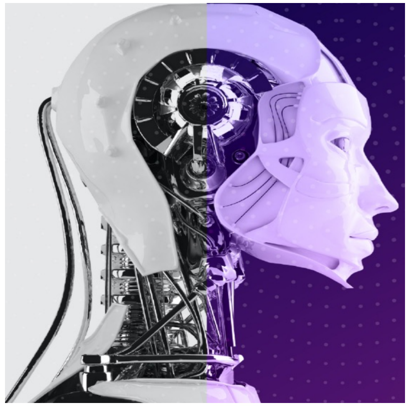


Hyper-heuristics II

Lecture 8

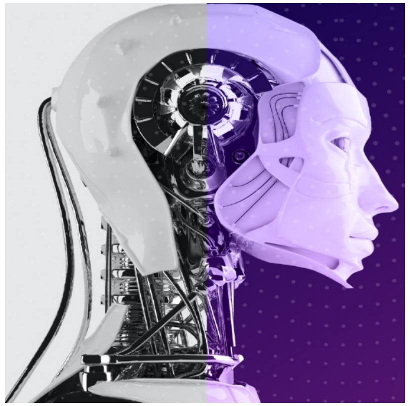
Ender Özcan

Computational Optimisation & Learning Lab

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Configuring/Tuning of Hyper/Metaheuristics for Cross-domain Search



Computational Optimisation & Learning Lab

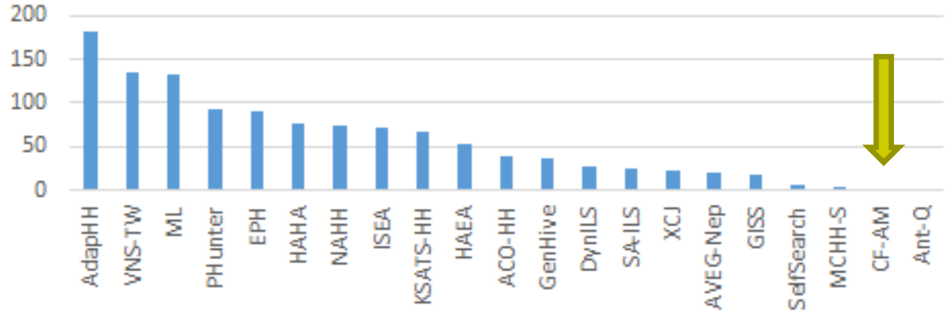
The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

A (Reconfigured) Modified Choice Function Hyper-heuristic

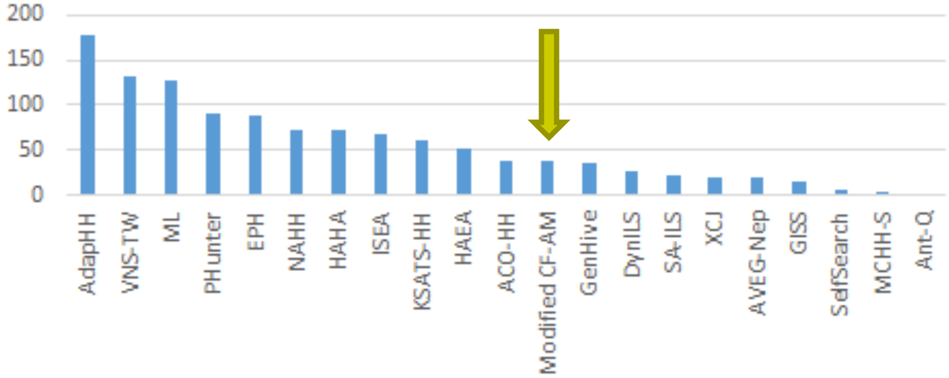
J. H. Drake, E. Özcan and E. K. Burke, A Modified Choice Function Hyper-heuristic Controlling Unary and Binary Operators, Proc. of the IEEE Congress on Evolutionary Computation (CEC), pp. 3389-3396. [PDF]

<code>



$$F(h_j) = \alpha f_1(h_j) + \beta f_2(h_k, h_j) + \delta f_3(h_j)$$

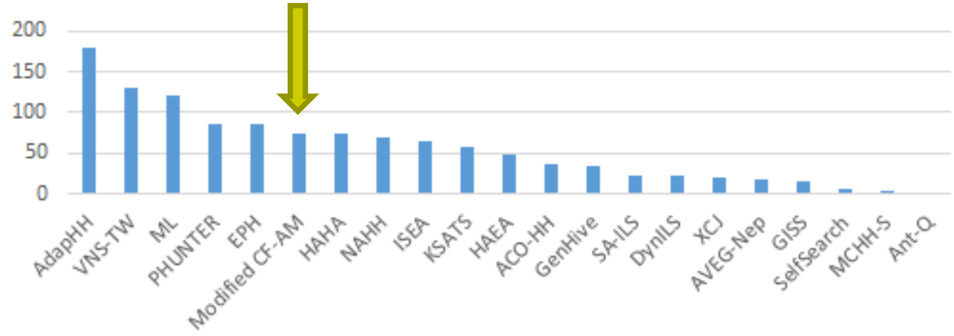
without crossover



$$F_t(h_j) = \phi_t f_1(h_j) + \phi_t f_2(h_k, h_j) + \delta_t f_3(h_j)$$

$$\delta_t \;= 1 - \phi_t$$

without crossover

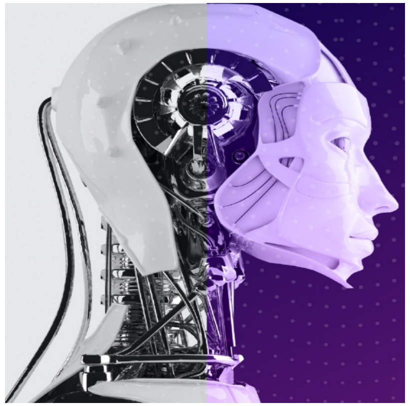


$$F_t(h_j) = \phi_t f_1(h_j) + \phi_t f_2(h_k, h_j) + \delta_t f_3(h_j)$$

$$\delta_t \;= 1 - \phi_t$$

with crossover

A Graph-based Hyper-heuristic



E. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu. A graph-based hyper-heuristic for educational timetabling problems. European Journal of Operational Research, 176(1):177-192

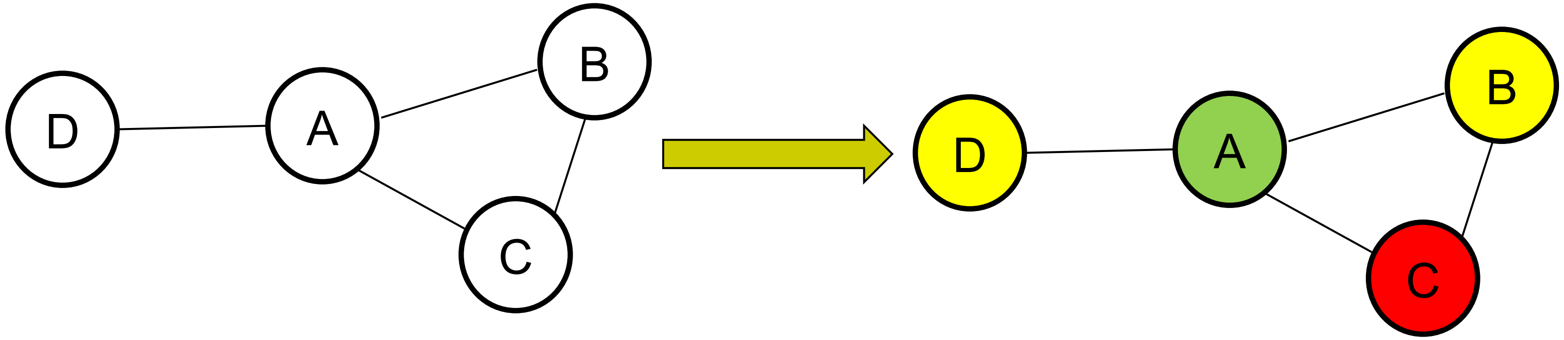
Computational Optimisation & Learning Lab

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

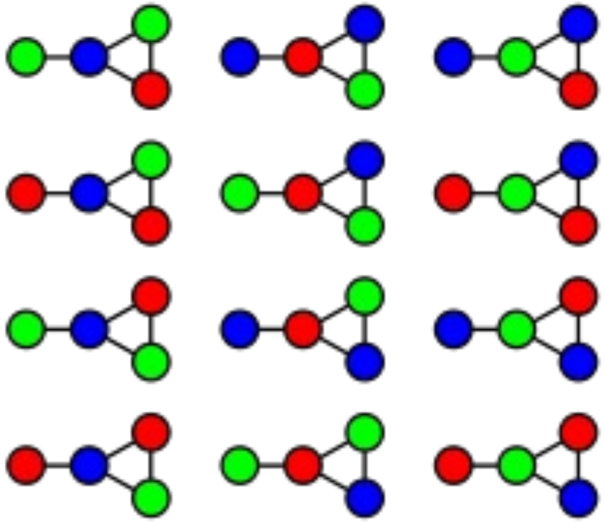
Graph Colouring

- An assignment of labels traditionally called "colours" to elements of a **graph** subject to certain constraints.
- A way of colouring the vertices of a graph such that no two adjacent vertices share the same colour; this is called a **vertex colouring**.



Graph Colouring

- ***k*-colouring problem**: Can the vertices of a graph be coloured using *k* colours so that no two vertices connected by an edge have the same colour?

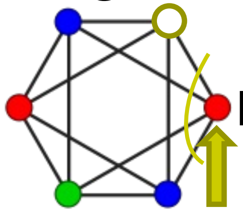


This graph can be 3-colored in 12 different ways.

- **Minimum colouring problem** is an NP-hard problem: colour the vertices of a graph using optimal (minimum) number of colours, so that no two vertices connected by an edge have the same colour.

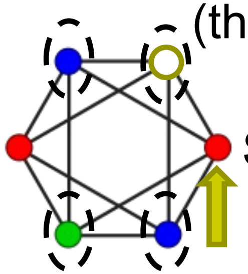
Degree and Saturation Degree of a Vertex

- Degree of a vertex: number of edges connected to that vertex.

Example:  Degree: 4

- Saturation degree of a vertex: number of differently coloured vertices already connected to it.

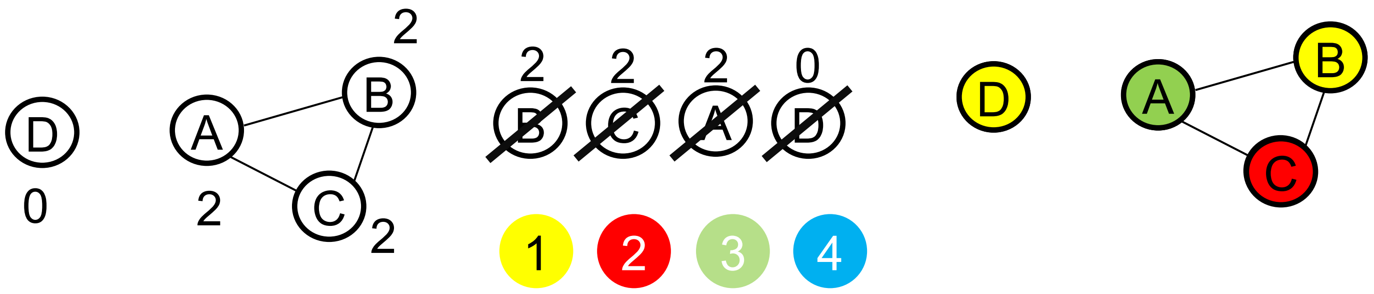
(this is uncoloured)

Example:  Saturation degree: 2 (blue and green)

- Can we use saturation degree in a heuristic to construct a solution to a graph colouring problem?

