Automated Search for Heuristic Functions*

Student: Pavel Cejnar
Supervisor: Roman Barták

Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, Praha 1, Czech Republic
`pavel.cejnar@st.cuni.cz`

Abstract. We present a metaheuristic algorithm for creating heuristic functions for CSP, especially SAT, with as low human interaction as possible. We use the concept of genetic programming to evolve local search heuristic functions encoded as a list of RAM model-like computer instructions. However, we've extended the RAM model language to reflect the features of genetic programming and the random mixing of code instructions. We present possible implementation of genetic operations upon these population members.

1 Introduction

If we want to solve a constraint satisfaction problem, we have two possibilities. Either we can search systematically the space of possible solutions (use systematic search algorithms), or we can start with a random point in the space and search randomly using a heuristic with a given "time" limit (use local search algorithms). To solve a very large problem, local search algorithms are usually preferred.

Due to extensive research in local search algorithms, there are many good heuristics solving related problems very efficiently. The more information about the solved problem we have, the more effective heuristic function we can construct. In real life, if we want to construct an effective heuristic function for a CSP problem, we need a help of an expert, who analyses the problem and states essential or at least important facts to find a solution quickly. Recently, there were published papers where such a heuristic function was automatically constructed by a computer (see [FU1], [CB1]). In these papers, a strong human influence on an architecture of such a function is still present.

A heuristic for SAT problem mentioned in [FU1] runs and solves it faster than human designed heuristics. The author started with some predefined primitives chosen in accordance with a human observation identifying expected good (precomputed) features for solving SAT. We can find examples in areas related to CSP, like game theory, where a computer startegy based on an artificial decision making function strongly beats the computer player strategy playing in

* Supported by the Czech Science Foundation under the contract 201/04/1102.

a way, like human player does (in bridge compare [GI1] and [SM1] or in chess). This leads to a question whether it is possible to construct an effective heuristic function only by a computer, a function constructed from very elementary program building blocks not only from predefined higher-level primitives.

To better demonstrate the ideas, we focus on SAT, because many other constraint satisfaction problems can be formulated as SAT instances, this problem is widely and intensively studied and there are many well-known heuristics we can compare with.

In this paper, we present a procedural programming language for the description of heuristic functions for local search algorithms and we show the application of genetic programming concept to these functions to get a metaheuristic algorithm with a capability to create a local search heuristic without a help of an expert.

The structure of the paper is as follows: The second section recapitulates common terms used in the paper, the third section presents the global concept of this metaheuristic algorithm, the fourth section describes the programming language for heuristic functions, the fifth section shows main data structures for problem instances and the sixth section presents genetic operations on the individuals (the code of heuristic functions).

2 Prerequisities

SAT, testing of satisfiability, is an NP-complete decision problem. We have given a set of boolean variables V, and a well-formed formula (CNF) F, consisting of literals (positive or negative variables from V) and connectives $\wedge$ and $\vee$. The question is whether there exists a truth assignment T of variables from V, such that the formula F is logically true. If such an assignment doesn't exist, the formula F is unsatisfiable.

Given a truth assignment T for a CNF formula F, let T' be the truth assignment of F, where the value of variable $v \in V$ is flipped. The **negative gain** of v is the number of clauses that are satisfied in T and that become unsatisfied in T'.

The **Walksat** heuristic picks a random unsatisfied clause C from F and if any variable in this clause has the negative gain equal to zero, then it selects randomly one of these variables and flips it. Otherwise, with probability p it selects a random variable from C and flips it and with probability $(1 - p)$ it randomly selects one of the variables in C with the lowest negative gain and flips it. This repeats until it finds the satisfying truth assignment or until it reaches a given number of steps (cutoff limit).

3 Architecture of the metaheuristic algorithm

The structure of local search algorithm in Figure 1 is common to many SAT local search procedures. The new truth assignment selection heuristic is executed in

```
T=random truth assignment;
for(j=0;j<cutoff;j++) {
    if (T satisfies formula F) return T;
    T=make_step(T);
}
return NOT_FOUND;
```

Fig. 1. The local search algorithm template.

each step until the satisfying truth assignment is found or until the step counter reaches the cutoff limit. Many SAT new truth assignment heuristics select one variable according to some rule from current truth assignment and flip only this variable's value. However, we don't want to restrict to this technique only.

