Learning Value Heuristics for Constraint Programming

Geoffrey Chu(✉) and Peter J. Stuckey

National ICT Australia, Victoria Laboratory, Department of Computing and Information Systems, University of Melbourne, Melbourne, Australia
{geoffrey.chu,pstuckey}@unimelb.edu.au

Abstract. Search heuristics are of paramount importance for finding good solutions to optimization problems quickly. Manually designing problem specific search heuristics is a time consuming process and requires expert knowledge from the user. Thus there is great interest in developing autonomous search heuristics which work well for a wide variety of problems. Various autonomous search heuristics already exist, such as first fail, domwdeg and impact based search. However, such heuristics are often more focused on the variable selection, i.e., picking important variables to branch on to make the search tree smaller, rather than the value selection, i.e., ordering the subtrees so that the good subtrees are explored first. In this paper, we define a framework for learning value heuristics, by combining a scoring function, feature selection, and machine learning algorithm. We demonstrate that we can learn value heuristics that perform better than random value heuristics, and for some problem classes, the learned heuristics are comparable in performance to manually designed value heuristics. We also show that value heuristics using features beyond a simple score can be valuable.

1 Introduction

Search heuristics are of paramount importance for finding good solutions to optimization problems quickly. Search heuristics can roughly be divided into two parts: the variable selection heuristic, which selects which variable to branch on, and the value heuristic, which determines which value is tried first. There has been significant research on autonomous search heuristics including: first fail [1], variable state independent decaying sum (VSIDS) [2], domain size divided by weighted degree (domwdeg) [3], impact based search [4], solution counting based search [5], and action[1] based search [6]. Most of these search heuristics concentrate on variable selection, as this is critical in reducing the size of the search tree, although some, in particular impact and action based search also generate value heuristics. Phase saving [7] if a value-only heuristic which reuses the last value of a Boolean variable (its phase) when it is reconsidered. In

[1] It was originally called activity-based search, we use the alternate name to distinguish it from the long established activity-based search used in SAT, SMT and LCG solvers.

L. Michel (Ed.): CPAIOR 2015, LNCS 9075, pp. 108–123, 2015.
DOI: 10.1007/978-3-319-18008-3_8

this paper, we focus on learning useful value heuristics for improving constraint programming search.

Given a current domain D and a variable x to branch on, assuming a maximization problem, the task of the value heuristic is to give the order in which the values in x's current domain should be explored. This is typically accomplished by defining a scoring function $g(D, x, v)$ which gives a score indicating how good assigning the value v to x is likely to be given the current domain D. The values are then sorted based on their scores and visited in decreasing score order. Ideally, g is a function such that $g(D, x, v_1) \geq g(D, x, v_2)$ iff the optimal value down the $x = v_1$ branch is greater than or equal to the optimal value down the $x = v_2$ branch. Such a value heuristic would immediately lead us to the optimal solution. In practice, a perfect scoring function is not likely to be feasible to compute and hence we will settle for a heuristic that is likely to have ordered the good/optimal branches near the front.

Many optimization problems have good, manually designed scoring function which allow the solver to find good solutions quickly. However, manually designing scoring functions can be a time consuming process and requires expert knowledge from the user. Thus there is significant value in developing autonomous search heuristics which work well for a wide variety of problems. One way to produce an autonomous value heuristic is to treat the design of the scoring function $g(D, x, v)$ as a machine learning problem. In order to do so, we have to characterize the current domain D using a set of appropriate features, generate a set of appropriate training instances along with their scores (i.e., values for D, x, v and the output value of the function for these arguments), and use an appropriate regression technique from machine learning to learn the function g.

Several autonomous search heuristics already exist, such as impact based search [4] and action based search [6]. The value heuristics suggested in these two methods can be seen as simple instances of machine learning. In both cases, the current domain is characterized by 0 features (i.e., both of these methods completely ignore the current domain when scoring a value), the training instances are collected during search or during an initial probing phase, and the score assigned to each training instance is the impact (i.e., proportional reduction in domain size) for impact based search, and the number of variables with reduced domains for action based search. In both cases, since there are no features used, the learning simply consists of taking the average score of all the training instances involving an assignment $x = v$ and assigning that average score as the value of $g(D, x, v)$.

