Evolutionary Computation and Learning

Genetic Programming 3: Advanced Topics

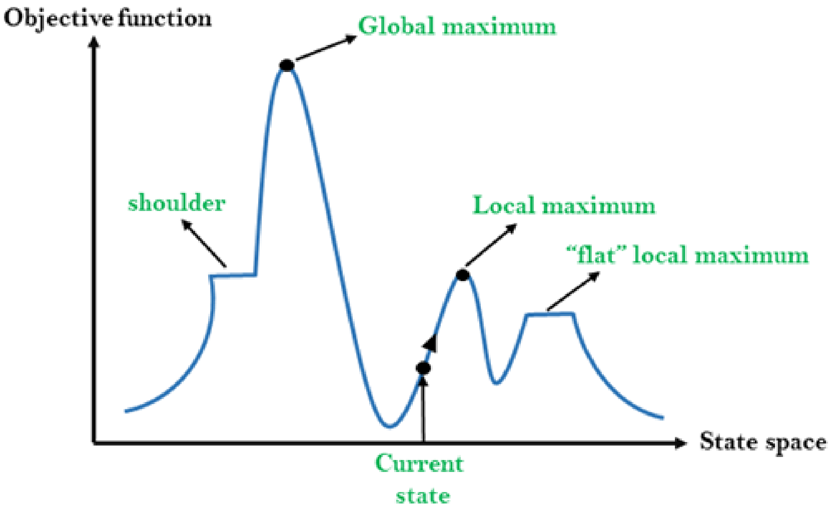
By: Dr. Vahid Ghasemi

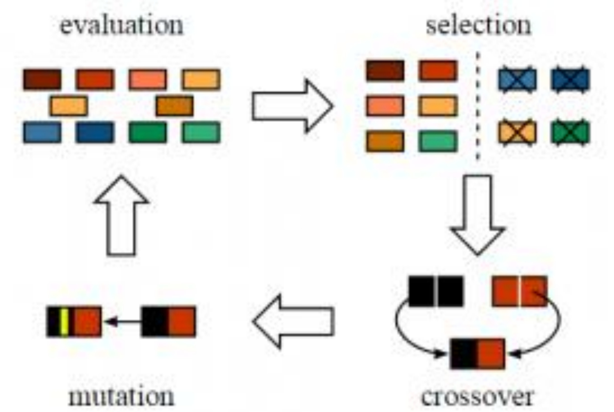
Outline

- Gradient Descent in GP
- Strongly Typed GP
- Grammar-based GP

Gradient Descent in GP

- Gradient descent search/hill climbing search is widely used in many techniques, including neural networks
- Gradient descent search has two "problems"
  - Only has one potential solution
  - It often is stuck in local optima
- Genetic algorithms/programming can tackle the local optima issue, but it does not exploit promising regions sufficiently (too random)
- Can we combine them together?