To create successful heuristic functions from an initial random set, we need some concept that maintains good features of heuristics in a set and removes the heuristics with poor behavior. For this task we use the concept of genetic programming applied like in Figure 2. The `Initialize` function creates a set

```
Initialize(Population);
for(j=0;j<maxIterations;j++) {
    Pick Parent1 and Parent2 from Population;
    Children=Composition(Parent1,Parent2);
    Mutation(Children);
    Evaluation(Children);
    Insert(Children,Population);
}
```

Fig. 2. Genetic programming algorithm.

(population) of random heuristic functions that are guaranted to return a valid truth assignment after each step of local search algorithm. The `Pick` function selects two heuristic functions from the population. The probability of being selected depends on the previous successibility of the heuristic function (score). The `Composition` function generates `n` children, some of them being slightly changed by the `Mutation` function. All the children are evaluated (scored) and two best children are inserted to the population, where they replace randomly selected members. The probability of being replaced depends on the successibility (score) of the member again.

We want to have individuals built from very elementary blocks, however the heuristic functions might be built from a large number of such elements. This forces us to keep the representation of population members as simple as possible, e.g. to code a heuristic function as an ordered list of instructions. This could keep the composition and mutation operations very simple and thus allows us to speed up the process of evolution of the population. In order to keep the representation simple, we left the idea of using a higher level programming language for description of heuristics. Otherwise, we must encode the representation in a more complex structure, like the program code syntax tree. The execution of such an individual requires complex operations either for the run emulation or for the compilation. However, to be able to simply describe many different instances of SAT problem, that can vary in the number of clauses or the number of variables, we keep the idea of programming language (with direct access to memory), like a RAM model. Coding the individual as a neural network or a set of instructions for Turing machine would make reading variable size input or storing results of temporary operations difficult.

4 Programming language for heuristic functions

The Appendix 1 shows the code for the Walksat heuristic. The first part computes the negative gain for all variables and the second part encodes the selection logic. If we don't require handling ties randomly, we can save instructions.

The RAM model language is promising, however, we've made some changes to increase the probability of creating meaningful heuristic function.

The biggest change is the existence of variable types and type checking. We assume that during the evolution of the population there will appear many randomly created instructions and operands. We've defined three types of variables, integer, domain, and boolean to eliminate the cases of assignment (semantically domain) values out of domain range to (domain) variables and the cases of comparison of boolean variables with constants other than 0 (false) or 1 (true). This also leads to the existence of global memory for values of each type.

In randomly created arithmetic instructions there could often occur constant operands. To reduce this and to promote operations on variable operands, we don't use constant operands and handle them by a new instruction for assigning constant values from a small set of constants to variables (`loadc`). To simplify the code, we've got rid of the accumulator variable and use two-operand arithmetic instructions. We've also removed indirect access operands and replaced them by adding two new instructions for these operations (`load`, `store`). We've added instructions `inc` and `dec` to increase or decrease the current value by one and instruction `rnd`, that loads a random value from 1 to given maximum value to the given variable.

We've also added conditional relative jumps for each of arithmetic comparison $(=, \neq, <, >, \leq, \geq)$ to simplify writing compound boolean expressions.

To have the language more robust we've agreed on some standard handling of error states. We read minValue each time we read from a cell out of the range of limited real memory. We've also agreed to get maxValue as a result of division by zero or an overflow. We also suppose that a jump out of the code means halting the program.