There are several possible improvements to these methods. First, both of these methods do not use the current domain in the scoring function at all. This may be fine for problems where the merit of an assignment $x = v$ is largely independent of what else has been assigned. However, in problem classes where the merit of an assignment depends significantly on what else has been assigned, we should be able to learn a much better scoring function by taking into account the current domain D. Thus we are interested in finding features of D which help us to predict the merit of an assignment $x = v$ and using them in our machine learning algorithm. We claim that features of variables which are closer to the decision variable in the constraint graph are more likely to be predictive of the merit of its values. Thus we propose using the features of variables within a

k-neighborhood of the decision variable in the constraint graph as our features, where k is a parameter of our algorithm.

Second, the scores assigned to the training instances in the above two methods, i.e., impact and the number of domain changes, are only indirect measures of how good the subtree is, and there may be better ways to assign scores to the training instances. Indeed neither of these scores consider the objective function of the problem. We propose an alternative scoring method based on the pseudo-cost [8,9], i.e., the change in bound of the objective function after propagating the decision.

Note that the application of machine learning to Constraint Programming in this paper is significantly different from the large body of work using machine learning for solver/algorithm selection in portfolio based solvers (e.g. [10]). There, machine learning is used to predict how well existing solvers/algorithms may perform on a particular instance in order to select a solver/algorithm which works well for the instance. Here, we are using machine learning to predict how well a particular value assignment may do in order to generate new search heuristics. Clearly, these two uses of machine learning are complementary and it is possible to use the search heuristics we generate as the input to the algorithm selection problem.

The contributions of this paper are:

- A framework for learning value heuristics by defining scoring functions, and using linear regression over a restricted class of features of the problem
- A new scoring function, analogous to that used in pseudo-costs [8,9] we can use to define a value heuristics
- A new method of taking the objective function into account for constraint programming search.
- Experiments demonstrating that learnt value heuristics can be as effective as programmed value heuristics

The remainder of the paper is organized as follows. In Section 2, we go through our definitions and background. In Section 3, we describe how to generate training instances for the machine learning algorithm. In Section 4, we discuss feature selection and the machine learning algorithm. In Section 5, we present experimental results. In Section 7, we conclude and discuss future work.

2 Background

Constraint programming A constraint optimization problem (COP) P is a tuple (V, D, C, f) where V is a set of variables, D is a set of domains, C is a set of constraints, and f is an objective function. Let D_x be the domain of variable x. Without loss of generality, we assume the objective function f is to be maximized. A CP solver solves a COP P by interleaving search with inference. It starts with the original problem $P = (V, D, C, f)$ at the root of the search tree. At each node in the search tree, it repeatedly propagates the constraints $c \in C$ to try to infer variable/value pairs in the current domain D which cannot take part in any improving solution to the problem within that subtree. Such pairs are removed from the current domain D to create a new domain D'. The process is

repeated until no more pairs can be removed. We denote this as $D' = \mathbf{solv}(C, D)$. If the resulting domain D' is a *false domain*, i.e. $D'(v) = \emptyset$ for some $v \in V$, then the subproblem has no solution and the solver backtracks. Once the propagation fixed point is reached, if all the variables are assigned, then a solution θ has been found. The solver adds a branch and bound constraint constraining the solver to find only solutions with better objective value than θ, and then continues the search. If not all variables are fixed, then the solver further divides the problem into a number of more constrained subproblems and searches each of those in turn. The search heuristic determines how this division is performed. Typically, the search strategy consists of two parts, a variable selection heuristic which picks an unassigned variable x to branch on, and a value heuristic which pick a value v to try. The search will then explore $x = v$ down one branch and $x \neq v$ down the other branch.

