The Programming Game

An Alternative to GP for Expression Search

DAASE/COW Open Workshop
Tuesday 23rd April 2013

David R White
SICSA Research Fellow · University of Glasgow

A Confession

Observation

"There have been exciting recent breakthroughs in the use of genetic programming to re-design aspects of systems to fix bugs, to migrate to new platforms and languages and to optimise non-functional properties."

Harman et al.,
Dynamic Adaptive Search Based Software Engineering,
ESEM 2012.

Genetic Programming as a Hyper-Heuristic

School of Computer Science and Information Technology
University of Nottingham
Jubilee Campus
NOTTINGHAM NG8 1BB, UK

Computer Science Technical Rep

A Genetic Programming Hyper-H

Edmund Burke, I
Graham Kendall an

First released: 2008

Exploring Hyper-heuristic Methodologies with Genetic Programming

Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward*

Abstract Hyper-heuristics represent a novel search methodology that is motivated by the goal of automating the process of selecting or combining simpler heuristics in order to solve hard computational search problems. An extension of the original hyper-heuristic idea is to generate new heuristics which are not currently known. These approaches operate on a search space of heuristics rather than directly on a search space of solutions to the underlying problem which is the case with most meta-heuristics implementations. In the majority of hyper-heuristic studies so far, a framework is provided with a set of human de
erature, and with good measures of performan
proach aims to *generate* new heuristics from a
The purpose of this chapter is to discuss this
Genetic Programming is the most widely used
is presented including the steps needed to ap
tive case studies, a literature review of relate
issues. Our aim is to convey the exciting pote
automating the heuristic design process.

942 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 14, NO. 6, DECEMBER 2010

A Genetic Programming Hyper-Heuristic Approach for Evolving 2-D Strip Packing Heuristics

Edmund K. Burke, *Member, IEEE,* Matthew Hyde, *Member, IEEE,* Graham Kendall, *Member, IEEE,* and John Woodward

***Abstract*—We present a genetic programming (GP) system to evolve reusable heuristics for the 2-D strip packing problem. The evolved heuristics are constructive, and decide both which piece to pack next and where to place that piece, given the current partial solution. This paper contributes to a growing research area that represents a paradigm shift in search methodologies. Instead of using evolutionary computation to search a space of solutions, we employ it to search a space of *heuristics* for the problem. A key motivation is to investigate methods to automate the heuristic design process. It has been stated in the literature that humans are very good at identifying good building blocks for solution methods. However, the task of intelligently searching through all of the potential combinations of these components is better suited to a computer. With such tools at their disposal, heuristic designers are then free to commit more of their time to the creative process of determining good components, while**

measured as the distance from the base of the sheet to the piece edge furthest from the base. This problem is known to be NP (non-deterministic polynomial-time) hard [2], and has many industrial applications as there are many situations where a series of rectangles of different sizes must be cut from a sheet of material (for example, glass or metal) while minimizing waste.

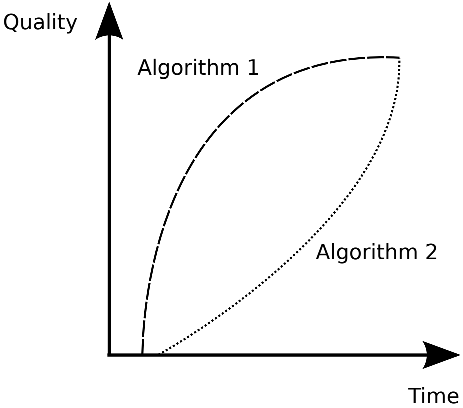
Indeed, many industrial problems are not limited to just rectangles (for example textiles, leather, etc.) and this presents another challenging problem [3]. There are many other types of cutting and packing problems in one, two and three dimensions. A typology of these problems is presented by Wascher *et al.* in [4]. As well as their dimensionality, the problems are

The Demands of DAASE

- Dynamic, online, run-time optimisation.
- Continuous adaptation.

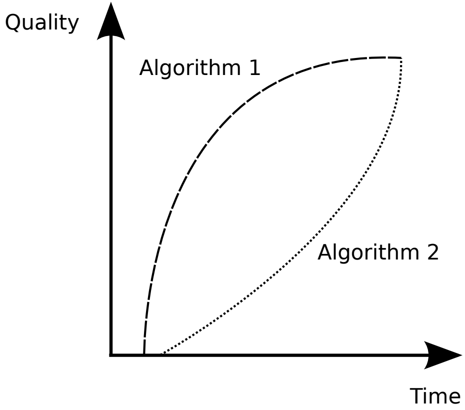
Anytime Algorithms

Evolutionary Algorithms are often viewed as anytime algorithms:



Anytime Algorithms

Evolutionary Algorithms are often viewed as anytime algorithms:



. . . but I would argue that they are *somewhat imperfect* anytime algorithms. Especially GP.

