Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 309 (2014) 95–107

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Algebraic Approach to Population-Based Evolutionary Algorithm Generation

Yu-Jun Zheng[1,2], Bei Zhang, Min-Xia Zhang

College of Computer Science & Technology
Zhejiang University of Technology
Hangzhou, China

Abstract

Evolutionary algorithms (EAs) are popular in solving a diversity of problems, but current algorithm design approaches typically require formulating an algorithmic structure for each individual problem. The paper presents an algebraic framework for high-level specification of general-purpose metaheuristic methods, which cover a wide range of population-based EAs. Based on specification composition and refinement, the framework support mechanical program generation for concrete problem solving. We illustrate the applications of the framework in two typical optimization problems, which show that the proposed approach can achieve a high level of abstraction and mechanization without losing performance.

Keywords: Algebraic specifications, evolutionary algorithms (EAs), code generation, generic types.

1 Introduction

In the areas of science and engineering, a very wide class of problems are found to be computationally intractable by traditional deterministic algorithmic methods. In recent two decades, evolutionary algorithms (EAs), including genetic algorithm (GA) [5], evolutionary strategy (ES) [1], evolutionary programming (EP) [4], swarm intelligence methods [2], etc., have received great interest and achieved great success in solving such problems. In general, EAs are stochastic search methods that are mainly inspired by biological evolution and that support a parallel trial and error of a population of different solutions. They do not guarantee finding the exact optimal solution in a single simulation run, but in most cases they are capable of finding acceptable solutions in a reasonable computational time.

Rigorously speaking, EAs are not real "algorithms"; Instead they are "metaheuristics" which are high-level strategies for designing heuristics procedures for

[1] This work was supported by National Natural Science Foundation of China under Grant No. 61020106009, 61105073 and 61272075.

[2] Email:zhengyujun@acm.org

http://dx.doi.org/10.1016/j.entcs.2014.12.008

solving different problems, e.g., the knapsack problem, the traveling salesman problem (TSP), the vehicle routing problem (VRP), etc. Nevertheless, due to the inherent complexity and diversity of the problems, the application of an EA typically requires formulating different algorithmic structures for different problems, which leads to poor reusability, maintainability, and extensibility.

The community has advocated the use of algebraic specification and program transformation technologies to improve software productivity and quality for many years [7], and a number of development tools and environments have been proposed for this purpose [3]. However, most of those methods and tools are used only in limited areas such as real-time and embedded systems. Moreover, few works have been done on the implementation of general-purpose EAs for a wide range of problems which are often encountered in a variety of real-world applications.

In order to minimize the user efforts and ensure product quality in algorithmic program development, we have studied the algebraic approach to transform abstract specifications to concrete programs based on data type refinement and functional refinement [10,11,15], which have been successfully applied to a set of classical algorithm design methods including dynamic programming, greedy, and branch-and-bound [13,17,18], and some heuristic methods such as tabu search [14]. The approach has been used in a number of industrial software projects and has demonstrated its advantage in software quality and productivity.

In this paper, we present a high-level but practical framework for mechanical implementation of population-based EAs for complex problem solving. The framework supports algebraic specification of metaheuristic methods and optimization problems, and mechanical generation of algorithmic programs for concrete problems. Using algebraic specification composition and refinement techniques, our approach achieves a high level of abstraction and mechanization without losing performance in detailed implementation.

The remainder of the paper is structured as follows: Section 2 introduces the preliminaries of algebraic data types and specifications, Section 3 presents our algebraic framework of metaheuristic EAs, including specifications of typical EAs such as GA, PSO, and biogeography-based optimization (BBO) [9]; Section 4 presents our algebraic approach to concrete program generation, and finally Section 5 concludes with discussion.

2 Preliminaries of Basic Concepts

The basic concepts and constructions used in our approach are based on algebraic data types and specifications [8]. Formally, a specification is the finite presentation of a theory with the signature describing objects, operations, and properties:

- A signature $\Sigma = \langle S, \Omega \rangle$ consists of a set S of sorts and operations Ω over S;
- A specification $SP = \langle S, \Omega, A \rangle$ consists of a signature $\Sigma = \langle S, \Omega \rangle$ and a set of Σ-sentences A called axioms;
- A specification morphism $F : \langle S_1, \Omega_1, A_1 \rangle \to \langle S_2, \Omega_2, A_2 \rangle$ maps S_1 to S_2 and Ω_1

to Ω_2 such that for each $a \in A_1$ we have $F(a) \in A_2$.