Given a constraint problem $P \equiv (V, D, C, f)$, let its constraint graph G be the graph with the variables V as nodes, and with an edge between two variables $x, y \in V$ iff x and y appear together in at least one constraint $c \in C$. Given a graph G, let the k-neighborhood of a node x in graph G be the set of all nodes within a distance k of node x.

Impact Based Search. Impact based search was proposed in [4]. The impact of a decision $x = v$ can be defined as follows. Let D be the domain before the decision, and $D' = \mathbf{solv}(C \cup \{x = v\}, D)$ be the domain after the decision has been propagated to fixed point. The impact of the decision is then: $1 - \prod_{x \in V} |D'_x|/|D_x|$. In impact based search, a running average $\bar{I}(x = v)$ of the impact of each assignment $x = v$ is maintained. The impact of a variable x given the current domain D is given by $\sum_{v \in D_x} 1 - \bar{I}(x = v)$. The variable heuristic picks the variable x with the highest impact and the value heuristic picks the value with the lowest impact.

Action Based Search. Action based search was proposed in [6]. At each decision in the search tree, the action $A(x)$ of each variable x is decayed by some factor α if x was not fixed before the decision, and $A(x)$ is increased by 1 if its domain was reduced after propagating the decision. The variable heuristic chooses the variable with the highest $A(x)/|D_x|$ value. The action of an assignment $x = v$ is the running average of the number of variables whose domains were reduced after propagating a decision $x = v$. The value heuristic chooses the value with the lowest action.

Linear Regression. In supervised learning, there is an underlying function $h : X \rightarrow Y$ which we wish to learn, and we are given a set of training instances $\{(\bar{x}_1, \bar{y}_1), \ldots, (\bar{x}_n, \bar{y}_n)\}$ such that $\bar{y}_i = h(\bar{x}_i)$ for each i. The goal is to learn an approximation h' of h which is as close to h as possible under some notion of error. The inputs $\bar{x}_i$ and the outputs $\bar{y}_i$ could be single values or could be a vector of values. In this paper, we are interested in the case where the inputs are a set of Boolean or numeric features $x_1, \ldots, x_m$, the output is a single numerical value y, and we are interested in learning a linear function $y = \sum a_i x_i + a_0$ which relates the inputs and the output, where Boolean features are considered 0-1 numeric variables. One common method for doing this is ordinary least

squares regression (OLS) (see e.g.[11]). Unfortunately, OLS is insufficient for our purposes as it requires there to be more training instances than there are features, and the features must be linearly independent. An alternative is partial least squares regression (PLS) [12]. PLS attempts to project the input into a lower dimensional space represented by *latent variables* such that these latent variables explain as much of the variance in the output as possible. The number of latent variables to use is a parameter of the algorithm. PLS is able to handle cases where features may be linearly dependent or where there may be far more features than training instances.

3 Generating Training Instances

In this paper, we would like to treat the design of the scoring function $g(D, x, v)$ used in the value heuristic as a machine learning problem. However, unlike a typical machine learning problem where we are given a set of training instances, in this case, we need to generate our own. Furthermore, it is not obvious what the function $g(D, x, v)$ is supposed to output. One possibility is try to learn a function $g(D, x, v)$ which outputs the optimal value of the subproblem $(V, D, C \cup \{x = v\}, f)$. To do this, we could generate some training instances by picking some D, x and v values and solving the COP's $(V, D, C \cup \{x = v\}, f)$ exactly to get the correct output values. However, this is clearly highly impractical, because solving $(V, D, C \cup \{x = v\}, f)$ exactly is very expensive and we would have to do this for each training instance we want to generate. Alternatively, we could try to learn a function g which outputs an easier to calculate measure which is predictive of how high the optimal value of $(V, D, C \cup \{x = v\}, f)$ is. As long as this measure tends to have higher values for subproblems with higher optimal value, it can still be a good value heuristic.