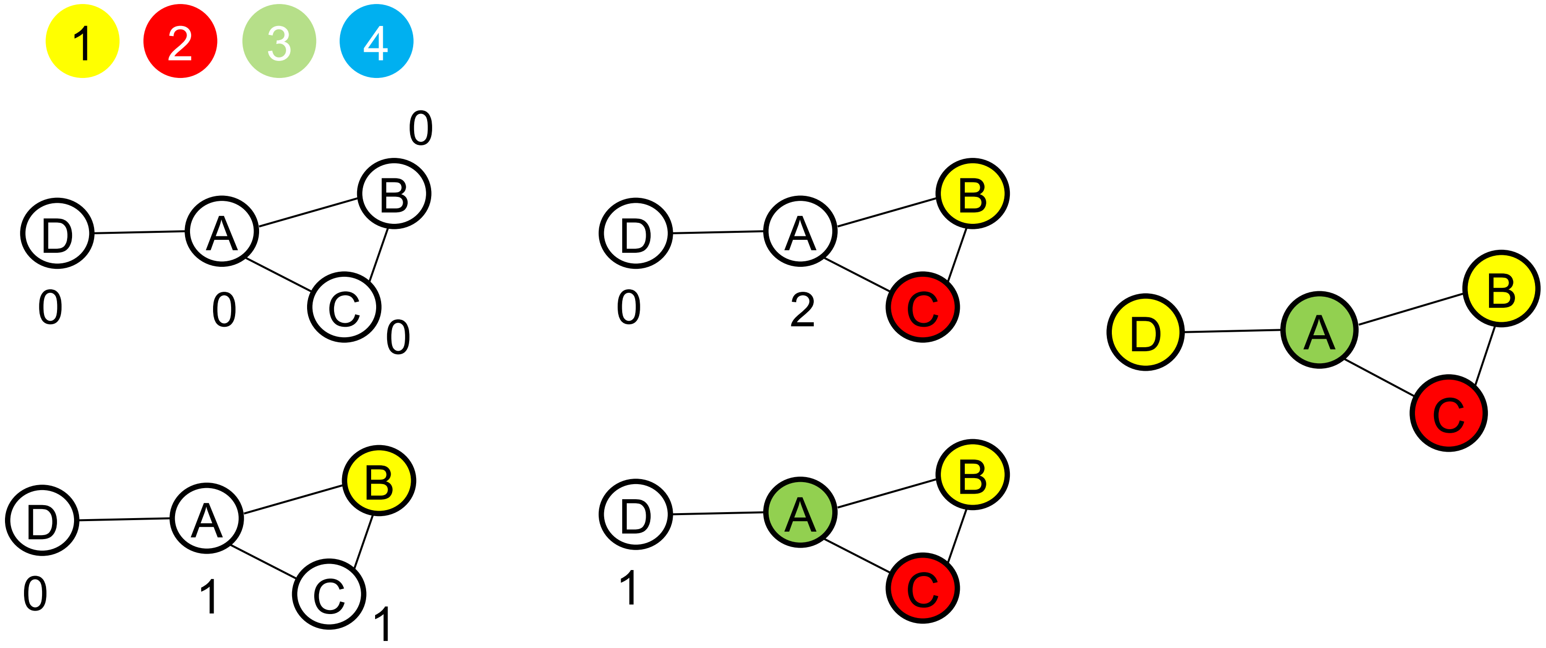
Graph Colouring Heuristics

- Largest Degree:
  - Compute the degree of all vertices
  - Sort the vertices from **largest degree** to smallest
  - Colour the first vertex in the list with the next colour (starting with the first) that is different than its neighbours
  - Delete the vertex from the list go to the previous step unless no vertices left.
  - Example:



Graph Colouring Heuristics

- Saturation Degree:
  - Use saturation degree at each step in the previous approach
  - Example:



Examination timetabling

- A number of exams *(e1, e2, e3, …, eE)*, taken by different students *(s1, s2, s3, …, sS)*, need to be scheduled to a limited time periods *(t1, t2, t3, …,tT)* and certain rooms *(r1, r2, r3, …, rR)*
- Hard Constraints
  - Exams taken by common students can't be assigned to the same time period
  - Room capacity can't be exceeded
- Soft Constraints
  - Separation between exams
  - Large exams scheduled early

Designing a Local Search Metaheuristic for Examination Timetabling

- *Representation*: An array of pair of integers, one representing the period assignment and the other representing the room assignment. The array size is the number of events *E* and each period entry has a value from 1 to *T*, while room has a value from 1 to *R (integer encoding)*
- *Initilisation:* randomly assign a period and a room (integer values within the given range) for each event
- *Objective function*: Number of constraint violations
- *Neighbourhood (perturbation) operator*:
  - OP1: Randomly pick an event and reschedule to random period
  - OP2: Randomly pick an event and assign a different room
  - OP3: Randomly pick an event and reschedule to random period, also assign a different room
  - Any of the above can be parametrised, e.g. pick X number of events and apply an operator
  - More elaborate operators can be designed

Designing Iterated Local Search for Examination Timetabling

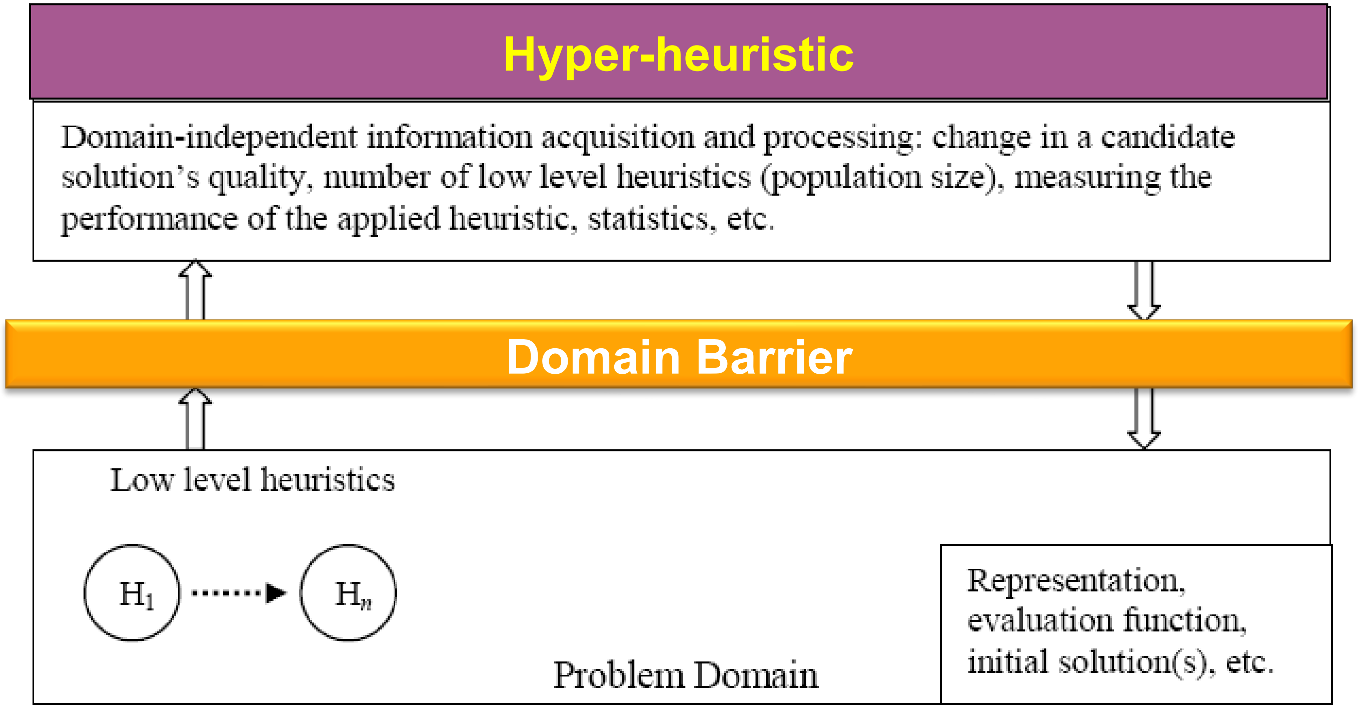
- *Local Search:* RMHC, DHC, SDHC, NDHC, in which the neighbourhood operator is OP3
- *AcceptanceCriterion*: accept improving and equal moves (non-worsening): accept a new solution s' if and only if $f(s') \leq f(s^*)$

Designing Simulated Annealing for Examination Timetabling

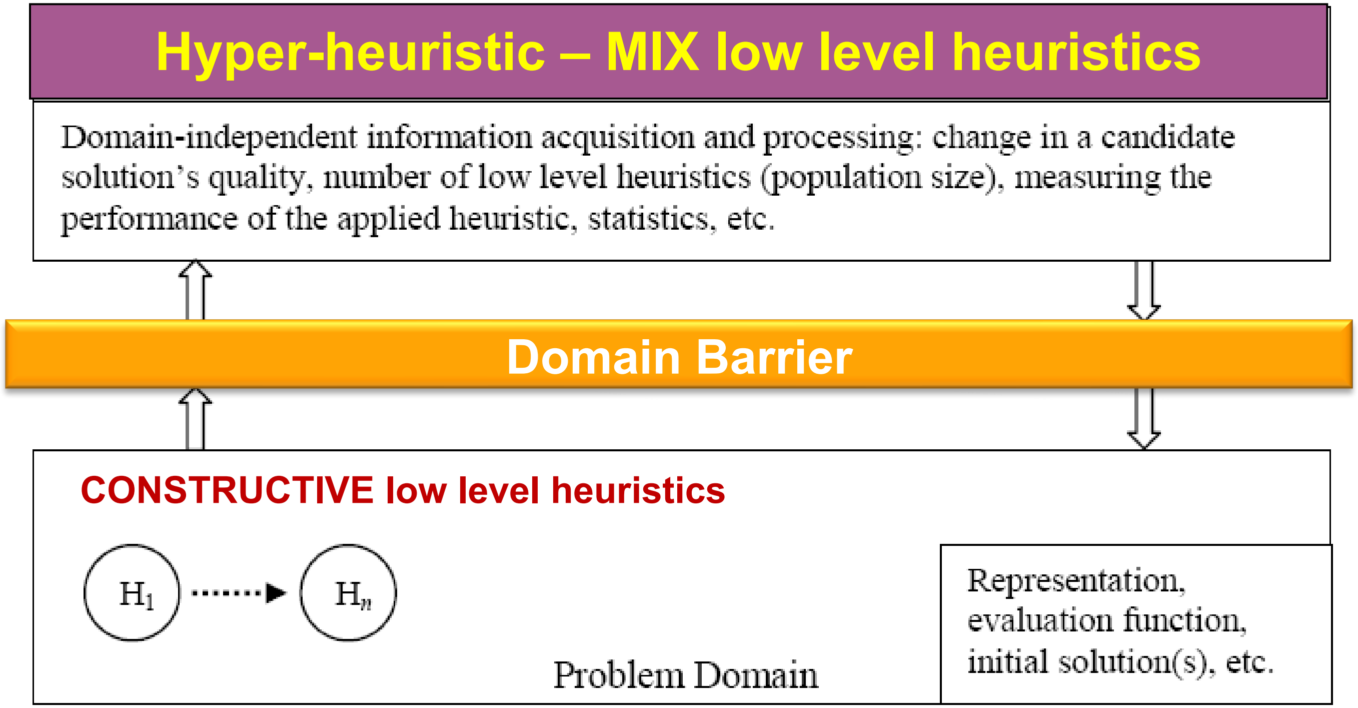
- *Initial Temperature:* Objective value of the initial solution generated
- *Cooling Schedule*: Geometric cooling, α=0.99

- *Termination criteria*: Stop and return the best solution found so far when the total number of violation is 0 or maximum number of iterations is exceeded

A Hyper-heuristic Framework – revisited



Methodologies to select constructive heuristics



Graph-based hyper-heuristics

Online learning

Hyper-heuristics

Classification

Heuristic selection

Methodologies to select

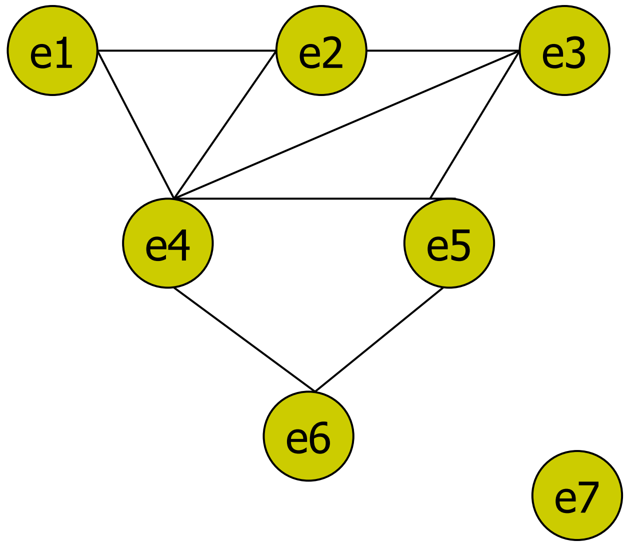
constructive heuristics

- A general framework (GHH)

  employing a set of low level constructive graph colouring heuristics
- Low level heuristics: sequential methods that order events by the difficulties of assigning them
  - 5 graph colouring heuristics
  - Random ordering strategy
- Applied to exam and course timetabling problem

Examination timetabling

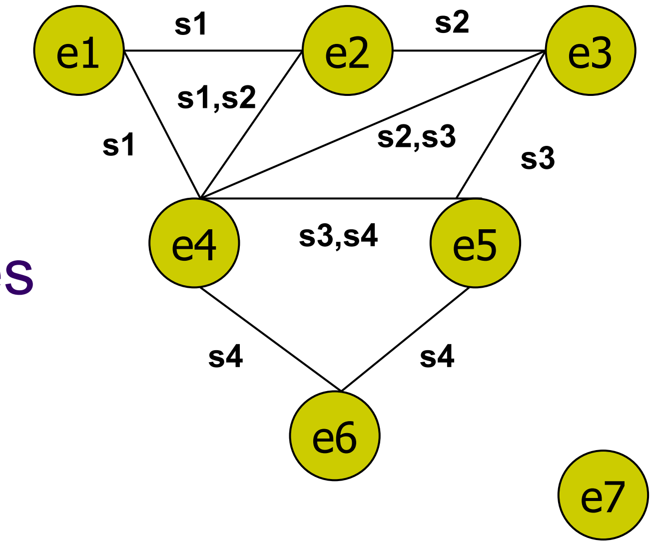
- How can we represent/model this problem?
  - There are 7 exams, e1 ~ e7
  - 5 students taking different exams
    - s1: e1, e2, e4
    - s2: e2, e3, e4
    - s3: e3, e4, e5
    - s4: e4, e5, e6
    - s5: e7
  - let’s ignore room allocation



Examination timetabling

Can be modelled as a graph colouring problem

- Nodes: exams
- Edges: adjacent exams (nodes) have common students
- Colours: time periods
- Objective: assign colours (time periods) to nodes (exams), adjacent nodes with different colour, minimising time periods used