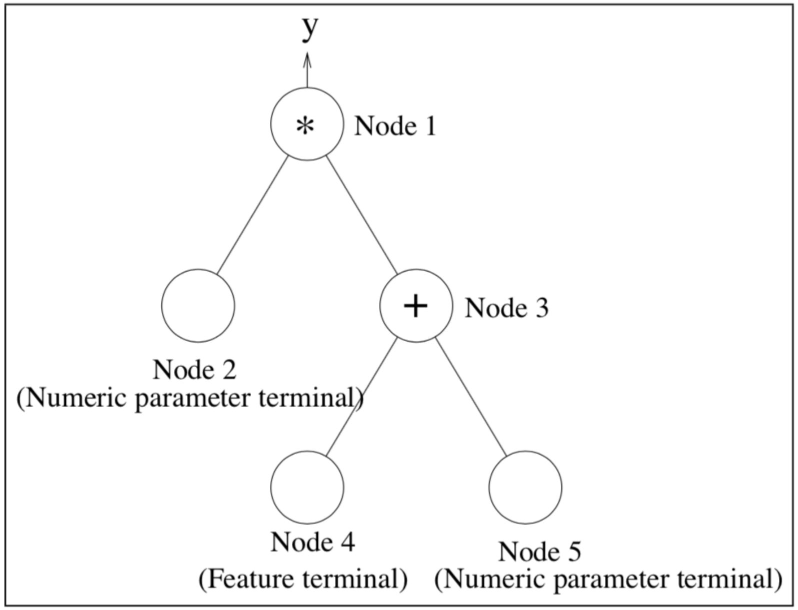


Gradient Descent in GP

- Traditional GP: randomly mutate the random constants
- GPGD: Apply gradient descent locally on the numeric terminals (random constants)

$y = C_1 * (X + C_2)$

Function f	meanings	$\frac{\partial f}{\partial a_1}$	$\frac{\partial f}{\partial a_2}$	$\frac{\partial f}{\partial a_3}$
$(+\ a_1\ a_2)$	$a_1 + a_2$	1	1	n/a
$(-\ a_1\ a_2)$	$a_1 - a_2$	1	-1	n/a
$(*\ a_1\ a_2)$	$a_1 \times a_2$	a_2	a_1	n/a
$(/\ a_1\ a_2)$	$a_1 \div a_2$	a_2^{-1}	$-a_1 \times a_2^{-2}$	n/a
(if $a_1\ a_2\ a_3$)	if $a_1 < 0$ then a_2 else a_3	0 0	1 if $a_1 < 0$ 0 if $a_1 \geq 0$	0 if $a_1 < 0$ 1 if $a_1 \geq 0$



$$\frac{\partial y}{\partial O_2} = \frac{\partial (O_2 * O_3)}{\partial O_2} = O_3$$

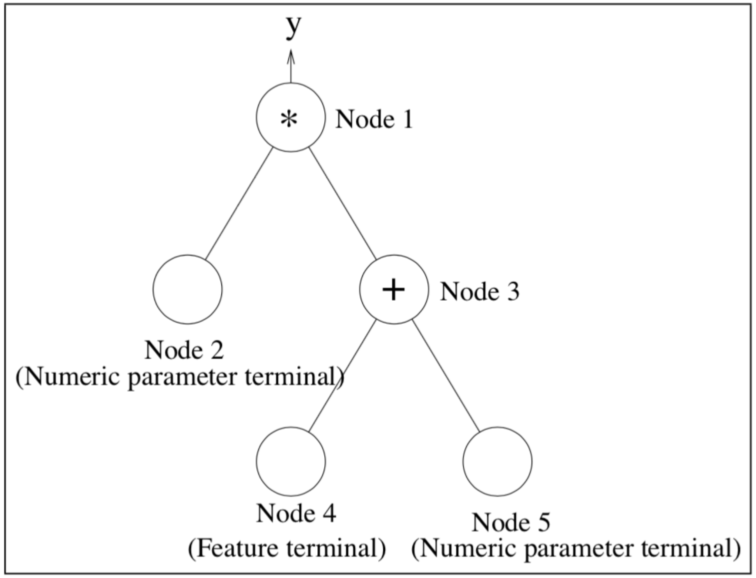
Partial derivative reflects how a decision variable affect the output.

$$\frac{\partial y}{\partial O_5} = \frac{\partial (O_2 * O_4 + O_2 * O_5)}{\partial O_5} = O_2$$

Gradient Descent in GP

- Cost function:
  - **Regression**: $C = MSE = \frac{1}{n}\sum_{i=1}^{n}(t_i - y)^2$, $\frac{\partial C}{\partial y} = -\frac{2}{n}\sum_{i=1}^{n}(t_i - y)$
  - Different problems may have different cost functions
- $\frac{\partial C}{\partial O_j} = \frac{\partial C}{\partial y} * \frac{\partial y}{\partial O_j} = -\frac{2}{n}\sum_{i=1}^{n}(t_i - y) * \frac{\partial y}{\partial O_j}$
- $O_j \leftarrow O_j - \eta * \frac{\partial C}{\partial O_j}$, η is the learning rate

$$O_2 = O_2 + \eta * \frac{2}{n}\sum_{i=1}^{n}(t_i - y_i) * O_{3,i}$$



$$\frac{\partial y}{\partial O_2} = \frac{\partial(O_2 * O_3)}{\partial O_2} = O_3$$

$$\frac{\partial y}{\partial O_5} = \frac{\partial(O_2 * O_4 + O_2 * O_5)}{\partial O_5} = O_2$$