We consider three different approximate measures for use in computing g: those used in impact and action based search which do not make use of the objective function f of the problem; and one other which attempts to take into account the objective. These measures are

score_impact Impact based search tries to learn a function g which predicts the impact of a particular assignment, with the assumption that lower impact tends to lead to better solutions. We will call this score_impact defined as $g_{score\_impact}(D, x, v) = 1 - \prod_{x \in V} |D'_x|/|D_x|$ where $D' = \mathbf{solv}(C \cup \{x = v\}, D)$.

score_num_red Action based search tries to learn a function g which predicts how many variables will have their domains reduced by a particular assignment, with the assumption that fewer domain reductions lead to better solutions. We'll call this score_num_red defined as $g_{score\_num\_red}(D, x, v) = |\{x \in V \mid D_x \neq D'_x\}$ where $D' = \mathbf{solv}(C \cup \{x = v\}, D)$.

score_pseudo_cost Pseudo-cost branching [8,9] is an important variable selection strategy in mixed integer programming. Recall that we are assuming that the objective f is to be maximized. We try to learn a g which predicts how much the upper bound of the objective function f will decrease by when the assignment is made. Value choices for which the upper bound of f decreased less are likely to lead to better solutions. We'll call this

score_pseudo_cost defined as $g_{score\_pseudo\_cost}(D, x, v) = \max D_f - \max D'_f$ where $D' = \mathbf{solv}(C \cup \{x = v\}, D)$.

Generating training instances to learn these three measures is much easier than generating instances to learn a function to predict the optimal value. Similar to impact based search and action based search, we propose to generate training instances with an initial probing phase followed by a normal search phase. In the probing phase, we use a random value heuristic, restart after every solution, and do not perform branch and bound. The aim of this phase is to get a good coverage of all the assignments. In the normal search phase, we use the learned value heuristic and perform branch and bound as normal. At each node during each of these two phases, when we get to the propagation fixed point and make a decision, we record those (D, x, v) values as a new training instance, and depending on which of the three scoring functions we are trying to learn, the score for this training instance will either be: the impact, the number of variables with reduced domains, or the change in the upper bound of f.

4 Feature Selection

In order to apply machine learning techniques to this problem, we need to define the set of features to be used in the model. Potentially, we could train a single model for $g(D, x, v)$ where x and v are considered features. However, we expect that the relevant features and the way that they affect the value could be very different for different values of x and v. Instead, we train a separate model for each possible assignment $x = v$, i.e., we learn a set of functions $g_{x_1,v_1}(D)$, $g_{x_1,v_2}(D)$, ..., $g_{x_n,v_m}(D)$ s.t. $g(D, x, v) = g_{x,v}(D)$.

We need to extract from D a set of good features for predicting the value of the function we are trying to learn. We claim that the domains of the variables in the problem contain many of the features which are useful for predicting the value. Furthermore, we claim that it is typically the features of the variables which are close to the decision variable in the constraint graph which are the most useful. This is borne out by our analysis of the custom search heuristics for a variety of problems. In most of these custom search heuristics, the features used in the scoring function are simply the lower bounds, upper bounds or assignments of variables close to the decision variable in the constraint graph.

Example 1. Consider the minimization of open stacks problem [13]. We have a set of customers and a set of products. Each customer requires some subset of the products, and has a stack which must be opened when any product they require begins production. The customer's stack can be closed when all the products that the customer requires have finished production. We wish to find the order in which to produce the products such that the maximum number of open stacks at any time is minimized. It has been shown that rather than a model where we determine the order in which to produce products, it is better to determine the order in which we close the stacks of the customers [13]. In the model proposed in [13], we create a customer graph G where the nodes are customers and there is an edge between two customers iff there is a product that they both require. Closing a particular customer's stack means that all the products they require

```
int: n;                              % number of customers
set of int: CUST = 1..n;
set of int: TIME = 1..n;
array[CUST,CUST] of bool: g;         % customer graph

array[TIME] of var CUST: x;          % which customer's stack is closed at time t
array[CUST,TIME] of var bool: open_before;  % customer c open before time t
array[CUST,TIME] of var bool: closed_before;% customer c closed before time t
array[CUST,TIME] of var bool: open_during;  % customer c is open at time t
var CUST: stacks;                    % number of stacks required

constraint forall (c in CUST) (not closed_before[c, 1]);
constraint forall (c in CUST, t in 2..n)
  ( (closed_before[c,t] = (closed_before[c,t-1] \/ x[t-1] = c)) /\
    (closed_before[c,t] -> x[t] != c) );
constraint forall (c in CUST, t in TIME)
  (open_before[c,t] = ((if t > 1 then open_before[c,t-1] else false endif)
                       \/ exists (d in CUST where g[c,d]) (x[t] = d)) );
constraint forall (c in CUST, t in TIME)
  (open_during[c,t] = (open_before[c,t] /\ not closed_before[c,t]));
constraint forall (t in TIME)
  ( sum (c in CUST) (bool2int(open_during[c,t])) <= stacks );

solve minimize stacks;
```