Pseudo-code of Tabu Search graph based hyper-heuristic

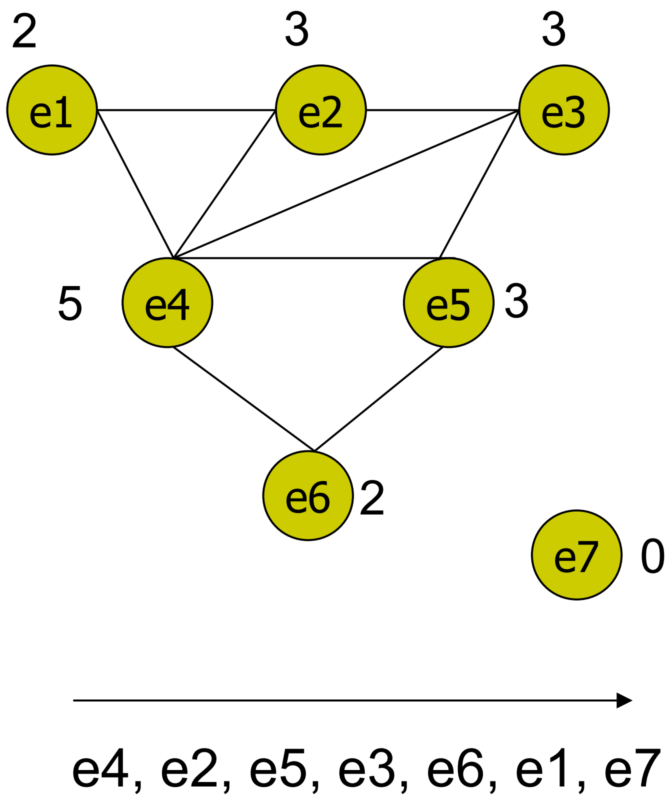
```
initial heuristic list hl = {h_1 h_2 h_3 … h_k}
//Begin of Tabu Search
for i = 0 to i = (5 * the number of events) //number of iterations
    h = change two heuristics in hl //a move in Tabu Search
    if h does not match a heuristic list in 'failed list'
       if h is not in the tabu list //h is not recently visited
          for j = 1  to j = k //h is used to construct a complete solution
             schedule the first N events in the event list ordered using h_j
          if no feasible solution can be obtained
             store h into the 'failed list' //update "failed list"
          else if cost of solution c < the best cost c_g obtained
             save the best solution, c_g = c //keep the best solution
             add h into the tabu list
             remove the first item from the tabu list if its length > 9
             hl = h
    //end if
    Deepest descent on the complete solution obtained
//end of Tabu Search
output the best solution with cost of c_g
```

number of events = $N * k$

Graph-based hyper-heuristics

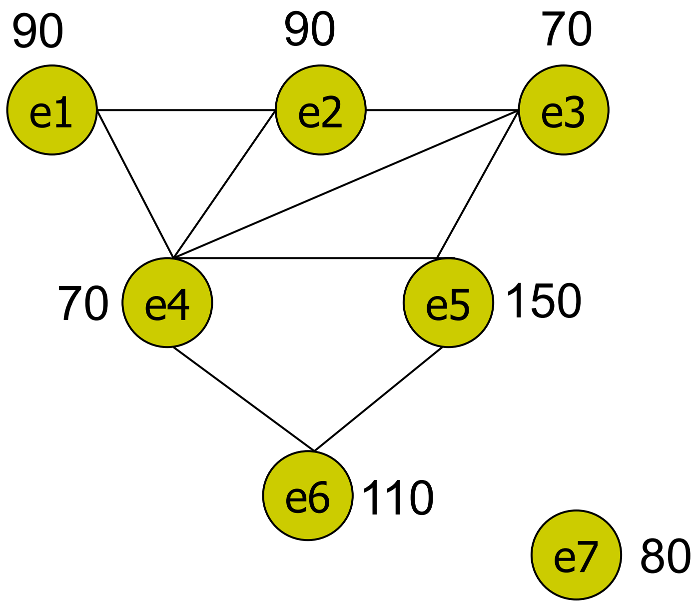
Graph Coloring Heuristics	Ordering strategies
Largest degree (**L**D)	Number of clashed events
Largest weighted degree (LW)	LD with number of common students
Largest enrolment (LE)	Number of students
Saturation degree (**S**D)	Number of valid remaining time periods
Colour degree (**C**D)	Number of clashed events that are already scheduled
+	
Random ordering (**R**O)	Orders a number events randomly

Example: Largest Degree Graph Colouring/Construction/Ordering Heuristic



Construct a timetable using one event at a time in a given order

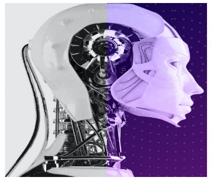
Example: Largest Enrolment Graph Colouring/Construction/Ordering Heuristic



e5, e6, e1, e2, e7, e4, e3

Construct a timetable using one event at a time in a given order

Graph-based hyper-heuristics



events

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11	e12	...

heuristic list

SD	SD	LD	CD	LE	SD	SD	LW	SD	LD	CD	RO	...

Schedule *N* events using a graph colouring heuristic

order of events

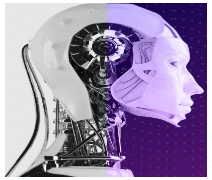
e1	e9	e3	e26	e25	e6	e17	e28	e19	e10	e31	e12	...

N=5

slots

e1 e9	e3		e26	e25								

Graph-based hyper-heuristics



Delete scheduled events

events

	e2		e4	e5	e6	e7	e8		e10	e11	e12	...

Repeat the process until all events are scheduled

heuristic list

SD	SD	LD	CD	LE	SD	SD	LW	SD	LD	CD	RO	...

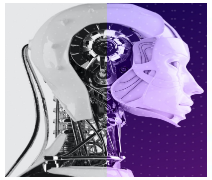
order of events

e6	e17	e28	e19	e10	e31	e12	e5	e22	e32	e27	e19	...

slots

e1 e9	e3	e6 e19	e26	e25	e28	e17	e10					

Graph-based hyper-heuristics



events

	e2		e4	e5		e7	e8			e11	e12	...

heuristic list

SD	SD	**LD**	CD	LE	SD	SD	LW	SD	LD	CD	RO	...

order of events

e5	e32	e19	e22	e13	e31	e12	e7	e2	e15	e27	e12	...

slots

e1 e9	e3	e6 e19	e26	e25	e28	e17	e10	e5 e13	e32 e19	e13		

Graph-based hyper-heuristics

- Tabu Search at the high level
  - Neighbourhood operator: randomly change two heuristics in the heuristic list
  - Objective function: quality of solutions built by the corresponding heuristic list
  - Tabu list: visits to the same heuristic lists forbidden
- Other high-level search strategies tested
  - Steepest Descent
  - Variable neighbourhood search → best performing
  - Iterated Steepest Descent

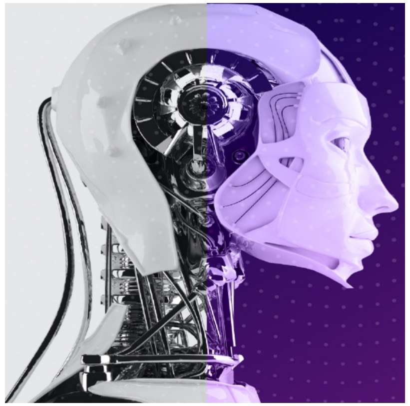
Results

Carter's benchmark

Problem	SL	SLR	SCL	SCLR	SCLx	SCLxR	SCLxR(10*)
car91	5.78	5.67	5.52	**5.36**	5.65	5.43	5.39
car92	4.76	4.68	4.21	**4.14**	4.53	4.78	4.63
ear83	38.8	38.57	38.68	38.5	**37.92**	38.22	38.03
hec92	12.35	12.27	12.30	12.81	12.39	12.25	***12.11***
kfu93	16.21	15.79	15.37	15.23	15.67	15.2	***15.12***
lse91	12.17	11.36	12.09	11.93	11.56	**11.33**	***11.33***
sta83	164.01	163.5	164.06	163.31	**158.19**	160.19	159.32
tre92	9.15	9.13	8.87	9.08	**8.75**	9.03	8.97
ute92	29.49	29.17	28.27	28.19	**28.01**	28.21	28.11
uta93	4.12	4.03	4.05	3.98	**3.88**	3.95	***3.78***
york83	44.54	42.67	42.44	42.37	**41.37**	42.01	41.52

Costs of solutions obtained by GHH upon a different number of heuristics (S: saturation degree, L: largest degree, C: color degree; R: random ordering, Lx: largest weighted)

Generation Hyper-heuristics



Computational Optimisation & Learning Lab

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Genetic Programming (GP)

- Challenge:

  "Get a computer to do what needs to be done, without telling it how to do it."
- GP provides a method for automatically creating a working computer program from a high-level problem statement of the problem (i.e., program synthesis or program induction)
- GP iteratively transforms a population of computer programs into a new generation of programs via evolutionary process

Why Genetic Programming?

- “It is difficult, unnatural, and overly restrictive to attempt to represent hierarchies of dynamically varying size and shape with fixed length character strings.” “For many problems in machine learning and artificial intelligence, the most natural representation for a solution is a computer program.” [Koza, 1994]

- A parse tree is a good representation of a computer program for Genetic Programming

Some selected real-world applications of Genetic Programming

- Automated design of mechatronic systems (NSF)
- Climatology: Estimation of heat flux between the atmosphere and sea ice, modelling global temperature changes
- Clinical decision support in ophthalmology
- Container loading optimisation
- Scheduling, timetabling
- Machine learning
- Vehicle routing …

A Computer Program in C

```
int foo (int time)
{
    int temp1, temp2;
    if (time > 10)
        temp1 = 3;
    else
        temp1 = 4;
    temp2 = temp1 + 1 + 2;
    return (temp2);
}
```

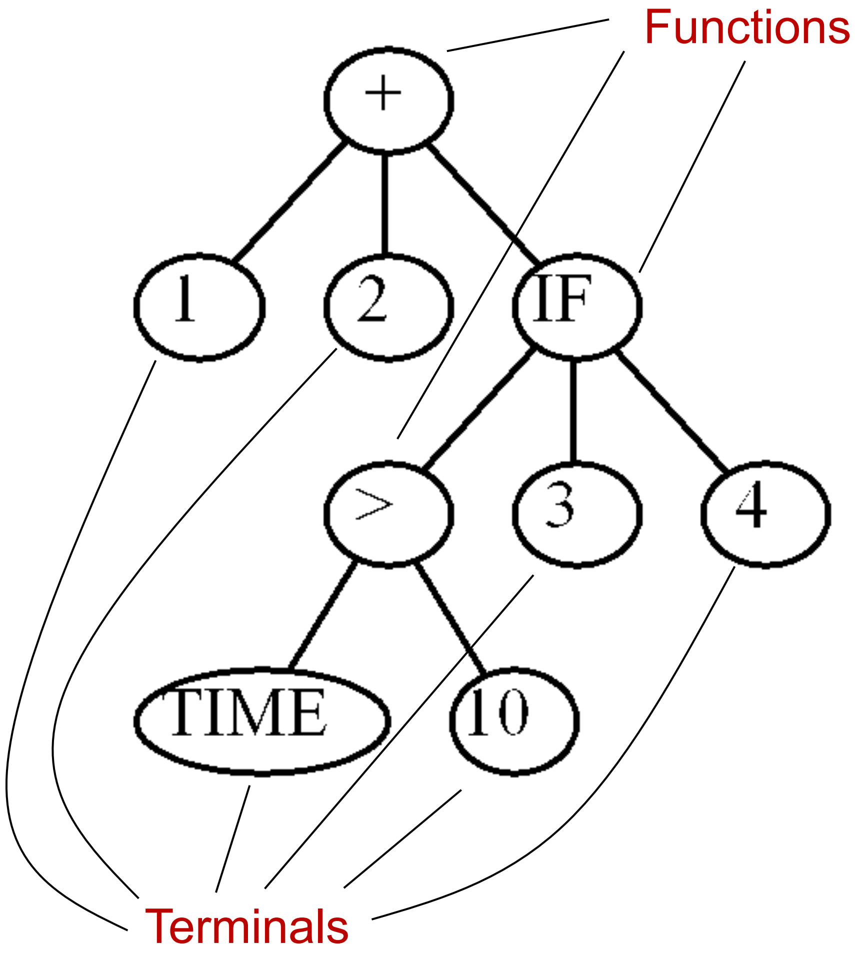
What would be the output of this code as time changes from 0 to 14?

A Computer Program in C

- If we have the observations on the right, can we reverse engineer the mapping function?