The following presents the algebraic specifications of basic data types *Boolean* and *Real* respectively.

type	*Boolean* (abbr. $\mathbb{B}$)
constants	$true, false :\rightarrow Boolean$
operations	$\neg : Boolean \rightarrow Boolean$
	$\wedge, \vee : Boolean \times Boolean \rightarrow Boolean$
axioms	$\neg true = false; \neg false = true$
	$(b, b_1, b_2 : Boolean)\ b_1 \wedge b_2 = b_2 \wedge b_1; b_1 \vee b_2 = b_2 \vee b_1$
	$false \wedge b = false; true \vee b = true; true \wedge b = b; false \vee b = b$
	$b \wedge (b_1 \wedge b_2) = (b \wedge b_1) \wedge b_2; b \vee (b_1 \vee b_2) = (b \vee b_1) \vee b_2$
	$b \wedge (b_1 \vee b_2) = (b \wedge b_1) \vee (b \wedge b_2); b \vee (b_1 \wedge b_2) = (b \wedge b_1) \vee (b \wedge b_2)$

type	*Real* (abbr. $\mathbb{R}$)
imports	*Boolean*
constants	$0, 1, \infty :\rightarrow Real$
operations	$+, -, \times, / : Real \times Real \rightarrow Real$
axioms	$(a, a_1, a_2 : Real)\ a_1 + a_2 = a_2 + a_1; a_1 \times a_2 = a_2 \times a_1$
	$0 + a = a; 1 \times a = a; 0 \times a = 0;$
	$a \neq \infty \Rightarrow a/\infty = 0; a \neq 0 \Rightarrow a \times \infty = \infty$
	$a + (a_1 + a_2) = (a + a_1) + a_2; a \times (a_1 \times a_2) = (a \times a_1) \times a_2$
	$a \times (a_1 + a_2) = (a \times a_1) + (a \times a_2); a_1 \times (a/a_1) = a$
	...

A parameterized specification has formal parameters that are themselves specifications, the binding of actual values to which is accomplished by specification morphisms. The incremental development of specifications involves developing simple specifications and then importing them into more complex ones. For example, the following gives the algebraic specification of data structure *Set*, in which type

parameter T denotes the abstract type of set elements.

type	$Set\langle T\rangle$
imports	$Boolean, Nat$
sorts	T
constants	$\emptyset :\rightarrow Set\langle T\rangle$
operations	$\{\} : T \rightarrow Set\langle T\rangle; \| \| : Set\langle T\rangle \rightarrow Nat$
	$\in: T \times Set\langle T\rangle \rightarrow Boolean$
	$\subset, \subseteq: Set\langle T\rangle \times Set\langle T\rangle \rightarrow Boolean$
	$\cup, \cap, \setminus : Set\langle T\rangle \times Set\langle T\rangle \rightarrow Set\langle T\rangle$
axioms	$(u, v : T;\ U, V : Set\langle T\rangle)\ \|\emptyset\| = 0; \|\{u\}\| = 1$
	$u \in \emptyset = false; u \in \{u\} = true; u \in U \cup \{u\} = true$
	$\emptyset \subset U = true; u \in U \wedge U \subseteq V = u \in V$
	$U \cup V = V \cup U; U \cap V = V \cap U; \emptyset \cup U = U; \emptyset \cap U = \emptyset$
	$\neg(u \in U) \Rightarrow U \setminus \{u\} = U$
...	

According to the theory of algebraic data types, a specification defines a problem by specifying a domain of problem inputs and the notion of what constitutes a solution to a given input [6].

type	$Problem\langle D, Z\rangle$
imports	$\mathbb{B}$
sorts	D, Z
operations	$I : D \rightarrow \mathbb{B}; O : D \times Z \rightarrow \mathbb{B}$

where the input condition $I(x)$ constrains the input domain D and the output condition $O(x, z)$ describes the condition under which the output domain value $z \in Z$ is feasible solution with respect to an input $x \in D$.