Improve GP tree performance with gradient descent.

$$O_5 = O_5 + \eta * \frac{2}{n}\sum_{i=1}^{n}(t_i - y_i) * O_{2,i}$$

Gradient Descent in GP

- **GP + Gradient Descent**
- Initialise the GP population;
- **Repeat** until stopping criteria is met:
- Evaluate individuals;
- Parent selection;
- Crossover/Mutation/Reproduction;
- Gradient descent local search
  - When/Which individual to do gradient descent?
  - Each generation / Every 5 generations?
  - Every individual / The top-performing individuals

- Gradient descent is slow, so cannot do many times

Strongly Typed GP

- Each primitive node in the GP tree has a type
- Just like human-written Java program

```
public double IF(boolean a, double b, double c) {
   if (a == True) return b;
   return c;
}
```

- Eliminates the closure constraint by requiring each function to specify precisely the **data type** of its arguments and its output values
- The data type of the output of a node **MUST** match the data type of the corresponding argument of its parent node
- The data type of the output of the root node **MUST** be the data type of the solution

Strongly Typed GP

- **Example: matrix/vector operations**
- Matrix addition/subtraction
  - **Require**: input matrices have the same #rows and #cols
  - **Return**: output matrix with the same #rows and #cols as the inputs
- Matrix multiplication M1*M2
  - **Require**: #cols(M1) = #rows(M2)
  - **Return**: output matrix with #rows(M1) and #cols(M2)

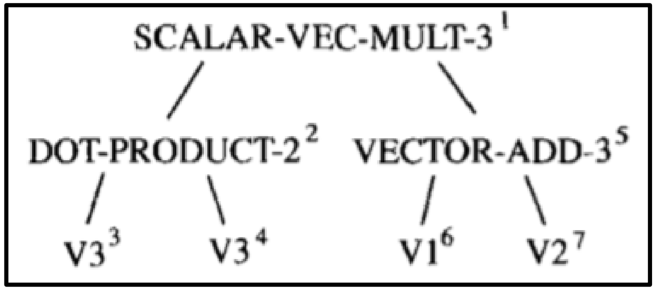
$$\begin{bmatrix}1\\2\\3\end{bmatrix} \cdot \begin{bmatrix}1\\2\\3\end{bmatrix} = 1 + 4 + 9 = 14$$

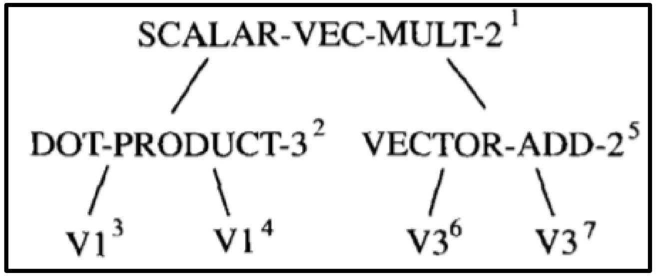
$$\begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}1\\2\\3\end{bmatrix} = \begin{bmatrix}2\\4\\6\end{bmatrix}$$