Time	Output
0	7
1	7
2	7
3	7
4	7
5	7
6	7
7	7
8	7
9	7
10	7
11	6
12	6
13	6
14	6

Using Trees To Represent Computer Programs




```
(+ 1 2 (IF (> TIME 10) 3 4))
```

Genetic Operations

- GP is an evolutionary algorithm containing the same algorithmic components, including:
  - Random generation of the initial population of possible solutions (programs)
  - Genetic crossover of two promising solutions to create new possible solutions (programs)
  - Mutation of promising solutions to create new possible solutions (programs)

Randomly Generating Programs

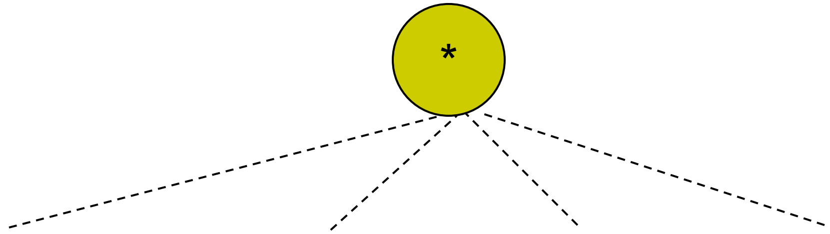
- Randomly generate a program that takes two (or more) arguments and uses basic arithmetic to return an answer
  - Function set = {+, -, *, /}
  - Terminal set = {integers, X, Y}
- Randomly select either a function or a terminal to represent our program
- If a function was selected, recursively generate random programs to act as arguments

Randomly Generating Programs

Random item: *

(* …) – assume multiplication with four arguments: arg1 + arg2 + arg3 + arg4

randomly create each argument one by one

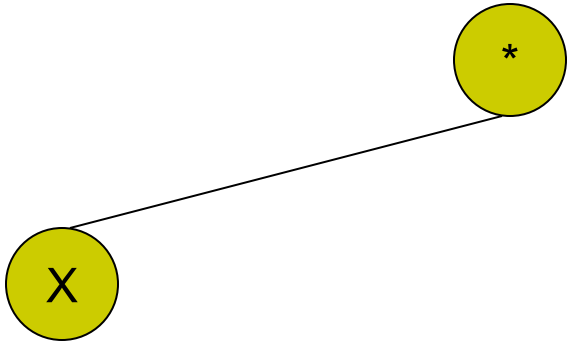


Randomly Generating Programs

<u>arg1</u> is a terminal, so stop

(* X …)

<u>Random item:</u> X

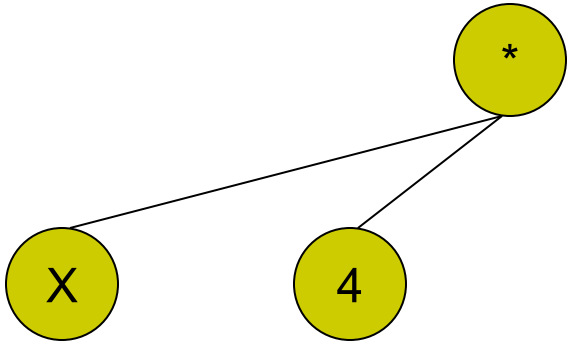


Randomly Generating Programs

arg2 is terminal stop

(* X 4 …)

Random item: 4



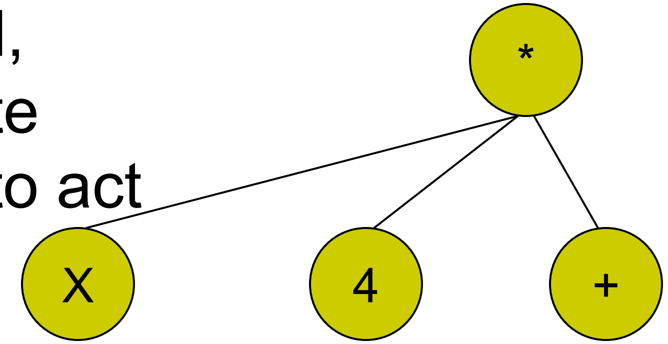
Randomly Generating Programs

arg3 is a function

(* X 4 (+ …) …)

assume addition requires two arguments, since a function is selected, recursively generate random programs to act as arguments

Random item: +

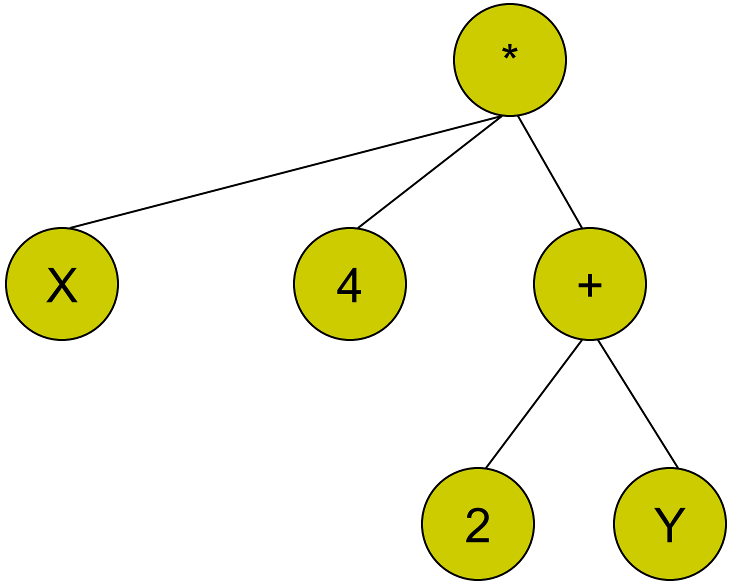


Randomly Generating Programs

Random items: 2, Y

(* X 4 (+ 2 Y) …)

arguments of addition are terminals, so stop recursion for each



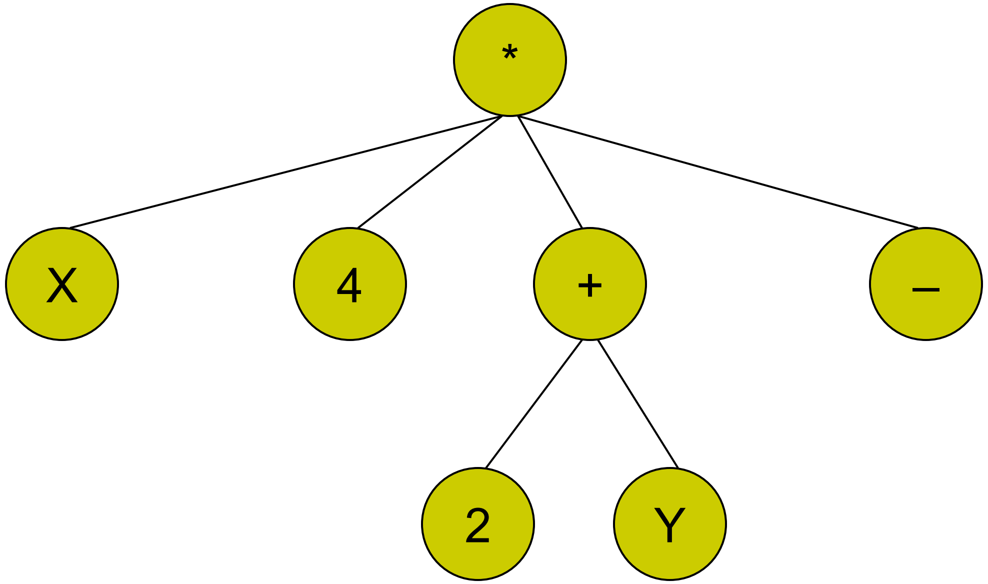
Randomly Generating Programs

arg3 is a function

Random item: –

(* X 4 (+ 2 Y) (– …))

assume subtraction requires two arguments, since a function is selected, recursively generate random programs to act as arguments

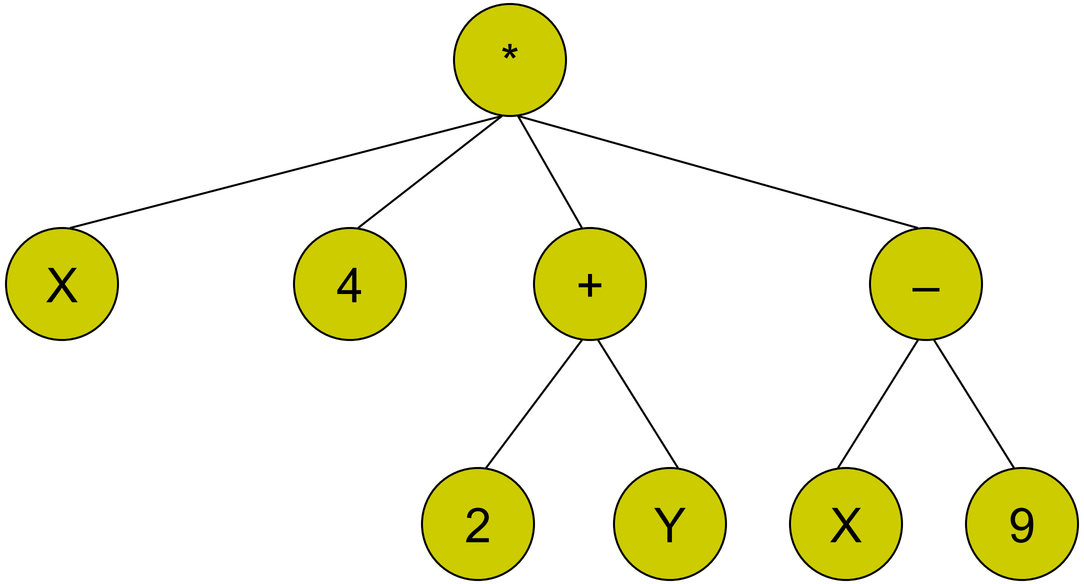


Randomly Generating Programs

Random items: X, 9

(* X 4 (+ 2 Y) (– X 9))

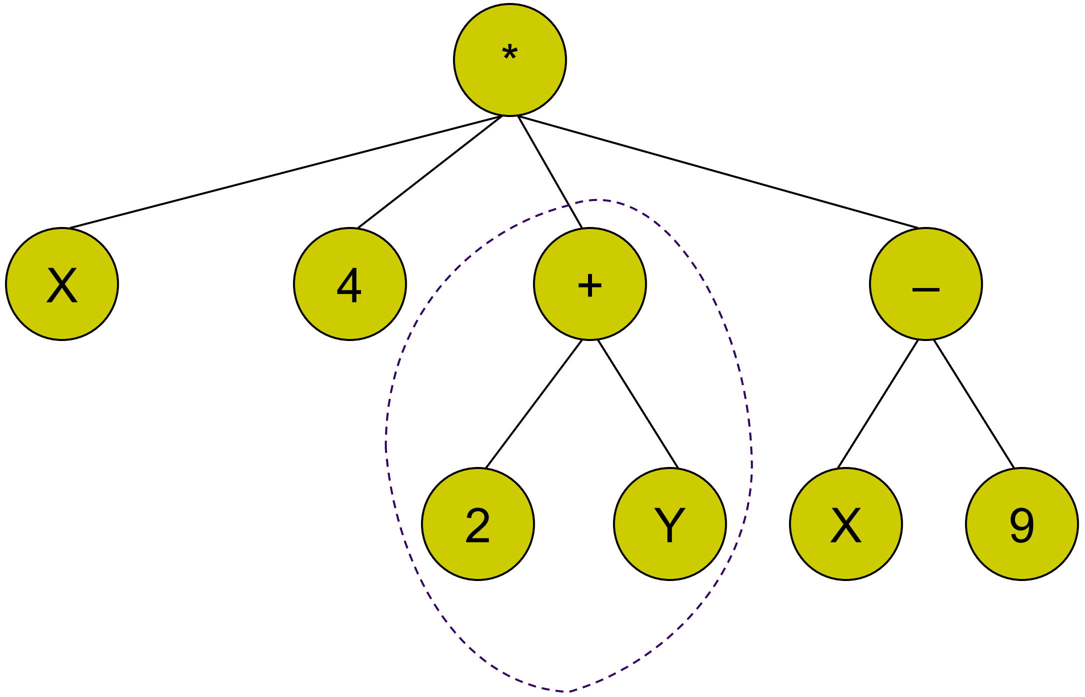
arguments of subtraction are terminals, so stop recursion for each



Mutation

First pick a random node (e.g., node(+))

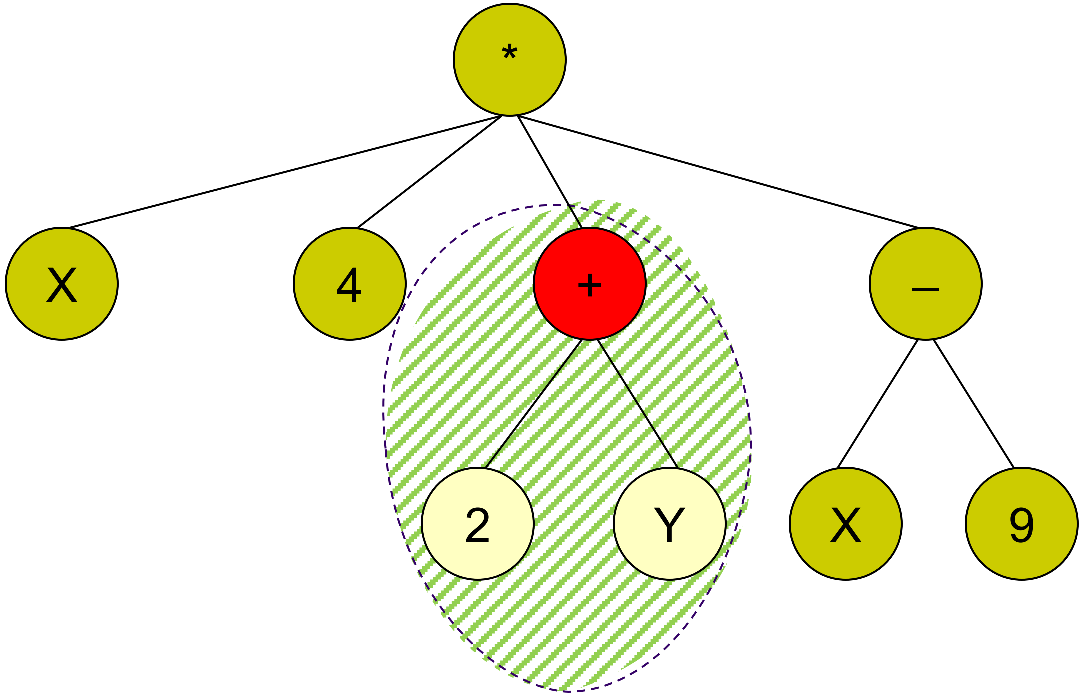
(* X 4 (+ 2 Y) (– X 9))