5 Description of an encoded problem instance

We have some special problem related constants, `Variables` of integer type, that contains the number of variables in given SAT instance, `Clauses` of integer type, that contains the number of clauses for given SAT instance (more general, the number of constraints for given problem instance) and the arrays `R[]` of domain type containing the current value of variables, `Limits[]` of integer type containing the number of different variables in a clause, and a two-dimensional array `C[][]` containing structured data. `C[i,j].Variable` of integer type contains the index of variable present in clause `i`. `C[i,j].isPositive` of boolean type contains whether this variable is positive in clause `i`. `C[i,j].isNegative` of boolean type contains whether this variable is negative in clause `i`. In `C[i][]` array there are only variables that are present in clause `i`, either as a positive or negative literal or both (in this case the variable is there only once). An example of an encoded CNF formula is in Appendix 2.

To access these arrays, we've added other variable assignment instructions (`readR`, `readLimits`, `readCvar`, `readCpos`, `readCneg`).

To avoid infinite running heuristics, we've limited the maximum number of instruction steps. After the heuristic function ends the execution or reaches the limit, we will read the array `R[]` to get the (new) truth assignment.

We've tried to make the language as simple as possible and we hope to be able to get appropriate results when evolving a population of heuristic functions on a distributed computer system. However, if we get very poor results, we can add other problem specific read-only arrays with precomputed values mentioned in [FU1] (for example negative gain, history,...) and decrease the maximum number of instruction steps to reflect this fact. We could test whether the previous population size and the number of iterations was inefficient to get as good results as the results with added more problem specific arrays and whether we are able to get the results comparable to [FU1].

6 Genetic operations

The composition operator on a linear sequence of instructions can be very simple. To save as many code from the parents, we will not choose the simplest solution, to cut the code in a random point (on an instruction border) and exchange the parts of the code. We will cut the code in two random points and exchange only the central part of the code. We will repair then the targets of jump instructions in the remaining part of the code by adding a difference in the length of exchanged code to jump targets. For example, if we have two individuals, each like the one in Appendix 1, the composition could exchange instructions 38-40 of the first one with instructions 41-43 of the second one. The exchanged parts have the same length and thus the jump targets in the remaining parts don't need any repair.

The mutation operator will choose from following actions:

1. With the probability P_1 it will change a randomly selected read or written variable or a constant in the operands of instructions. For example instruction `store [i14],i7` would become `store [i3],i7` or `store [i5],i9`.
2. With the probability P_2 it will change a randomly selected jump instruction with the one with different comparison function. Instruction `jumpNE +2,i7,i13` would become `jumpL +2,i7,i13`.
3. With the probability P_3 it will change the target of a randomly selected jump instruction. Instruction `jumpNE +7,i12,i11` would become `jumpNE +1,i12,i11`.
4. With the probability P_4 it will change a randomly selected instruction. Instruction `loadc i3,0` would become `mov b2,b1`.

The exact values of probabilities $\boldsymbol{P}$ are a matter of discussion and we expect to tune them during the runs of the population evolution.

The scoring function of an individual in genetic programming should represent the features we exactly want from an evolved individual. We took an inspiration from the scoring function mentioned in [FU1]. To train the heuristic function to solve 100-var SAT instances, we need a filter for poor heuristics and then for mediocre heuristics. Let's run the heuristic first on a small set of 15-variable satisfiable instances to filter the poor ones. If more than 75% of them are successfully solved, we will run it on a larger set of 50-variable satisfiable instances to filter the mediocre ones. If more than 60% of them are successfully solved, we will run it on a large set of 100-variable satisfiable instances. We construct the scoring function in a way like: $c_1 \times$ (# of 15-var successes) $+ c_2 \times$ (# of successes of 50-var successes) $+ c_3 \times$ (# of 100-var successes) $+$ (# of successfully solved instances of problem per unit of time).

7 Conclusions

We've presented the idea of automated creation of SAT heuristic functions with minimum interaction with a human observer. We hope to be able to find heuristic functions with comparable results to currently used heuristics. We expect strong demands on computer performance and on the time for the evolution of the population. We are prepared to tune the parameters of the metaheuristic algorithm, but we want to avoid the changes that would considerably decrease the expressive power of the programming language and the algorithm.