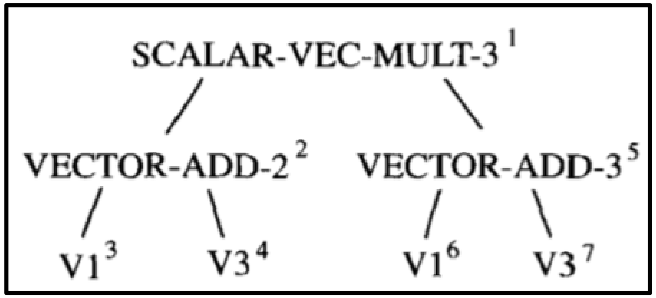
Function Name	Arguments	Return Type
DOT-PRODUCT-3	VECTOR-3 VECTOR-3	FLOAT
VECTOR-ADD-2	VECTOR-2 VECTOR-2	VECTOR-2
MAT-VEC-MULT-4-3	MATRIX-4-3 VECTOR-3	VECTOR-4
CAR-FLOAT	LIST-OF-FLOAT	FLOAT
LENGTH-VECTOR-4	LIST-OF-VECTOR-4	INTEGER
IF-THEN-ELSE-INT	BOOLEAN INTEGER INTEGER	INTEGER

Strongly Typed GP

- For returning VECTOR-3,
  - **Terminals** = {V1, V2, V3}
    - V1 and V2 are VECTOR-3
    - V3 is VECTOR-2
- **Functions**
  - DOT-PRODUCT-2
    - (VECTOR-2, VECTOR-2) -> FLOAT
  - DOT-PRODUCT-3
    - (VECTOR-3, VECTOR-3) -> FLOAT
  - VECTOR-ADD-2
    - (VECTOR-2, VECTOR-2) -> VECTOR-2
  - VECTOR-ADD-3
    - (VECTOR-3, VECTOR-3) -> VECTOR-3
  - SCALAR-VEC-MULT-2
    - (FLOAT, VECTOR-2) -> VECTOR-2
  - SCALAR-VEC-MULT-3
    - (FLOAT, VECTOR-3) -> VECTOR-3
- Which one(s) are legal?