In particular, an optimization problem, which is typically defined on a partial ordered set ($Poset$), can be treated as an extension of *problem* as follows:

type	$OptProblem\langle D, Z\rangle$
refines	$Problem\langle D, Z\rangle$
imports	$\mathbb{B}, \mathbb{R}, Set$
sorts	D, Z
operations	$\xi : D \to Set\langle Z\rangle; \quad c : D \times Z \to \mathbb{B}; \quad f : Z \to \mathbb{R}$
axioms	$(d_1, d_2 : D) \quad d_1 \leq_D d_2 \Rightarrow \xi(d_1) \subseteq \xi(d_2)$

where ξ is the generative function for generating the solution space, c is the constraint function defining the feasibility, f is the objective function for evaluating the optimality, and $\leq_D$ is the ordering relation on D.

3 Algorithm Framework

3.1 A General Framework for Population-Based EAs

In its search procedure, an EA typically evolves a population of candidate solutions to a given problem, using operators inspired by natural or biological evolution. We define a very high-level specification of a population-Based EA as follows, which consists of the signatures of a problem of $OptProblem\langle D, Z\rangle$ and a set of abstract functions:

type	$Alg\langle D, Z\rangle$
imports	$\mathbb{B}, \mathbb{R}, Set, List, OptProblem$
sorts	$D, Z;$
	$P : OptProblem\langle D, Z\rangle; \quad POP : Set\langle Z\rangle;$
	$OP : List\langle Z \times Z \to Z\rangle$
operations	$init : D \times \mathbb{N} \to Set\langle Z\rangle; \quad evol :\to Set\langle Z\rangle;$
	$solve : D \times \mathbb{N} \to Z; \quad best :\to Z; \quad tune :\to$
axioms	$(z : Z) \quad z \in POP \Rightarrow P.f(best()) \leq P.f(z)$

In the above specification, *init* is used for initializing a set of solutions for a given problem input d, *evol* performs an iteration of evolution of the algorithm, *solve* runs a given number of iterations to produce a result solution, *best* returns the optimal solution found so far, and *tune* adjusts related control parameters after each iteration; POP maintains a population of solutions, and OP is a set of evolutionary operators of the algorithm.

Among the abstract functions, the default implementation of *evol* applies each evolutionary operator to the solutions in POP one by one:

```
def fun evol() : Set⟨Z⟩
begin
  let POP1 = new Set⟨Z⟩();
  for each z ∈ POP do
    POP1 ← POP1 ∪ {z};
  for each o ∈ OP do
    for each z ∈ POP1 do
      z ← o(z);
  best();
  tune();
  return POP1;
end
```

And the default implementation of *solve* evolves the population for a given number of generations:

```
def fun solve(d : D; size, iters : ℕ) : Z
begin
  POP ← init(d, size);
  for k = 1 to iters do
    POP ← evol();
  return best();
end
```

3.2 Specifications of Typical EAs

By specifying different evolutionary operators and their application procedures, the top-level specification *Alg* can be refined to different EA specifications. GA is such a typical EA that uses two well-known evolutionary operators: crossover and mutation, and the specification of GA can be easily defined based on *Alg*:

type	$GA\langle D, Z\rangle$
refines	Alg
imports	$\mathbb{B}, \mathbb{R}, Set, List, OptProblem$
operations	$mutate : Z \to Z; \quad crossover : Z \times Z \to Z \times Z;$
	$select : Set\langle Z\rangle \to Z$

Note that the crossover operator of GA takes two parent solutions and produces two child solutions, and its signature does not meet that defined in specification *Alg*. Thereby, we redefine its *evol* operator by overriding the default implementation in *Alg*:

Beside overriding a default implementation, another common way to tackle with variation of operation signatures is wrapping. For example, the BBO algorithm uses a migration operator that migrate features from a probably high quality solution to a low quality one. The following specification defines a $selMigrate$ to perform

override fun $evol() : Set\langle Z\rangle$
begin
  let $POP1 =$ **new** $Set\langle Z\rangle()$;
  while $|POP1| < |POP|$ **do**
    let $z_1 = select(POP), z_2 = select(POP)$;
    $POP1 \leftarrow POP1 \cup \{corssover(z_1, z_2)\}$;
    for each $z \in POP1$ **do**
      $z \leftarrow mutate(z)$;
  $best()$;
  return POP1;
end

such an operation, and lets OP contains the other three functions that satisfy the signatures defined in Alg:

type	$BBO\langle D, Z\rangle$
refines	Alg
imports	$\mathbb{B}, \mathbb{R}, Set, List, OptProblem$
sorts	$OP = \{migrate, mutate\}$
operations	$migrate : Z \to Z; \quad mutate : Z \to Z;$
	$selMigrate : Z \times Z \to Z; \quad select : Set\langle Z\rangle \to Z;$