Why GP is not so Anytime

- Bloat
- Parameter Setting
- Difficulty of Allocating Computational Budget
- Notions of Progress and Coverage

Why GP is not so Anytime

- Bloat
- Parameter Setting
- Difficulty of Allocating Computational Budget
- Notions of Progress and Coverage

How well do we understand a GP search? How can we hope to control it? ("Insight")

Steal from Artificial Intelligence Research

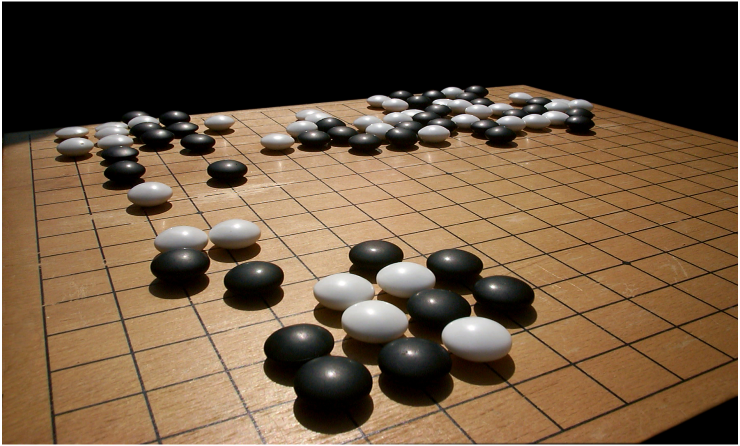
DAASE and Genetic Programming

An Alternative Program Search Method

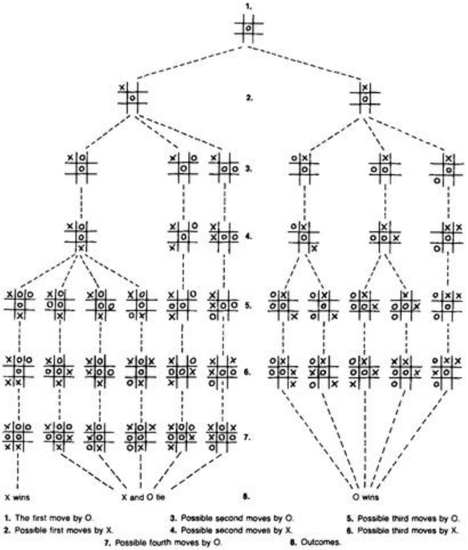
Two Experiments

Wrap-Up

Monte Carlo Tree Search

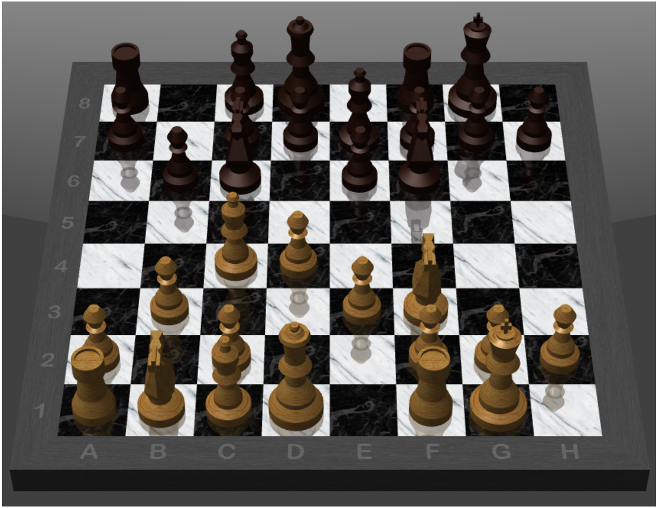


Game Tree



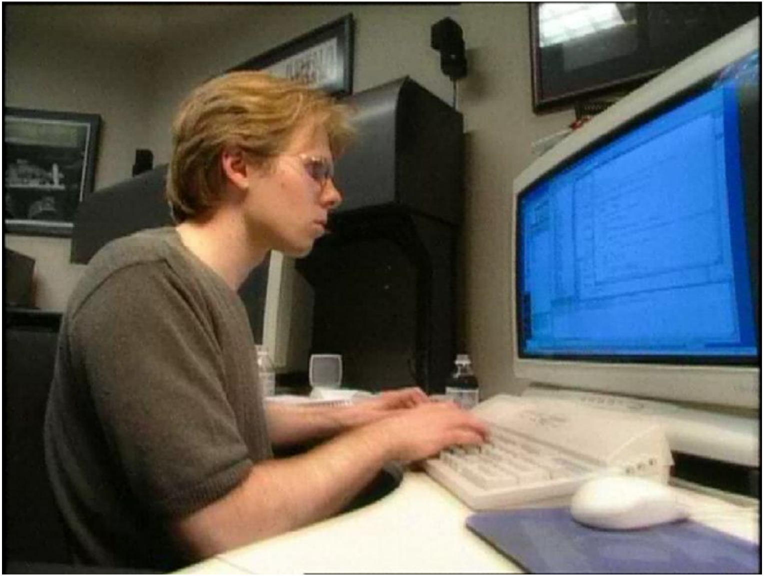
1.
2.
3.
4.
5.
6.
7.
8.
X wins
X and O tie
O wins
1. The first move by O.
2. Possible first moves by X.
3. Possible second moves by O.
4. Possible second moves by X.
5. Possible third moves by O.
6. Possible third moves by X.
7. Possible fourth moves by O.
8. Outcomes.

Sampling



29 possible moves for White here.

Programming is a One-Player Game



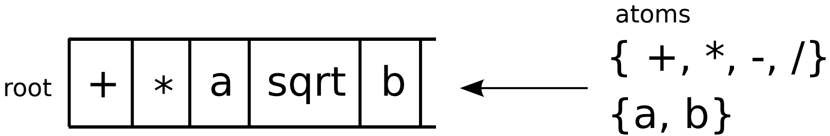
Tristan Cazenave's Work

Nested Monte-Carlo Expression Discovery, Cazenave, ECAI 2010.

Monte-Carlo Expression Discovery, International Journal on Artificial Intelligence Tools, Cazenave, 22 (1) 2013.

A Stack Machine

Stack using Reverse Polish notation.



Each atom added is a move through the game tree.

Building the Game Tree

1. Selection
2. Expansion
3. Sampling
4. Update

Python Implementation

```
def uct(max_evals, terms, nonterms, ucb_constant, max_nodes, scoref):