Mutation

First pick a random node (e.g., node(+)),
Delete that node and its children

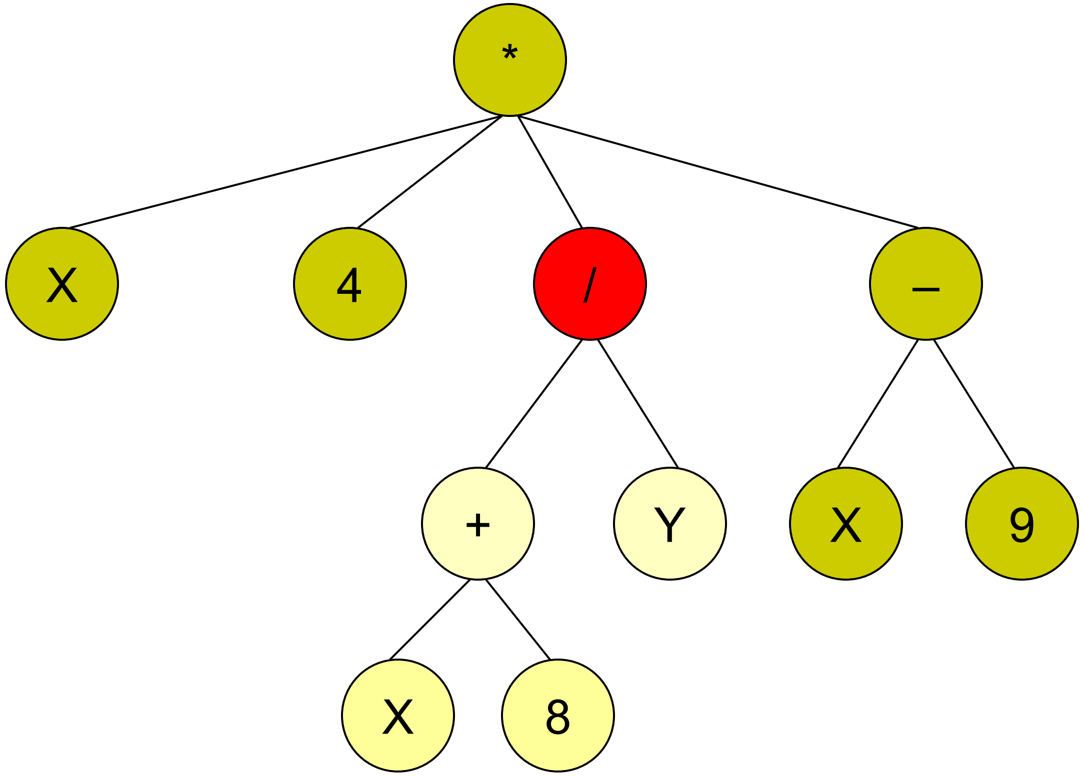
(* X 4 **(+ 2 Y)** (– X 9))



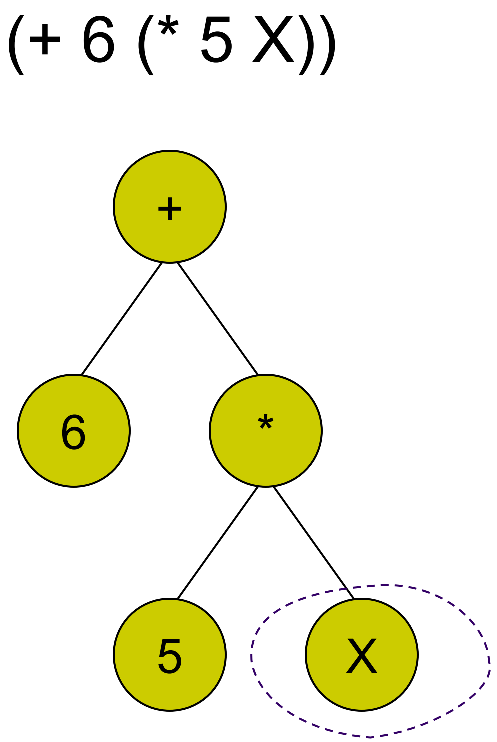
Mutation

Insert a randomly generated program in place of the deleted node (e.g., (\ (+ X 8) Y))

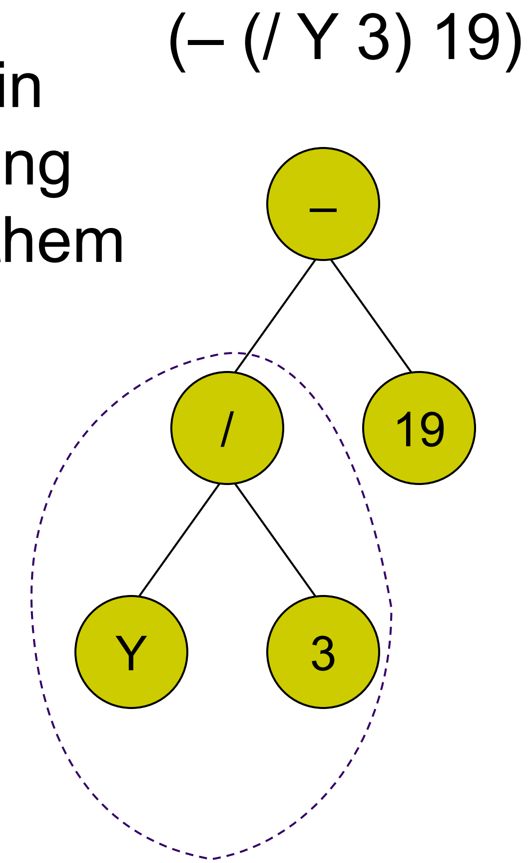
(* X 4 **(\ (+ X 8) Y)** (– X 9))



Crossover

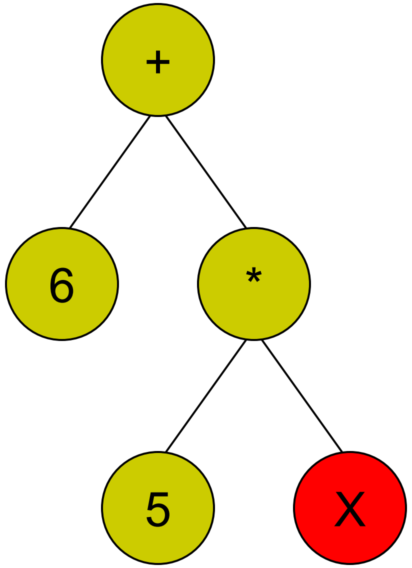


Pick a random node in each program including the subtrees having them as the root nodes (e.g., X and /)



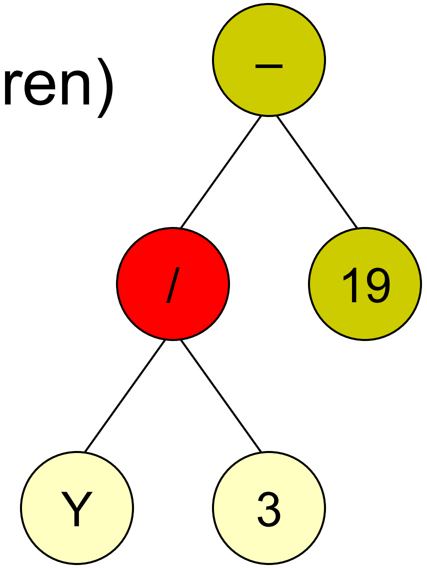
Crossover

(+ 6 (* 5 **X**))



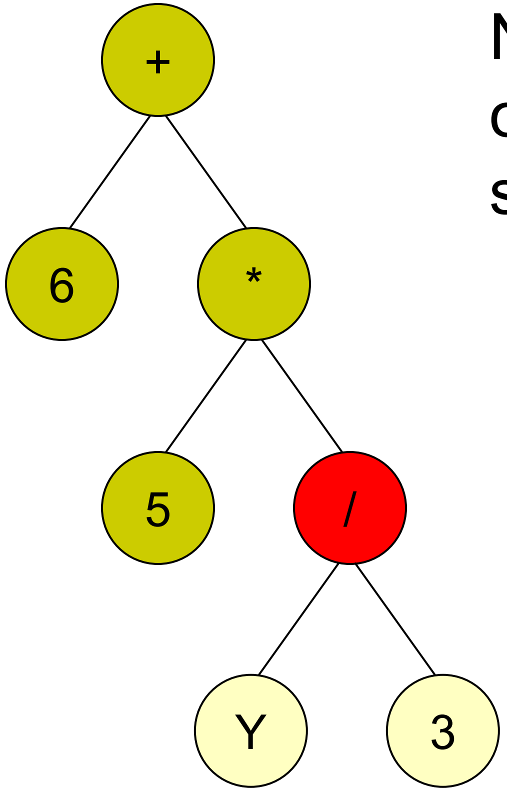
Swap the two nodes
(including their all children)

(– **(/ Y 3)** 19)



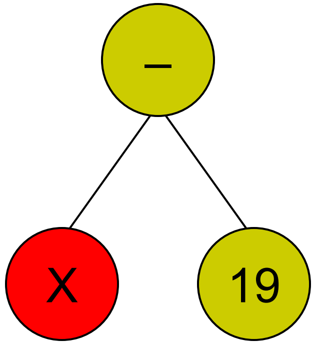
Crossover

(+ 6 (* 5 **(/ Y 3)**))



Now we have two offspring/new candidate solutions

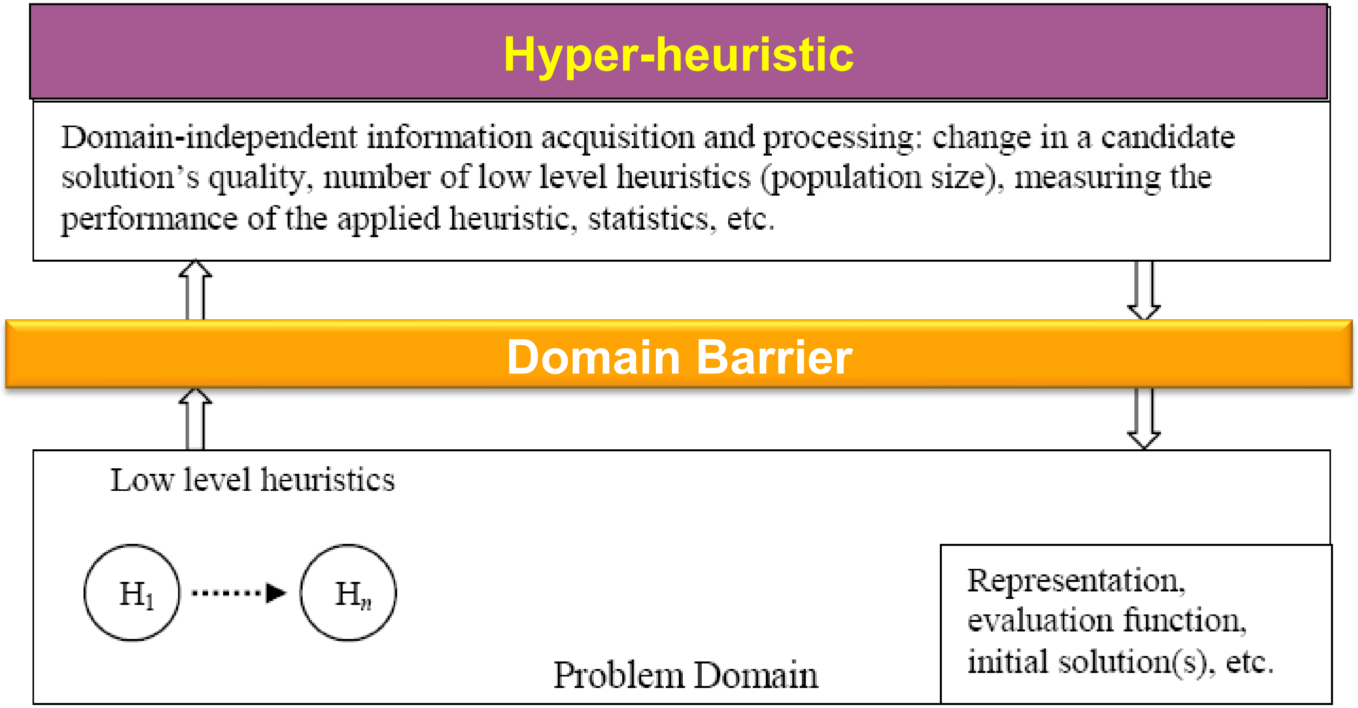
(– **X** 19)