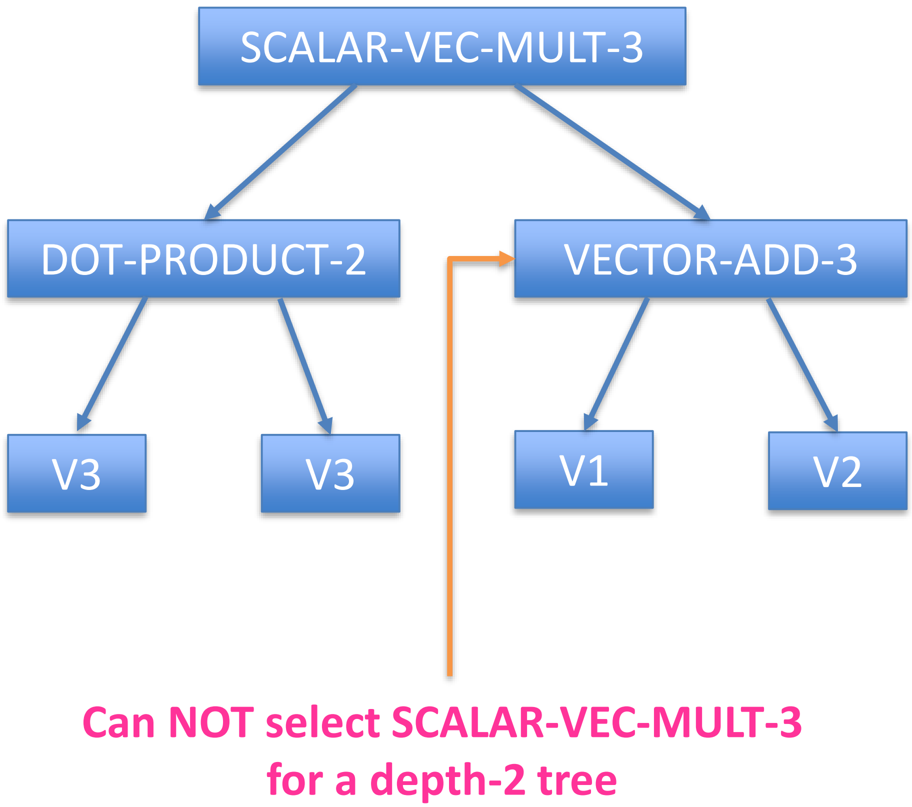






Strongly Typed GP

- Using a depth-3 **FULL** method to generate a strongly typed tree that returns VECTOR-3
- **Terminals** = {V1, V2, V3}
  - V1 and V2 are VECTOR-3
  - V3 is VECTOR-2
- **Functions**
  - DOT-PRODUCT-2
    - (VECTOR-2, VECTOR-2) -> FLOAT
  - DOT-PRODUCT-3
    - (VECTOR-3, VECTOR-3) -> FLOAT
  - VECTOR-ADD-2
    - (VECTOR-2, VECTOR-2) -> VECTOR-2
  - VECTOR-ADD-3
    - (VECTOR-3, VECTOR-3) -> VECTOR-3
  - SCALAR-VEC-MULT-2
    - (FLOAT, VECTOR-2) -> VECTOR-2
  - SCALAR-VEC-MULT-3
    - (FLOAT, VECTOR-3) -> VECTOR-3



Strongly Typed GP

- Crossover and mutation are **restricted**
- Crossover:
  - In the first parent, randomly pick a subtree (the same as standard crossover)
  - In the second parent, identify the subtrees (nodes) with the same output type as the picked subtree
  - Randomly select from the subtrees with the same output type
  - Swap the two subtrees
  - What if no subtree with the same output type?
- Mutation:
  - In the parent, randomly select a subtree
  - Generate a new subtree with the same output type as the selected subtree
  - Replace
  - How to generate the new subtree with the given output type?

Strongly Typed GP

- We don't want to have "VECTOR-ADD-2", "VECTOR-ADD-3", …, but just one generic function "VECTOR-ADD"
- This can be achieved by generic type and generic function
- A generic function is a function which can take a variety of different argument types and return values of a variety of different types

Function Name	Arguments	Return Type
DOT-PRODUCT	VECTOR-i VECTOR-i	FLOAT
VECTOR-ADD	VECTOR-i VECTOR-i	VECTOR-i
MAT-VEC-MULT	MATRIX-i-j VECTOR-j	VECTOR-i
CAR	LIST-OF-t	t
LENGTH	LIST-OF-t	INTEGER
IF-THEN-ELSE	BOOLEAN t t	t

STGP for Evolving Scheduling Rules

- GPHH **terminals** for Job Shop Scheduling
  - TIME-type: processing time, current time, due date
  - COUNT-type: number of operations remaining, number of jobs in the queue
  - WEIGHT-type: the weight of a job (importance)
  - …

Notation	Description	Dimension
WIQ	Work In Queue	TIME
MWT	Machine Waiting Time	TIME
PT	Processing Time	TIME
NPT	Next Processing Time	TIME
OWT	Operation Waiting Time	TIME
NWT	Next Machine Waiting Time	TIME
WKR	Work Remaining	TIME
WINQ	Work In Next Queue.	TIME
rFDD	Relative FDD	TIME
rDD	Relative DD	TIME
TIS	Time In System	TIME
SL	Slack	TIME
NIQ	Number of operations In Queue	COUNT
NOR	Number of Operations Remaining	COUNT
NINQ	Number of operations In Next Queue	COUNT
W	Weight	WEIGHT

STGP for Evolving Scheduling Rules

- It makes no sense to add a time and a count
- Define typed functions
  - **X**: job weight
  - **C**: count
  - **D**: time duration
  - **T**: absolute time (e.g., clock time)
- Terminals belong to the four types
- Can be more interpretable

*	X	C	D	T
X	X	X	D	
C	X	C	D	
D	D	D		
T				

%	X	C	D	T
X	X	X		
C	X	X		
D	D	D	X	X
T			X	X

+	X	C	D	T
X	X			
C				
D			D	T
T			T	

−	X	C	D	T
X	X			
C				
D			D	T
T			T	D

min	X	C	D	T
X	X			
C		C		
D			D	
T				T

max	X	C	D	T
X	X			
C		C		
D			D	
T				T

Properties of STGP

- STGP works by cutting down the search space:
  - Program generation process is restricted.
  - **Closure in un-typed GP**: any function is well-defined for all possible values that could be returned by other functions or terminals.
  - **Closure in STGP** is restricted to a particular data type: any function that needs to take an argument should consider the return type of the argument.
  - This immediately cuts down the number of branches by constraining the tree construction process.