  root = TreeNode(None, terms, nonterms, None, ucb_constant, 1, 0, max_nodes)
  for i in xrange (max_evals):
    if root.explored:
      break
    stack = ExpressionStack(max_nodes)
    leaf = tree_policy(root, terms, nonterms, ucb_constant, stack, max_nodes)
    score = playout(stack, terms, nonterms, scoref)
    backup(leaf, score)
  return root

def tree_policy(node, terms, nonterms, ucb_constant, stack, max_nodes):
  while stack.leaves > 0:
    if not node.all_atoms_tried():
      new_child = expand(node, stack, terms, nonterms, ucb_constant, max_nodes)
      stack.push(new_child.node_atom)
      if stack.leaves == 0:
        new_child.explored = True
        new_child.possible_atoms = []
      return new_child
    else:
      node = best_child(node)
      stack.push(node.node_atom)
  return node
```

Python Implementation (Cont.)

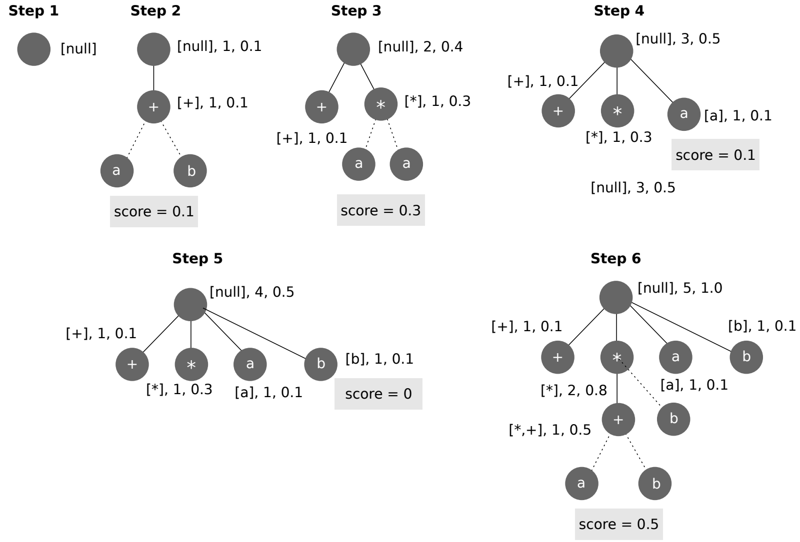
```python
def expand(node,stack,terms,nonterms,ucb_constant,expr_size,max_nodes):
  atom = node.next_atom()
  e_leaves = stack.leaves + atom.arity - 1
  e_size = len(stack.expression)+1
  c = TreeNode(atom,terms,nonterms,node,ucb_constant,e_leaves,e_size,max_nodes)
  node.add_child(c)
  return c

def backup(node,score):
  while node is not None:
    node.visits = node.visits + 1
    node.sum_scores = node.sum_scores + score
    if node.all_atoms_tried():
      done = True
      for c in node.children:
        done = done and c.explored
      node.explored = done
    node = node.parent
```

A Simple Example

Symbolic regression with the language $\{+, *, \mathsf{a}, \mathsf{b}\}$.