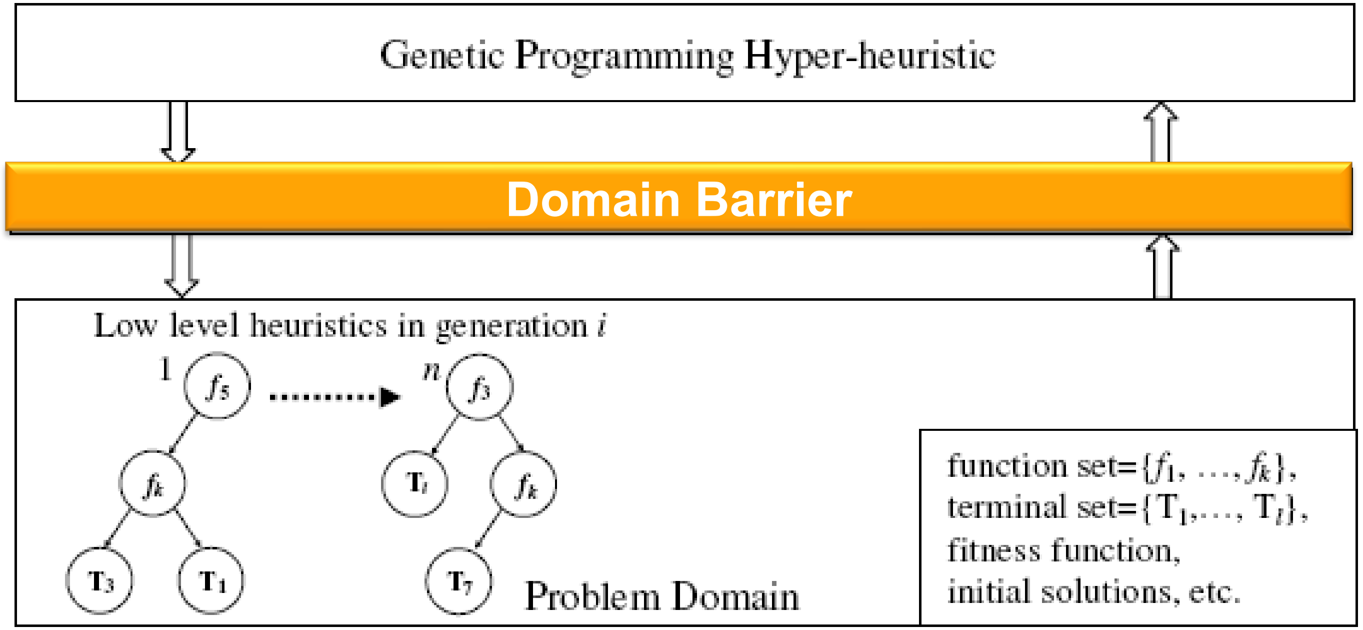
Some Java based Software Libraries

- ECJ: http://cs.gmu.edu/~eclab/projects/ecj/
- TinyGP: http://cswww.essex.ac.uk/staff/rpoli/TinyGP/
- GEVA (grammatical evolution): http://ncra.ucd.ie/Site/GEVA.html
- Cartesian GP resources: http://www.cartesiangp.co.uk/resources.html

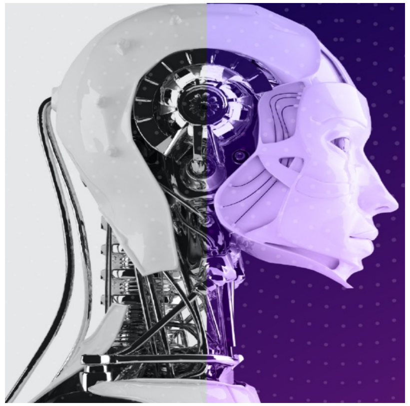
A Hyper-heuristic Framework – revisited



A Generation Hyper-heuristic Framework



Genetic Programming for Packing



from the PhD Thesis (2010) of

Matthew Hyde

Offline learning

Hyper-heuristics

Heuristic generation

Methodologies to generate

constructive heuristics

Computational Optimisation & Learning Lab

The University of Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

1D Offline Bin Packing

Pack a **set** of items of sizes s_i for $i = 1, \ldots, n$

- Sizes are integer values and $s_i \in [1, C]$
- C is the fixed capacity of each bin

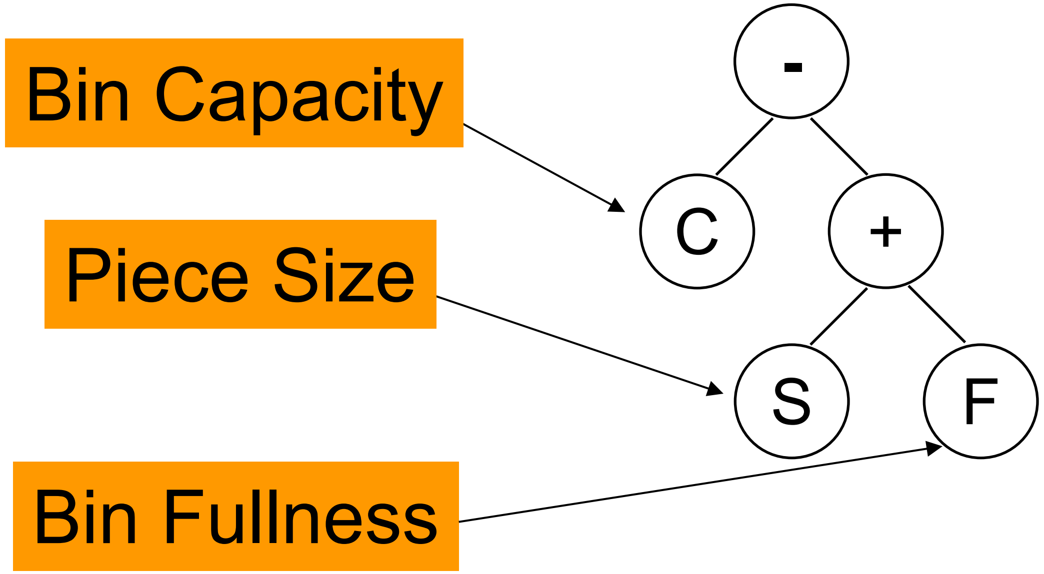
in such a way that

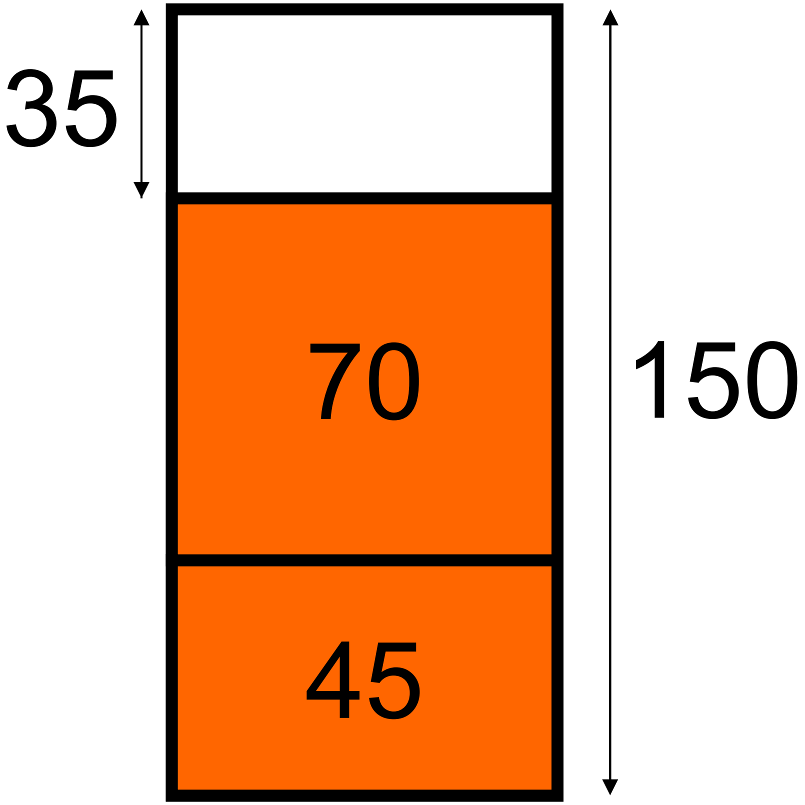
- Never exceed bin capacity
- Minimise number of bins used

Standard NP-hard problem

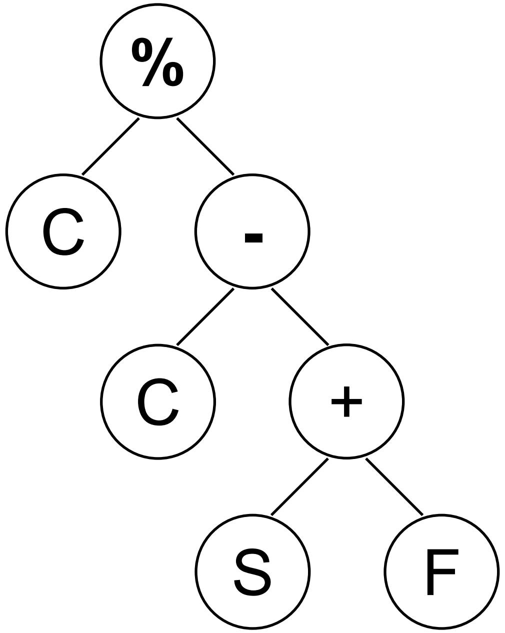
Genetic Programming

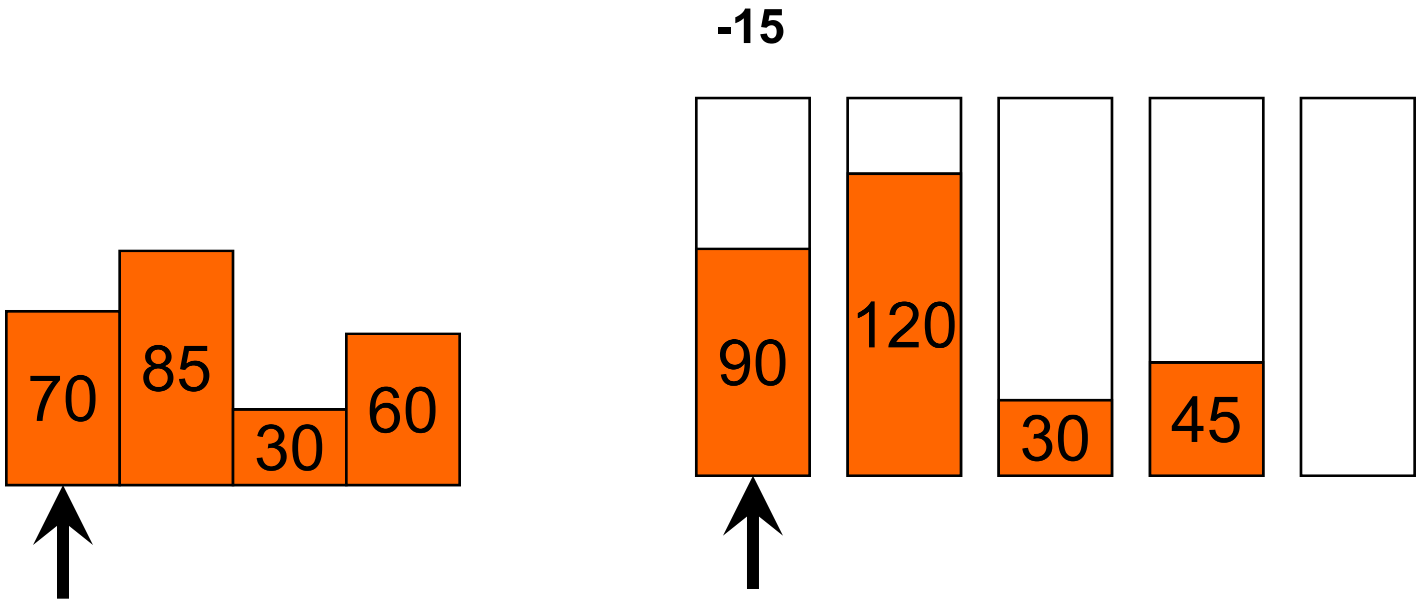
- Evolves trees representing a program
- Following tree is a program that calculates the space left at the top of the bin
- Train and test