Fig. 1. A MiniZinc [14] model for minimization of open stacks

must be produced before that time, which in turn means that the stacks of all its neighbors in the customer graph must be opened before that time. This leads to the model shown in Figure 1.

A good variable ordering is simply to label the x variables in order, as that produces the best propagation. The value heuristic proposed in [13] picks the customer which opens the fewest new stacks at each stage. In terms of the variables in this model, the score for the decision $x[t] = c$ can be written as: $\sum_{d \in CUST \text{ where } g[d,c]} (opened\_before[d,t] - 1)$, where we are simply giving a penalty of 1 to each stack that closing customer c's stack would force open and which is not already open. Clearly, this scoring function is simply a linear combination of the values of variables which already exist in the model.

We divide integer variables into two classes: value type integer variables and bound type integer variables.

- Value type integer variables typically have small domains. The value are unordered and each value means a completely different thing. They are typically involved in constraints like `alldifferent`, `element` or `table` where there is a lot of propagation based on values. For value type integer variables x for each value v in its original domain, we take the truth values of $D \Rightarrow x = v$ and $D \Rightarrow x \neq v$ as features where D is the current domain.
- Bound type integer variables on the other hand could have much larger domains, and the values are ordered, so values close together are closely related. They are typically involved in constraints like `cumulative` or linear constraints where there is only propagation based on bounds. For bounds type integer variables, we take their lower bound and upper bounds as features.

For a Boolean variable b, we take the truth values of $D \Rightarrow b$ and $D \Rightarrow \neg b$ as features. When used in a linear regression, integer features are kept as is, while Boolean features are converted to 0-1 integers.

Example 2. Suppose we have Boolean variables b_1, b_2 and b_3, with current domain $b_1 \in \{true\}$, $b_2 \in \{false\}$ and $b_3 \in \{false, true\}$. The two features for a Boolean variable b are the truth values of $D \Rightarrow b$ and $D \Rightarrow \neg b$. For b_1, they are 1 and 0 respectively. For b_2, they are 0 and 1 respectively. For b_3, they are 0 and 0 respectively. Suppose we have value type integer variables x_1 and x_2, both with original domain $\{1, 2, 3\}$ and current domains $x_1 \in \{1, 3\}$ and $x_2 \in \{2\}$. The features are the truth values of $D \Rightarrow x = v$ and $D \Rightarrow x \neq v$ for each v in the original domain. For x_1, this gives 0 and 0 for $v = 1$, 0 and 1 for $v = 2$, and 0 and 0 for $v = 3$. For x_2, this gives 0 and 1 for $v = 1$, 1 and 0 for $v = 2$ and 0 and 1 for $v = 3$. Suppose we have bound type integer variable x with current domain $\{2, \ldots, 153\}$. The two features are simply its lower and upper bound, i.e., 2 and 153.