References

[FU1] Fukunaga, A. S.: Evolving local search heuristics for SAT using genetic programming. GECCO-2004, Part II, volume 3103 of LNCS, 483-494, (2004)

[CB1] Carchrae, T., Beck, J.: Low-knowledge algorithm control. AAAI-04, (2004)

[GI1] Ginsberg, M. L.: GIB: Imperfect information in a computationally challenging game. Journal of Artificial Intelligence Research 14, (2001)

[SM1] Smith S. J. J., Nau D., Throop T.: Computer Bridge: A big win for AI planning. AI Magazine 19, (1998)

Appendix 1

The code of individual computing the negative gain for all variables of a randomly selected clause for the Walksat heuristic:

```
loadc i5,1
add i5,i5
mul i5,i5
mul i5,i5
add i5,i5
rnd i1,Clauses
loadc i2,maxValue
loadc i3,0
loadc i4,0
readLimits i9,i1
loadc i6,1
;for(i6=1;i6<=Limits[i1];i6++)
jumpG +55,i6,i9
readCvar i11,i1,i6
readR d1,i11
inc d1
loadc i7,0
loadc i8,1
;for(i8=1;i8<=Clauses;i8++)
jumpG +35,i8,Clauses
loadc b1,true
loadc b2,true
readLimits i10,i8
loadc i9,1
;for(i9=1;i9<=Limits[i8];i9++)
jumpG +23,i9,i10
readCpos b4,i8,i9
loadc b5,true
;if (C[i8,i9].isPositive)
jumpNE +6,b4,b5
readCvar i12,i8,i9
readR d2,i12
loadc d3,maxValue
;if(R[C[i8,i9].Variable]==
;==DOMAIN_1)
jumpNE +2,d2,d3
loadc b1,false
readCneg b6,i8,i9
;if(C[i8,i9].isNegative)
jumpNE +4,b6,b5
loadc d4,minValue
;if(R[C[i8,i9].Variable]==
;==DOMAIN_0)
jumpNE +2,d2,d4
loadc b1,false
;if(C[i8,i9].Variable==
;==C[i1,i6].Variable)
jumpNE +7,i12,i11
;if(C[i8,i9].isPositive)
jumpNE +3,b4,b5
;if(d1==DOMAIN_1)
jumpNE +2,d1,d3
loadc b2,false
;if(C[i8,i9].isNegative)
jumpNE +3,b6,b5
;if(d1==DOMAIN_0)
jumpNE +2,d1,d4
loadc b2,false
inc i9
jump -22 ;endfor
mov b3,b1
inc b3
jumpNE +3,b3,b5 ;if(!b1)
jumpNE +2,b2,b5 ;if(b2)
inc i7
inci8
jump -34 ;endfor
loadc i13,0
jumpNE +2,i7,i13 ;if(i7==0)
inc i4
jumpL +6,i2,i7 ;if(i2>=i7)
jumpLE +3,i2,i7 ;if(i2>i7)
mov i2,i7
loadc i3,0
jumpNE +2,i2,i7 ;if(i2==i7)
inc i3
mov i14,i5
add i14,i6
store [i14],i7
inc i6
jump -54 ;endfor
```

```
i1  selected broken clause
i2  current best negative gain
i3  number of variables with the negative gain equal
    to the current value of i2
i4  number of variables with the negative gain equal to 0
i5  index of the first variable of the negative gain array
    (decreased by 1)
i7  negative gain for the current variable
i13 zero constant
b1  is the current clause false?
b2  will the current clause become false when the current
    variable is flipped?
b5  true constant
d1  flipped value of the current variable
d3  domain maxValue constant, i.e. true
d4  domain minValue constant, i.e. false
```

Appendix 2

Having formula $(a \vee b \vee c) \wedge (c \vee \neg b \vee b) \wedge (\neg a \vee \neg b)$ we code it as

```
Variables==3, Clauses==3, Limits[]=={3,2,2},
C[][]=={
  {{1,true,false},{2,true,false},{3,true,false}},
  {{2,true,true},{3,true,false}},
  {{1,false,true},{2,false,true}}}}
}
```