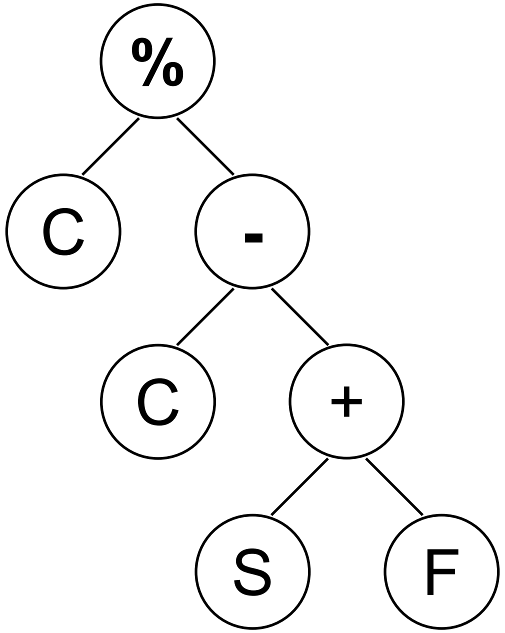


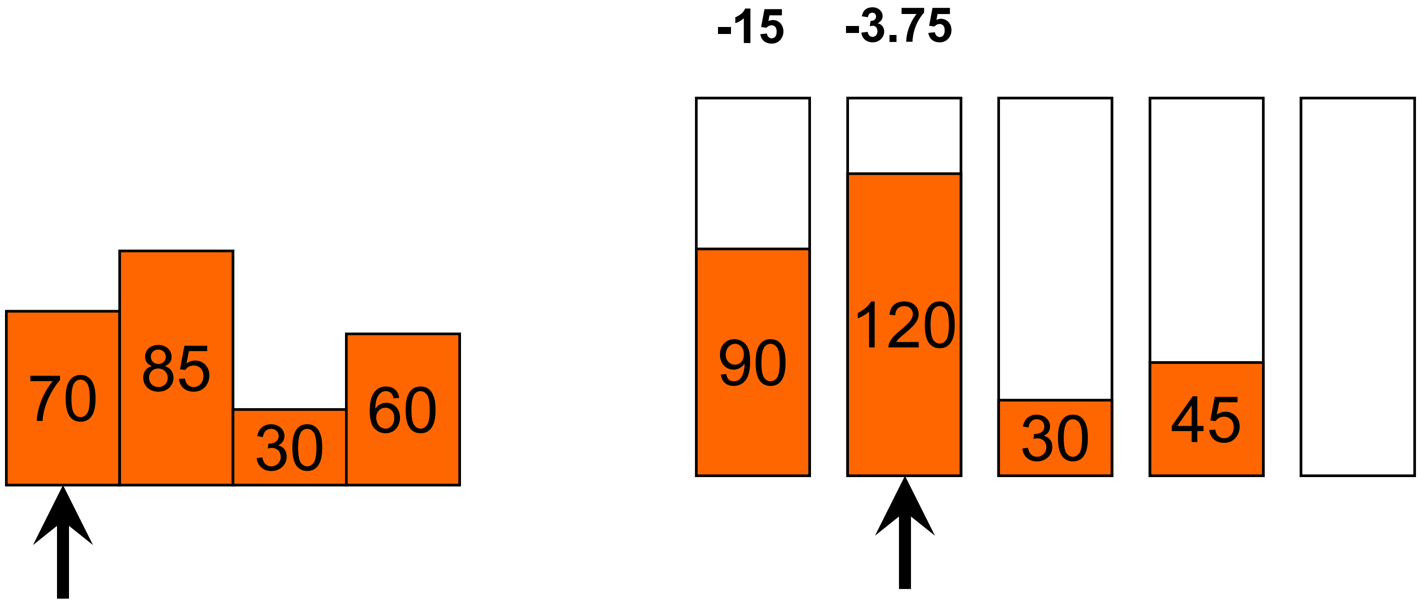
Genetic Programming Heuristics – Bin Packing



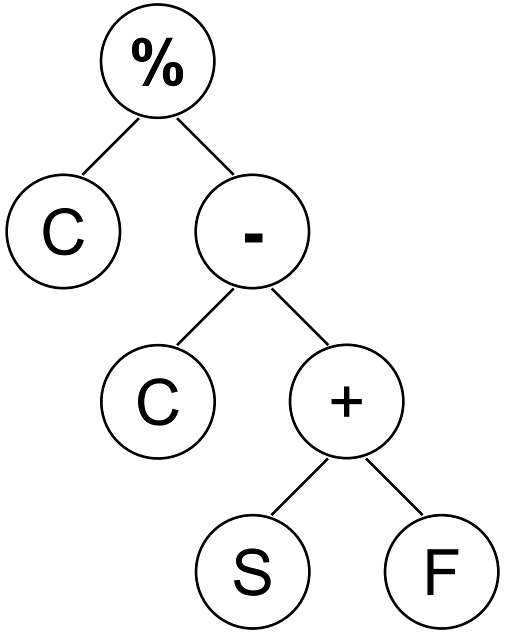


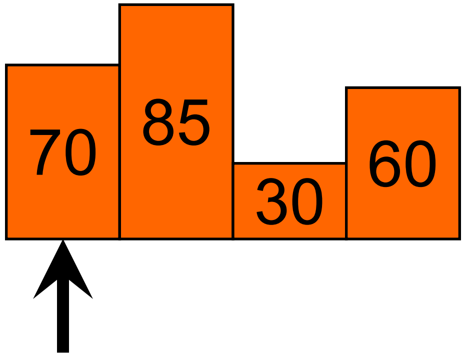
Genetic Programming Heuristics – Bin Packing

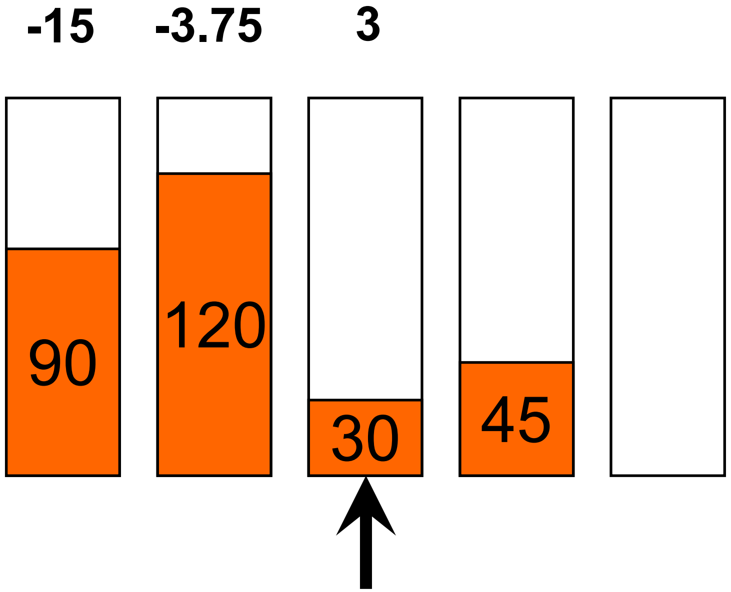




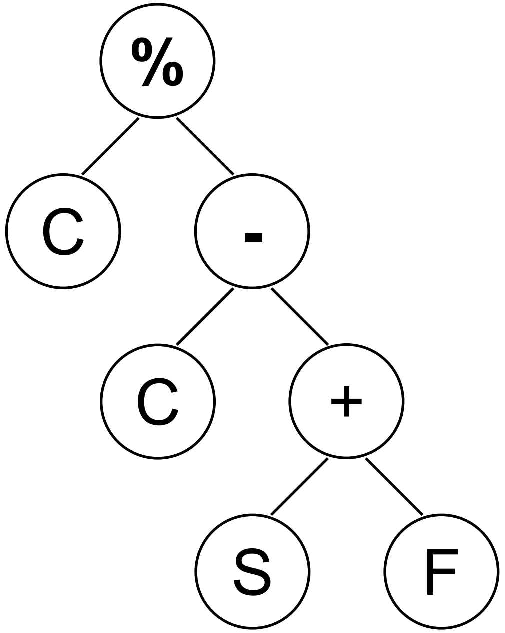
Genetic Programming Heuristics – Bin Packing

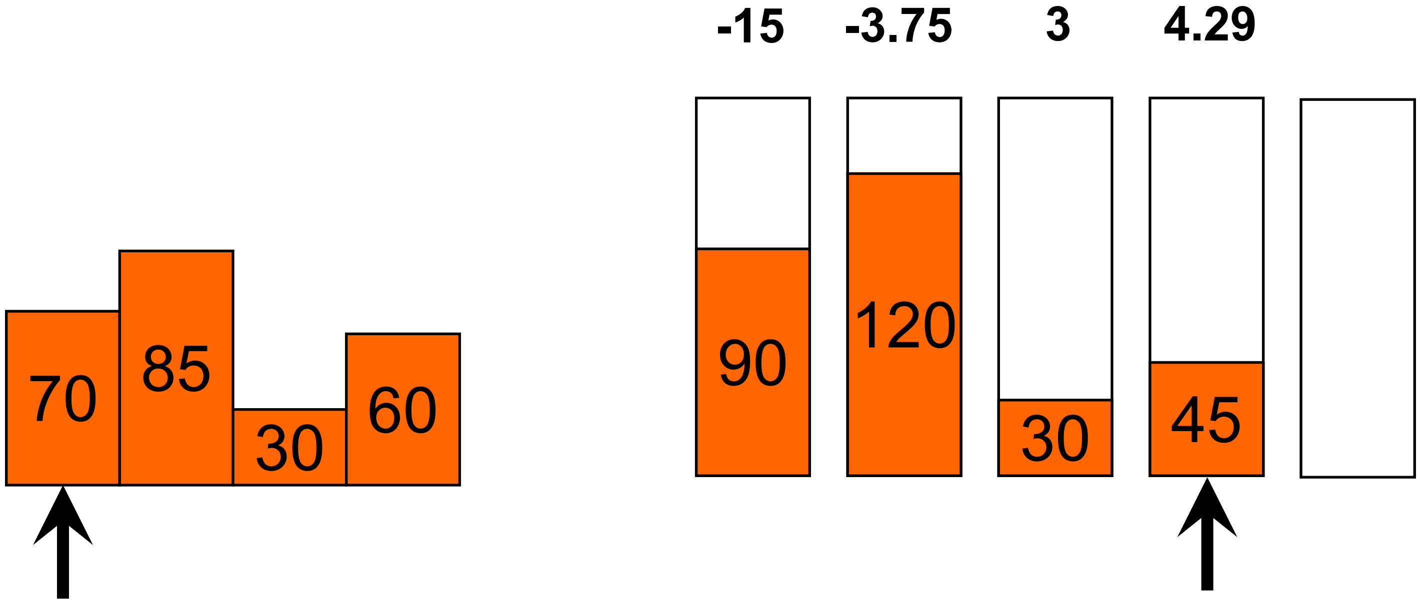




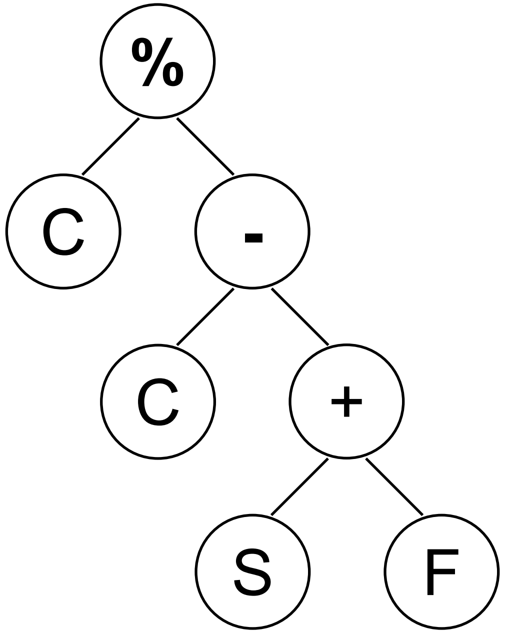


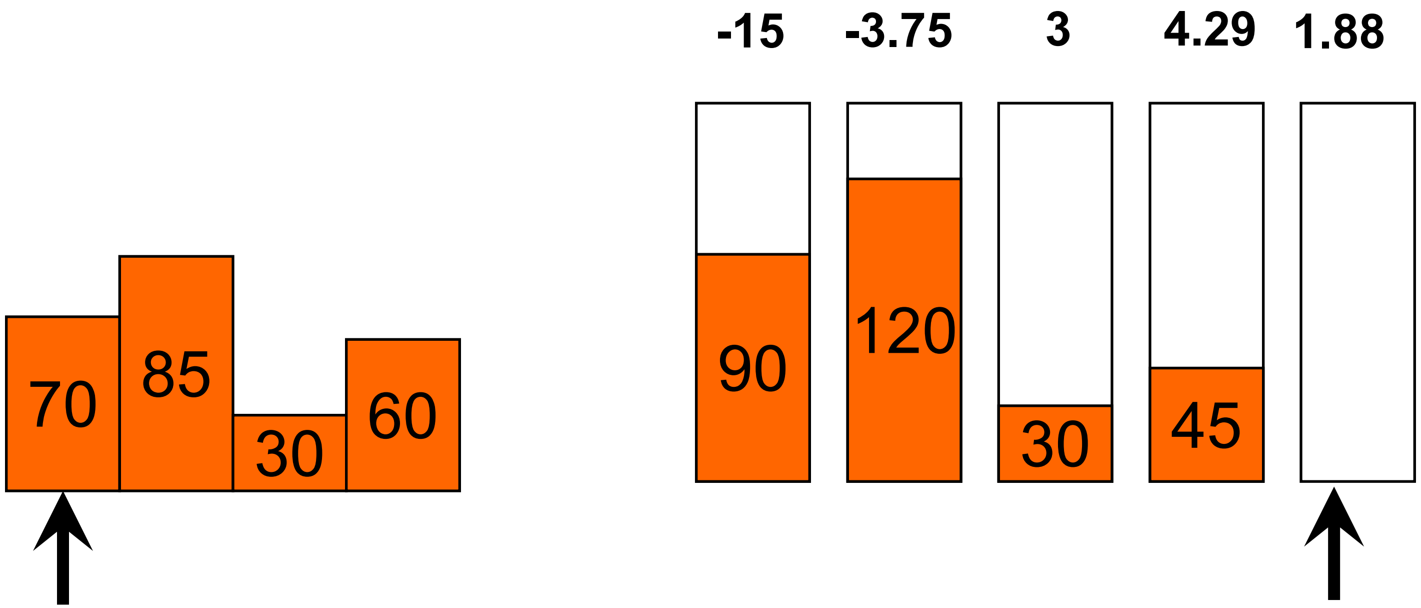
Genetic Programming Heuristics – Bin Packing