Properties of STGP

- Each primitive can do a lot more "meaningful" work in the space of a single node.
  - Such work usually needs a large subtree to implement in an un-typed GP system.
  - Solutions obtained in this way can be often much smaller than in un-typed GP.

- Program evolved by STGP is generally easier to interpret.
  - In general GP, it usually needs to get the value of the fitness or program output, the programs evolved are some kinds of transformation, not an algorithm.
  - STGP can produce "meaningful" results.
  - In STGP, the program domain can match the problem domain very closely.

- STGP can evolve complex data structures.

Problems of STGP

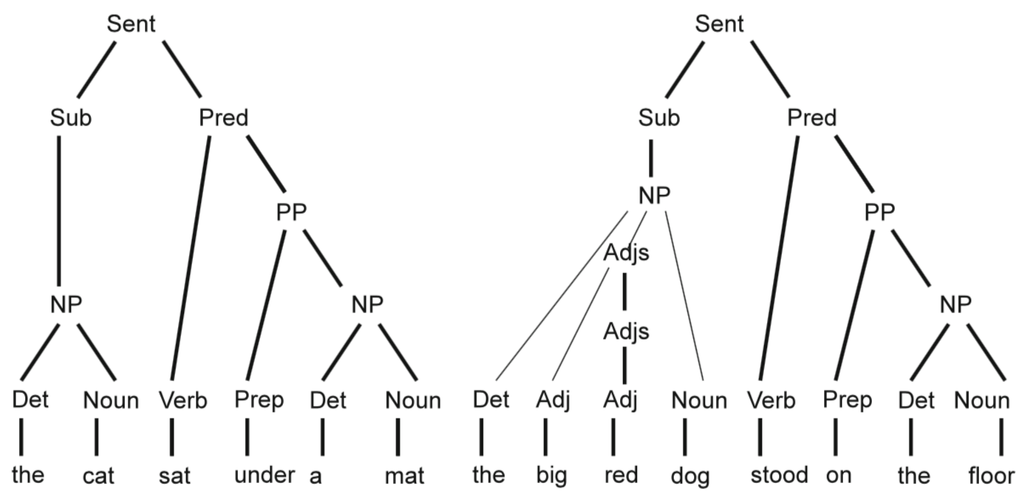
- Terminals and functions have to be carefully designed to be consistently matched.
- It is very difficult to define good fitness (evaluation) functions, even for relatively simple problems (how to consider the MANY type-inconsistent programs).
- A STGP system is usually a domain dependent system/method.
- The performance of such a system will be even worse than standard GP systems if primitive set and fitness were not properly defined.