And the $migrate$ function encapsulates $selMigrate$ in its default implementation as follows:

def fun $migrate(z : Z) : Z$
begin
  let $z_1 = select(POP)$;
  return $selMigrate(z, z_1)$;
end

The following presents the algebraic specification of the PSO algorithm and its standard implementations of main operations, where $Particle$ is a data type extends the basic definition of problem solution.

type $Particle\langle Z\rangle$

sorts $z : Z; \quad pb : Z; \quad v : Vector$

type	$PSO\langle D, Z\rangle$
refines	Alg
imports	$\mathbb{B}, \mathbb{R}, Set, List, OptProblem$
sorts	$POP = List\langle Particle\langle Z\rangle\rangle; \quad PB : List\langle Z\rangle;$
	$gb : Z; \quad w, c_1, c_2 : \mathbb{R};$
	$OP = \{learn, move\}$
operations	$learn : Z \to Z; \quad move : Z \times Vector \to Z;$

def fun $learn(z : Z) : Z$
begin
  $z.V \leftarrow w * (z.V + rand() * c_1 * (z.pb - z.z) + rand() * c_2 * (gb - z.z));$
  $z.z \leftarrow move(z.z, z.V);$
  return z;
end

4 Program Generation for Concrete Problem Solving

Given a concrete problem specification of $OptProblem\langle D, Z\rangle$, the process for generating algorithmic program from the algebraic specification can be divided into the following steps:

(i) Construct the refinement morphisms from type parameters in the algebraic specification to their concrete types;

(ii) For each abstract function in the specification, if no user-defined implementation is provided, then use its default implementation in the framework;

(iii) Construct the refinement morphisms from abstract functions to their implementations;

(iv) Generate the concrete algorithmic program by colimit computation on generic specification and its refinements [16];

(v) Transform the abstract algorithmic program to one or more executable programs [12].

Next we illustrate the process using two different problems.

4.1 Algorithms for Integer Programming

Integer programming problem is a class of mathematical optimization problems where the decision variables are restricted to integer values. Based on our algebraic

framework, an integer programming problem can be specified as:

type	$IPProblem$
refines	$OptProblem\langle Vector\langle\mathbb{Z}\rangle \to \mathbb{R}, Vector\langle\mathbb{Z}\rangle\rangle$
imports	$\mathbb{B}, \mathbb{Z}, \mathbb{R}, Vector, Set$
sorts	$obj : Vector\langle\mathbb{Z}\rangle \to \mathbb{R}$
	$VL, VU : Vector\langle\mathbb{Z}\rangle$
refinement with	$c\ \ d\ \ z = \forall i : (0 \le i < \lvert d\rvert) : VL[i] \le d[i] \le VU[i];$
	$f = obj$

To apply the GA specification to the problem, we respectively construct the morphisms from the *crossover* and *mutate* operations to the following two implementations:

fun $crossover(z_1, z_2 : Z) : Z \times Z$
begin
  let $p = rand(1, |z| - 1)$;
  let $z'_1 = z_1[0..p] \# z_2[p + 1..]$;
  let $z'_2 = z_2[0..p] \# z_1[p + 1..]$;
  return (z'_1, z'_2);
end

fun $mutate(z : Z) : Z$
begin
  if $rand() < mr$ //mutation rate
    let $p = rand(0, |z|)$;
    $z[p] \leftarrow round(VL[p] + rand() * VU[p])$;
  return z;
end

Based on categorical computation, we directly work out the following GA program for solving an integer programming problem:

Algorithm 1 ***GA***

$POP : Set\langle Vector\langle\mathbb{Z}\rangle\rangle$; $mr : \mathbb{R}$
fun $main(d : IPProblem; size, iters : \mathbb{N})$
begin
  $POP \leftarrow init(d, size)$;
  for $k = 1$ ***to*** $iters$ ***do***
    $POP \leftarrow evol()$;
  return $best()$;
end
fun $init(d : IPProblem; size : \mathbb{N}) : Set\langle Vector\langle\mathbb{Z}\rangle\rangle$
begin
  $POP \leftarrow newSet\langle Vector\langle\mathbb{Z}\rangle\rangle()$;
  for $k = 1$ ***to*** $size$ ***do***
    $POP \leftarrow POP \cup \{rand(VL, VU)\}$;
  return POP;
end

fun $evol() : Set\langle Vector\langle\mathbb{Z}\rangle\rangle$
begin
  let $POP1 =$ ***new*** $Set\langle Vector\langle\mathbb{Z}\rangle\rangle()$;
  while $|POP1| < |POP|$ ***do***
    let $z_1 = select(POP), z_2 = select(POP)$;
    let $p = rand(1, |z| - 1)$;
    let $z'_1 = z_1[0..p] \# z_2[p + 1..]$;
    let $z'_2 = z_2[0..p] \# z_1[p + 1..]$;
    $POP1 \leftarrow POP1 \cup \{z'_1, z'_2)\}$;
    for each $z \in POP1$ ***do***
    if $rand() < mr$
      $p \leftarrow rand(0, |z|)$;
      $z[p] \leftarrow VL[p] + rand() * VU[p]$;
  $best()$;
  $tune()$;
  return $POP1$;
end

If we use PSO to solve the integer programming problem, we can keep the default implementation of *learn* and simply construct the morphisms from *move* to the following implementation, and thereby obtain a PSO program for integer programming (the detailed code is omitted here).

fun $move(z : Vector\langle\mathbb{Z}\rangle, v : Vector) : Vector\langle\mathbb{Z}\rangle$
begin
  for $k = 0$ **to** $|z| - 1$ **do**
    $z[k] \leftarrow round(z[k] + v[k])$;
  return z;
end

4.2 Algorithms for the Traveling Salesman Problem

The traveling salesman problem (TSP) is a well-known combinatorial optimization problem, which takes a weighted graph as the input and a Hamiltonian cycle of the graph with minimum weight as a solution. A weighted graph can be represented by a matrix of real numbers and a Hamiltonian cycle can be represented by a permutation of nodes, and thus the TSP can be specified as:

type	TSP
refines	$OptProblem\langle Matrix, Perm\rangle$
imports	$\mathbb{B}, \mathbb{Z}, \mathbb{R}, Matrix, Set, Perm$
refinement with ξ	$d = allperms(\lvert a\rvert);$
	$c\ d\ \ z = (\lvert z\rvert = \lvert m\rvert);$
	$fz = \sum_{i=0}^{\lvert z\rvert-2} m[i, i+1] + m[\lvert z\rvert - 1, 0]$

If we use BBO to solve the TSP, the *migration* operation can be used for migrating a subsequence of the emigrating solution to the current one, meanwhile keeping the solution a permutation. Thus the target implementation can be respectively defined as:

fun $selMigrate(z, z_1 : Perm) : Perm$
begin
  let $p_1 = rand(0, |z| - 2)$; $p_2 = rand(p1, |z| - 1)$;
  for $k = p_1$ **to** p_2 **do**
    let $p = indexof(z_1[k], z)$;
    $(z[k], z[p]) \leftarrow (z[p], z[k])$;
  return z;
end

And the *mutate* operation can be simply defined as swapping two randomly chosen nodes in the permutation:

fun $mutate(z : Perm) : Perm$
begin
  let $p_1 = rand(0, |z| - 2)$; $p_2 = rand(p1, |z| - 1)$;
  $(z[p_1], z[p_2]) \leftarrow (z[p_2], z[p_1])$;
  return z;
end

The result BBO program for solving the TSP is as follows:

Algorithm 2 ***BBO***

$POP : Set\langle Perm\rangle$; $ems, ims, ms : List\langle\mathbb{R}\rangle$;//*emigration, immigration, and mutation rates*

fun $main(d : TSP; size, iters : \mathbb{N})$
begin
  $POP \leftarrow init(d, size)$;
  for $k = 1$ ***to*** $iters$ ***do***
    $POP \leftarrow evol()$;
  return *best()*;
end
fun $init(d : TSP; size : \mathbb{N}) : Set\langle Perm\rangle$
begin
  $POP \leftarrow newSet\langle Perm\rangle()$;
  for $k = 1$ ***to*** $size$ ***do***
    $POP \leftarrow POP \cup \{randperm(|d|)\}$;
  return *POP*;
end

