^2 + x + 1$ over the range $-1 \leq x \leq +1$.
Function set:	$+$, $-$, % (protected division), and $\times$; all operating on floats
Terminal set:	x, and constants chosen randomly between -5 and $+5$
Fitness:	sum of absolute errors for $x \in \{-1.0, -0.9, \ldots 0.9, 1.0\}$
Selection:	fitness proportionate (roulette wheel) non elitist
Initial pop:	ramped half-and-half (depth 1 to 2. 50% of terminals are constants)
Parameters:	population size 4, 50% subtree crossover, 25% reproduction, 25% subtree mutation, no tree size limits
Termination:	Individual with fitness better than 0.1 found

Outline

- Representation of Software
- The Primitive Sets – the Function Set and the Terminal Set
- The ‘fitness’ of programs
- The major steps
- Initialization
- Crossover
- Mutation
- Selection
- Closure and Sufficiency
- The Parameter Set
- The current impetus

The current impetus

- GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done
- It has generated a plethora of human-competitive results and applications, including novel scientific discoveries and patentable inventions
- One can see various applications in the “Handbook of Genetic Programming Applications” at http://www.springer.com/in/book/9783319208824
- One can freely download the book ISBN 978-1-4092-0073-4 (2008) at:

http://www0.cs.ucl.ac.uk/staff/w.langdon/ftp/papers/poli08_fieldguide.pdf

The current impetus

- To be honest, its main theme of automatically generating S/W from scratch has not had the desired degree of success
- However, it has had good success in automatically improving existing programs, also called *Genetic Improvement*
- In this driving age of automation of S/W, GP is expected to play a more and yet more powerful role
- Most importantly, it is major step in the direction of the ultimate aims of AI – creation of software that can automatically breed new software to serve any existing problems / ambitions at hand.

THANK YOU