Grammar-based GP

- Employ the language grammar to restrict the combinations

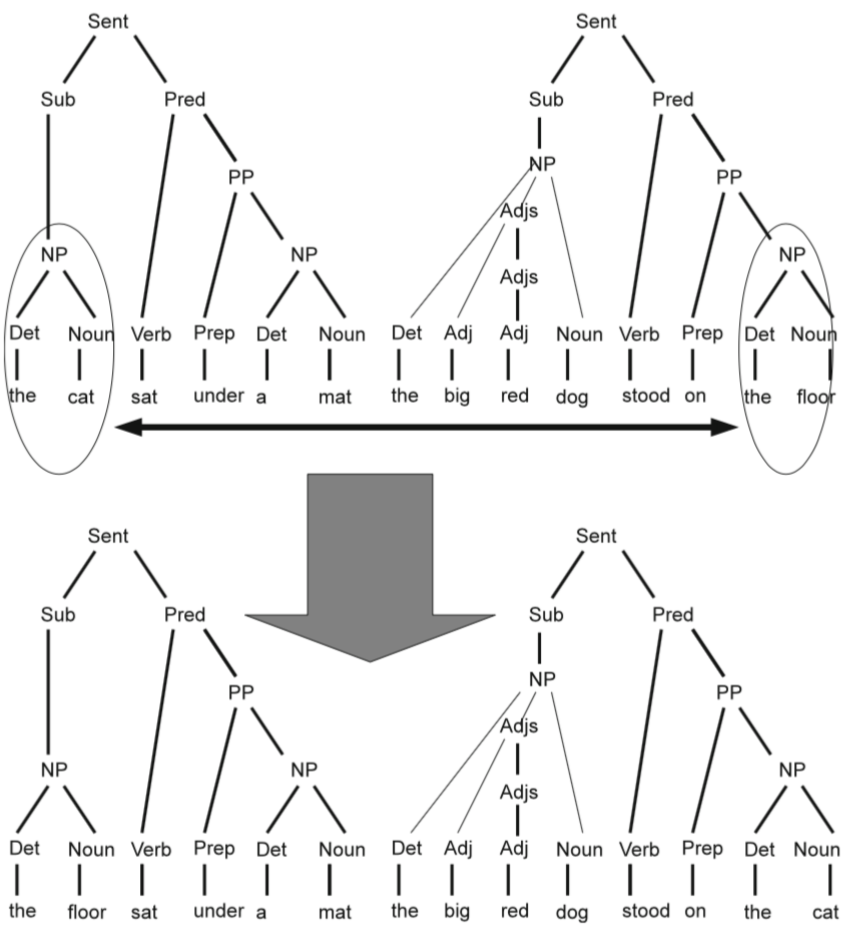
Table 1 English grammar fragment

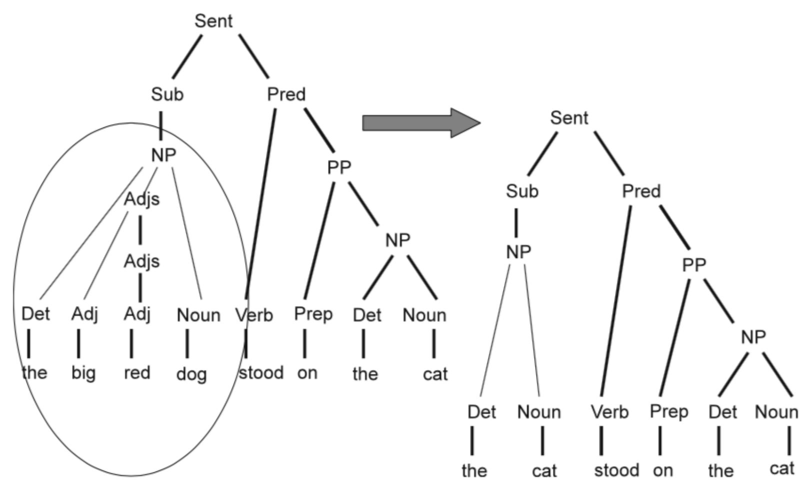
Sent → Sub Pred	PP → Prep NP	Prep → ''on''|''under''
Sub → NP	Adjs → Adj Adjs	Noun → ''cat''|''dog''
Pred → Verb PP	Adjs → Adj	|''floor'' |''mat''
NP → Det Noun	Verb → ''sat''|''stood''	Adj → ''big''|''small''
NP → Det Adjs Noun	Det → ''a''|''the''	|''red'' |''black''



Grammar-based GP

- **Program representation**: derivation tree generated by a grammar G
- **Genetic operators**
  - Crossover: require the two crossover points to have the same grammar level
  - Mutation: generate a new subtree at the selected grammar level