fun $evol() : Set\langle Perm\rangle$
begin
  let $POP1 =$ ***new*** $Set\langle Perm\rangle()$;
  for each $z \in POP$ ***do***
    if $rand \leq ims(z)$
      let $z_1 = select(POP)$;
      let $p_1 = rand(0, |z| - 2)$; $p_2 = rand(p1, |z| - 1)$;
      for $k = p_1$ ***to*** p_2 ***do***
      let $p = indexof(z_1[k], z)$;
      $(z[k], z[p]) \leftarrow (z[p], z[k])$;
    $POP1 \leftarrow POP1 \cup \{z)\}$;
    for each $z \in POP1$ ***do***
    if $rand() < ms(z)$
      $p_1 \leftarrow rand(0, |z| - 2)$; $p_2 \leftarrow rand(p1, |z| - 1)$;
      $(z[p_1], z[p_2]) \leftarrow (z[p_2], z[p_1])$;
  best();
  tune();
  return *POP1*;
end

5 Conclusion

The paper presents an algebraic framework for high-level specification of general-purpose metaheuristic methods and mechanical generation of algorithmic programs for concrete problem solving. Our algebraic approach is mathematically abstract and computationally efficient. Currently we are extending the approach to support

algorithms for multiobjective optimization problems.

References

[1] Beyer, H.-G. and H.-P. Schwefel, *Evolution strategies -- a comprehensive introduction*, Natural Comput. **1** (2002), pp. 3–52.

[2] Bonabeau, E., M. Dorigo and G. Theraulaz, **4**, Oxford University Press, New York, 1999.

[3] Claszen, I., H. Ehrig and D. Wolz, "Algebraic specification techniques and tools for software development: the ACT approach," World Scientific, 1993.

[4] Fogel, L. J., A. J. Owens and M. J. Walsh, "Artificial intelligence through simulated evolution," John Wiley & Sons Chichester, 1967.

[5] Holland, J. H., "Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence," MIT Press, Cambridge, MA, USA, 1992.

[6] Lowry, M., "Algorithm synthesis through problem reformulation," Ph.D. thesis, Stanford University (1989).

[7] Partsch, H., "Specification and transformation of programs: a formal approach to software development," Springer, 1990.

[8] Sannella, D. and A. Tarlecki, *Essential concepts of algebraic specification and program development*, Formal Aspects of Computing **9** (1997), pp. 229–269.

[9] Simon, D., *Biogeography-based optimization*, IEEE Trans. Evol. Comput. **12** (2008), pp. 702–713.

[10] Xue, J., *A unified approach for developing efficient algorithmic programs*, Journal of Computer Science and Technology **12** (1997), pp. 314–329.

[11] Xue, J., *Formal derivation of graph algorithmic programs using partition-and-recur*, Journal of Computer Science and Technology **13** (1998), pp. 553–561.

[12] Xue, J., *Par method and its supporting platform*, in: *Proc. 1st Intl Workshop of Asian Working Conf. Verified Software*, Macao, China, 2006, pp. 11–20.

[13] Zheng, Y., H. Shi and J. Xue, *Toward a unified implementation for dynamic programming*, High Technology Letters **12** (2006), pp. 31–34.

[14] Zheng, Y., H. Shi and J. Xue, *An algebraic approach to mechanical tabu search algorithm generation*, in: *IEEE International Conference on Progress in Informatics and Computing*, 2010, pp. 988–992.

[15] Zheng, Y. and J. Xue, *A problem reduction based approach to discrete optimization algorithm design*, Computing **88** (2010), pp. 31–54.

[16] Zheng, Y., J. Xue and W. Liu, *Object-oriented specification composition and refinement via category theoretic computations*, in: J.-Y. Cai, S. Cooper and A. Li, editors, *Theory and Applications of Models of Computation*, Lecture Notes in Computer Science **3959**, Springer Berlin Heidelberg, 2006 pp. 601–610.

[17] Zheng, Y., J. Xue and H. Shi, *A category theoretic approach to search algorithms: Towards a unified implementation for branch-and-bound and backtracking*, in: *4th International Conference on Computer Science Education*, 2009, pp. 845–850.

[18] Zheng, Y., J. Xue and Z. Zuo, *Toward an automatic approach to greedy algorithms*, in: X. Deng, J. Hopcroft and J. Xue, editors, *Frontiers in Algorithmics*, Lecture Notes in Computer Science **5598**, Springer Berlin Heidelberg, 2009 pp. 302–313.