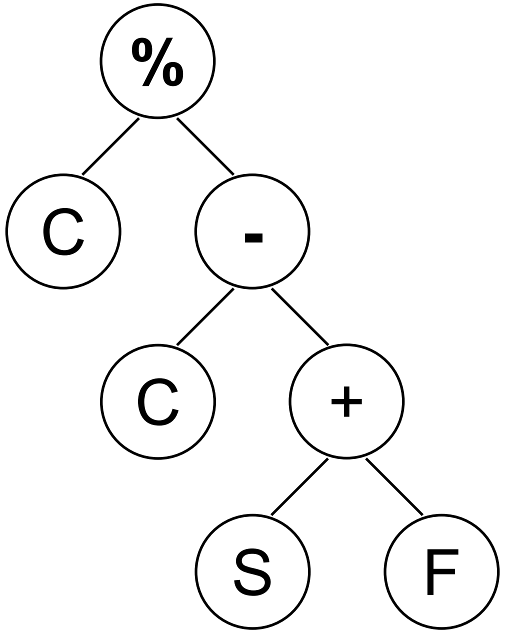


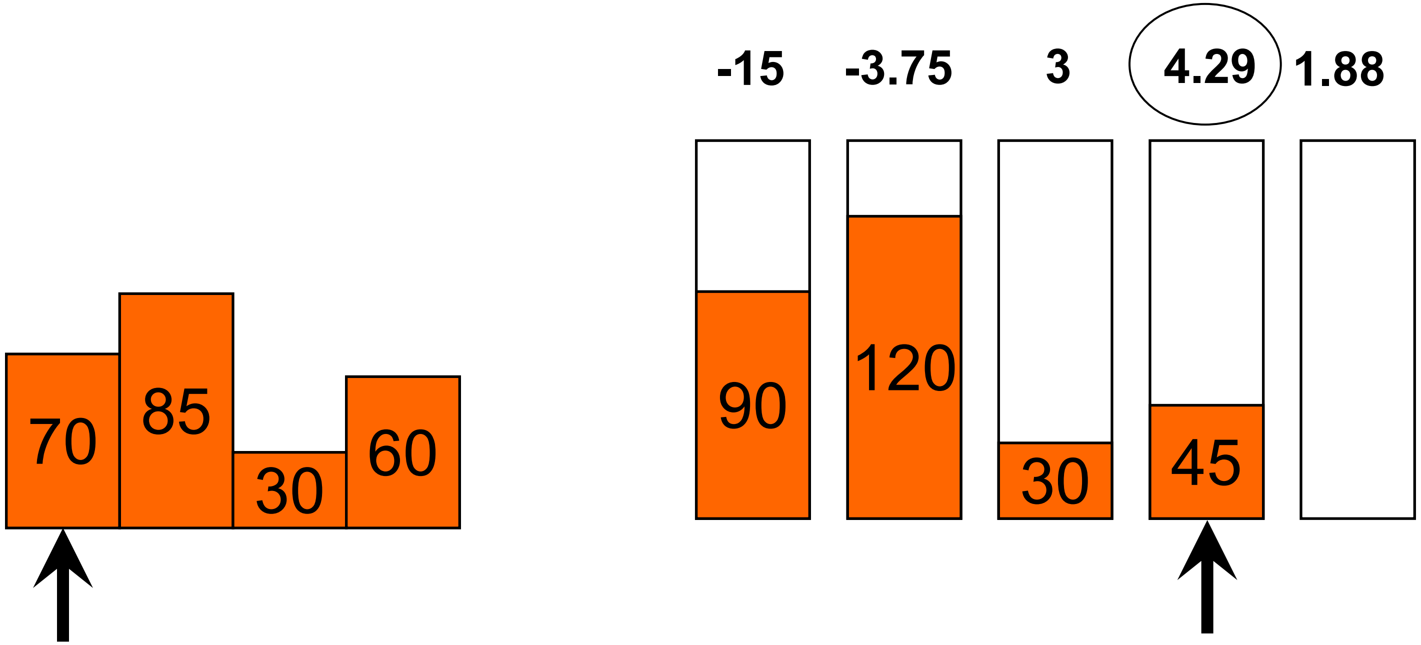
Genetic Programming Heuristics – Bin Packing



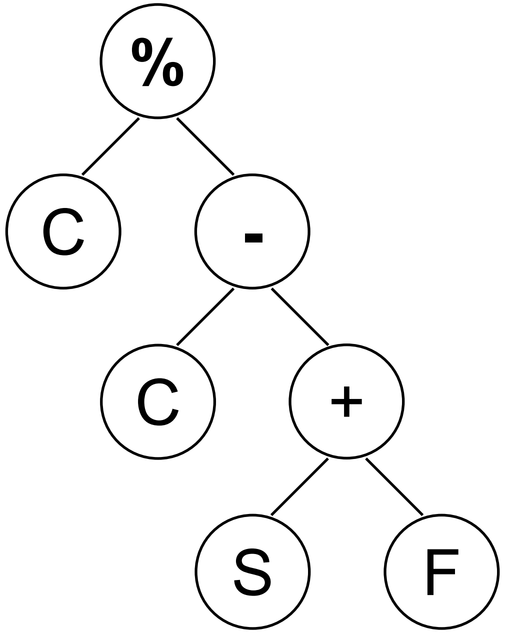


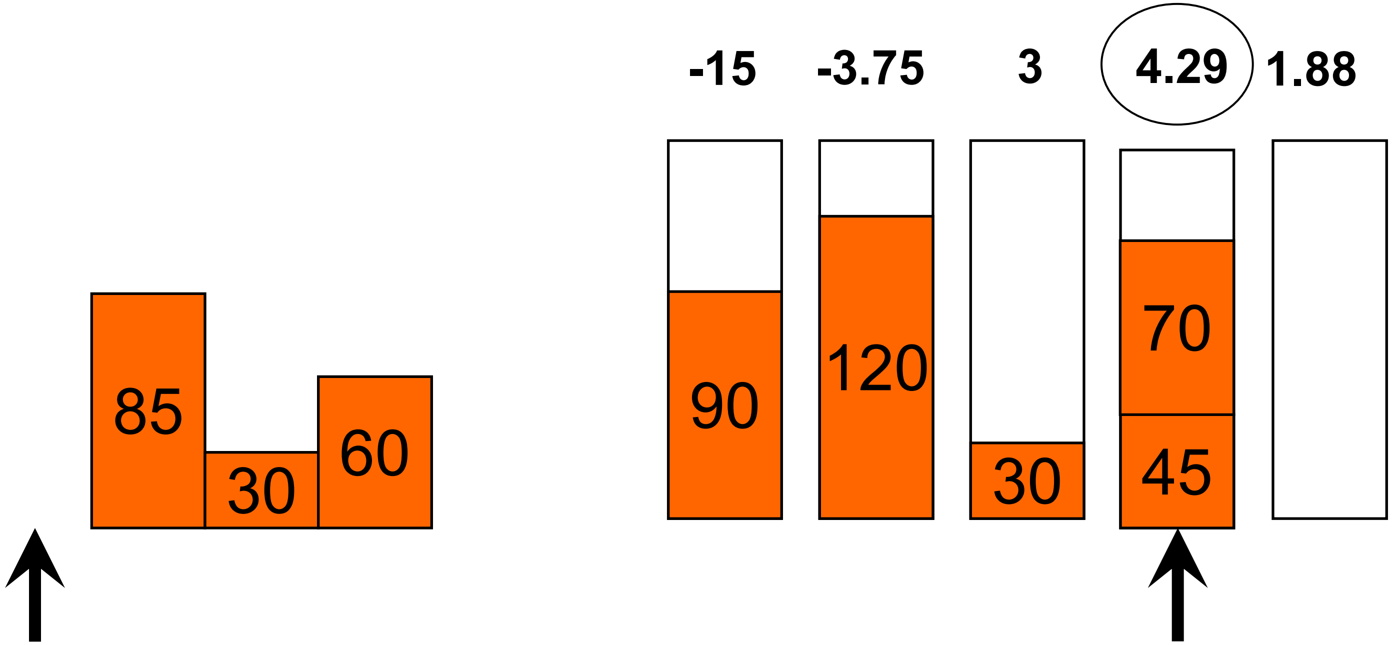
Genetic Programming Heuristics – Bin Packing



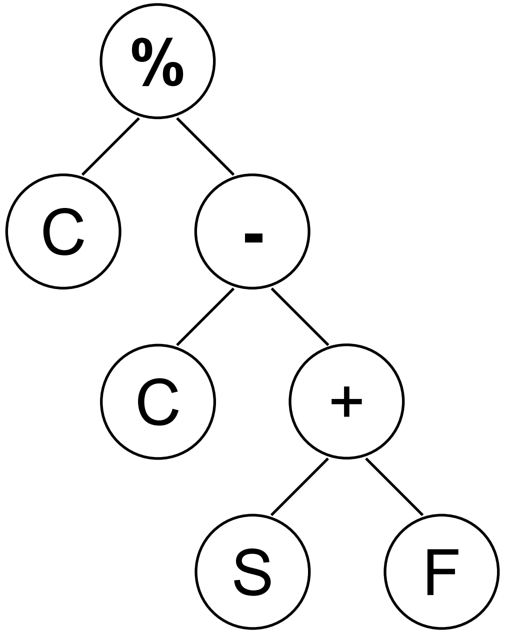


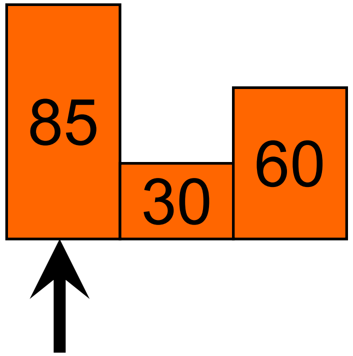
Genetic Programming Heuristics – Bin Packing

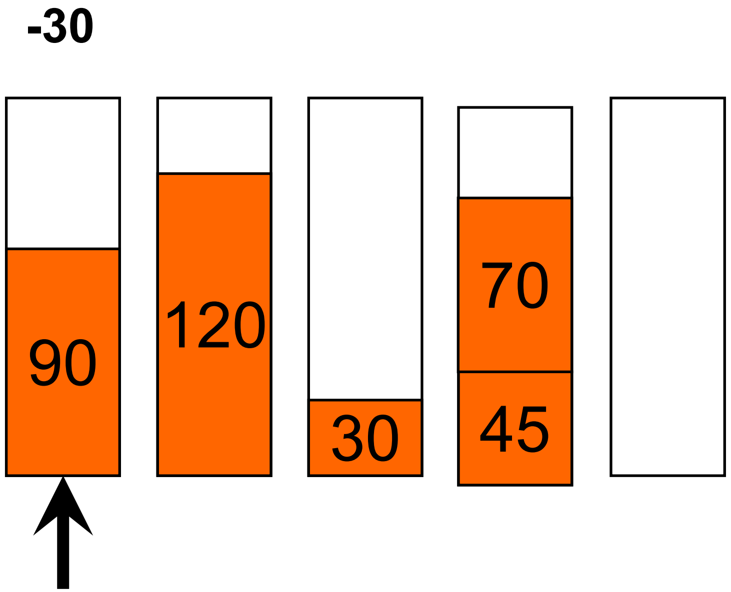




Genetic Programming Heuristics – Bin Packing







Experiments

- Compared against results from 18 Papers
- 1D Bin Packing
  - 2 Instance Sets
- 2D Knapsack and Bin Packing
  - 10 Instance Sets
- 3D Knapsack and Bin Packing
  - 6 Instance Sets

Results

Bin Packing Problem

Dimensions	Instance Name	Percent Improvement
1D	Uniform	0
	Hard	-0.4
2D	Beng	0
	Ngcut	0
	Gcut	-1.2
	Cgcut	0
3D	Thpack9	-2.3

Knapsack Problem

Dimensions	Instance Name	Percent Improvement
2D	Okp	**+1.2**
	Wang	0
	Ep30	-0.9
	Ep50	**+0.4**
	Ep100	-2.6
	Ep200	**+2.4**
	Ngcut	-4.3
	Gcut	**+1.2**
	Cgcut	-1.7
3D	Ep3d20	**+13.0**
	Ep3d40	**+10.2**
	Ep3d60	**+6.0**
	Thpack8	-0.5
	Thpack9	**+0.7**
	BandR	-2.9

GP hyper-heuristic for packing

- A **more general** packing methodology for 1D, 2D and 3D bin packing and knapsack problems

- **Achieved generality without the loss of solution quality**

Summary

- A search method with different components, algorithmic configurations and/or parameter settings often performs differently
- A metaheuristic performs search over space of solutions while a hyper-heuristic (which can be a metaheuristic) performs search over the space of (low level) heuristics
- Selection hyper-heuristics can be used to mix perturbative as well as constructive low level heuristics

Summary II

- The choice of low level heuristics used in a hyper-heuristic approach influences its performance
- Genetic programming hyper-heuristic can be used to build heuristics or heuristic components
  - Often they operate in a train and test fashion and
  - Training on selected sample instances could take long time while application to unseen instances is generally fast
  - Each tree generated by GP can be evaluated using an indicator showing how good it is in building high quality solutions to the sample problem instances, such as, mean quality of solutions over the sample instances

Q&A

Thank you.

Ender Özcan

ender.ozcan@nottingham.ac.uk

University of Nottingham, School of Computer Science
Jubilee Campus, Wollaton Road, Nottingham
NG8 1BB, UK
http://www.cs.nott.ac.uk/~pszeo