Grammar-based GP

- Need to design grammar carefully
- Example for evolving scheduling rules
  - Implement the STGP through grammar
- STGP is a special type of grammar-based GP

$$
\begin{array}{rlll}
S = & \langle A \rangle & & \\
N = & \{A, X, C, T, D\} & & \\
\Sigma = & \{*, \%, -, +, \mathrm{PR}, \mathrm{RT}, \mathrm{RO}, \mathrm{RJ}, \mathrm{DD}, \mathrm{W}, \mathrm{RM}, \mathrm{NQ}, & & \\
 & \mathrm{QW}, \mathrm{CT}, \mathrm{NPR}, \mathrm{NNQ}, \mathrm{AQW}, \mathrm{NQW}\} & & \\
P = & \{\langle A \rangle & ::= & \langle X \rangle \mid \langle C \rangle \mid \langle D \rangle \mid \langle T \rangle \\
 & \langle X \rangle & ::= & (\mathrm{W}) \mid (+ \langle X \rangle \langle X \rangle) \mid (- \langle X \rangle \langle X \rangle) \\
 & & & \mid (* \langle X \rangle \langle X \rangle) \mid (\% \langle X \rangle \langle X \rangle) \\
 & & & \mid (\% \langle C \rangle \langle C \rangle) \mid (\% \langle C \rangle \langle X \rangle) \\
 & & & \mid (\% \langle X \rangle \langle C \rangle) \mid (* \langle C \rangle \langle X \rangle) \\
 & & & \mid (* < X > \langle C \rangle) \mid (\% \langle D \rangle \langle D \rangle) \\
 & & & \mid (\% \langle T \rangle \langle D \rangle) \mid (\% \langle T \rangle \langle T \rangle) \\
 & & & \mid (\% \langle D \rangle \langle T \rangle) \\
 & \langle C \rangle & ::= & (\mathrm{RO}) \mid (\mathrm{NQ}) \mid (\mathrm{NNQ}) \mid (* \langle C \rangle \langle C \rangle) \\
 & \langle D \rangle & ::= & (\mathrm{PR}) \mid (\mathrm{RT}) \mid (\mathrm{QW}) \mid (\mathrm{NPR}) \mid (\mathrm{NQW}) \mid (\mathrm{AQW}) \\
 & & & \mid (+ \langle D \rangle \langle D \rangle) \mid (- \langle D \rangle \langle D \rangle) \\
 & & & \mid (- \langle T \rangle \langle T \rangle) \\
 & & & \mid (* \langle D \rangle \langle X \rangle) \mid (* \langle X \rangle \langle D \rangle) \\
 & & & \mid (* \langle C \rangle \langle D \rangle) \mid (* \langle D \rangle \langle C \rangle) \\
 & & & \mid (\% \langle D \rangle \langle C \rangle) \mid (\% \langle D \rangle \langle X \rangle) \\
 & \langle T \rangle & ::= & (\mathrm{DD}) \mid (\mathrm{CT}) \mid (\mathrm{RJ}) \mid (\mathrm{RM}) \\
 & & & \mid (+ \langle D \rangle \langle T \rangle) \mid (+ \langle T \rangle \langle D \rangle) \\
 & & & \mid (- \langle T \rangle \langle D \rangle) \mid (- \langle D \rangle \langle T \rangle)\,\}
\end{array}
$$

Problems of Grammar-based GP

- Grammar can be very hard to design
  - Requires a lot of domain knowledge
- Fitness can be very hard to design (can we violate the grammar? How much can be allowed?)
- Performance can be even worse than standard GP if the grammar and fitness are not designed properly

Summary

- Gradient descent in GP can help evolving the random constant (coefficient), but can be time consuming
  - Important to manage when and how to do the gradient descent
- Strongly typed GP defines type of terminals and functions, and closer to human-written programs
  - Easy to interpret, makes more sense
  - If properly designed, can perform better
  - If not properly designed, can perform even worse
- Grammar-based GP is based on grammar to restrict the combinations
  - STGP is a special case
  - Key issue to design grammar properly

- **Next lecture**:
  GP for Combinatorial Optimisation (academia and industry)