Example Game Tree Construction



Balancing Exploration and Exploitation

Choose child with highest UCT score.

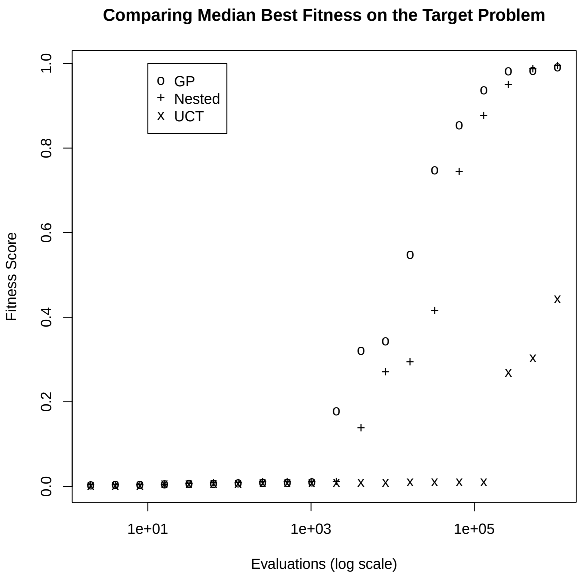
$$\frac{S_c}{n_c} + K\sqrt{\frac{2\ ln\ n_c}{n_p}}$$

S_c total score for playouts involving this node.
n_c number of visits to this node.
n_p number of visits to the parent of this node.
K constant

The Target Problem

Find an equation using the numbers $\{1 \ldots 10\}$ exactly once and the arithmetic operators $+$,-,$/$,* so that the result is as close to 737 as possible.

Target Problem: Results

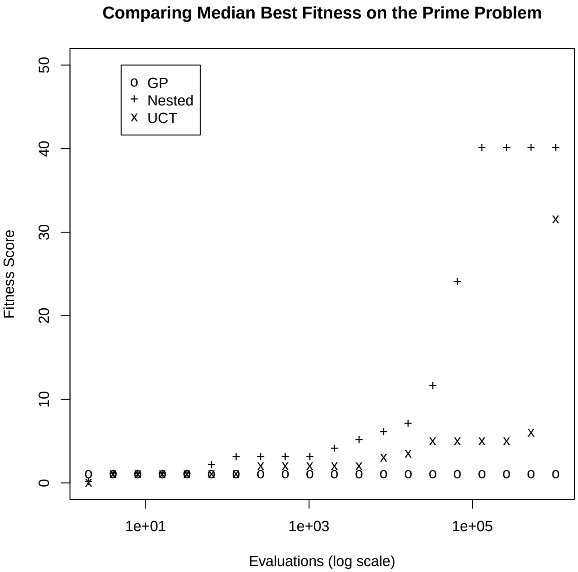


Prime Generation

Find an equation that generates unique prime numbers when fed with the natural numbers as input.

The function set is $+$,-,*,$/$ and the terminal set is $\{1 \ldots 10\}$ and all the prime numbers under 100.

Prime Problem: Results



Advantages of MCTS

Concise Solutions.

Game Tree is Human-Readable.

Parallelisation.

Relevant Previous Work

Real-time Games
UCT for Tactical Assault Planning in Real-Time Strategy Games, Balla and Fern, ICAI 2009.

Scheduling Problems
Monte-Carlo Tree Search in Production Management Problems, Chaslot et al., Benelux Conference on AI, 2006.
(includes a comparison to EAs)

Feature Selection
Feature Selection as a One-Player Game, Gaudel and Sebag, ML 2010.

What next?

A better paper!

Further adapting MCTS for program search.
e.g. use of grammars to introduce typing.

Application to *challenging* problems.

Acknowledgements


Juan E. Tapiador

Tristan Cazenave


Further Reading

Highly recommended:

A Survey of Monte Carlo Tree Search Methods, Browne et al., IEEE Trans. on Computational Intelligence and AI in Games, 2012.