In general, it is difficult to tell which variables have features which are useful for the function we wish to learn. We could of course, use the features of all the variables in the problem and use some standard feature selection algorithm to find a good subset of them. However, such methods are far too expensive in this context and are prone to over-fitting due to the large number of potential features and a limited number of training instances. Instead, we exploit our knowledge that variables closer to the decision variable in the constraint graph tend to be more useful and define a series of subsets of features to check. For each $k = 0, 1, 2, \ldots$, we pick the features of the variables in the k-neighborhood of the decision variable in the constraint graph as our features. Using a larger neighborhood may mean that useful features get included, improving the performance of the learned function, but it may also add irrelevant features and produce over-fitting as well as increase overhead. Note that using the 0-neighborhood with score_impact and score_num_red corresponds to the value heuristics used in impact based search and action based search respectively. However, here we have the potential to use higher k to learn that other assignments have an effect on the current decision.

After the training instances are generated and the features are selected, we can run our regression algorithm. We choose to use the partial least squares regression method. The reason is that the vast majority of custom scoring functions we analyzed were simple linear combinations of features, and thus we believe a linear function should do well. Secondly, we have to deal with co-linearity in the features as well as the possibility that there are more features than training instances. Partial least squares regression is able to handle all these and is therefore a good choice. We run the regression algorithm once when the probing phase is complete. After that, we re-run it every time we double the number of our training instances. In the special case where we are using a 0-neighborhood of features, there are actually no features at all, so we can simply keep a running average and update the scoring function whenever we get a new training instance.

Example 3. Consider the minimization of open stacks problem again. Suppose we are branching on $x[t]$. A 1-neighborhood will include the *open_before* and *closed_before* variables from time t and $t - 1$. A 2-neighborhood would include the *open_before* and *closed_before* variables from time $t - 2$ to $t + 1$, as well as the *open_during* variables from time t. Suppose we use a 1-neighborhood with the score_num_red scoring function. We pick a random decision from a random

instance for illustrative purposes. In this instance, the custom scoring function for the assignment $x[3] = 4$ is: $1 * open\_before[3, 4] + 1 * open\_before[3, 15] + 1 * open\_before[3, 27] + 1 * open\_before[3, 29] - 4$.

The scoring function learned using partial least squares regression after the training instance has significantly more terms. However, the terms with the largest (absolute value of) coefficients are: $28.716 * open\_before[3, 4]$, $28.740 * open\_before[3, 15]$, $24.047 * open\_before[3, 27]$, $24.485 * open\_before[3, 29]$, $33.880 * closed\_before[3, 16]$, $-26.664 * closed\_before[3, 19]$, $-24.726 * closed\_before[3, 26]$, and it can be seen that the features considered important in the custom scoring function also have large coefficients in this learned scoring function. However, several terms not in the custom scoring function also have large coefficients here, possibly representing other useful features. In practice however, despite the differences, our experiments in Section 5 show that this learned value heuristic is almost identical in strength to the custom one.

5 Experiments

The experiments are run on Intel Xeon 2.40GHz processors using the CP solver Chuffed. We use the minimization of open stacks problem (see e.g. [13]) (MOSP), the talent scheduling problem (see e.g. [15]) (Talent), the resource constrained project scheduling problem (see e.g. [16]) (RCPSP), the nurse scheduling problem [17] (Nurse), the traveling salesman problem (TSP), and the soft car sequencing problem [18] (CarSeq). We select some hard instances from the J60 benchmark for RCPSP and generate 100 random instances for the other 5 problem classes. MiniZinc models and data for the problems can be found at: www.cs.mu.oz.au/~pjs/learn-value-heuristic/.

For each problem, we use a k-neighborhood for feature selection as described in Section 4 with $k = 0, 1, 2$. For RCPSP and TSP, $k = 1$ is identical to $k = 2$ since it already includes all the variables, so we only give results for $k = 1$. We try each of the three scoring methods for the training instances described in Section 3. We use a 10 second probing phase followed by a 590 second search phase for a total of 10 minutes per instance. We use a limit of 10 latent variables in the partial least squares regression method.

Since we are principally interested in the value heuristic part of the search heuristic in this paper, for the first experiment we use the same variable selection heuristic for all the different settings of the value selection heuristics so we can just compare the effect of the value heuristic.

We use an in-order variable heuristic for the minimization of open stacks problem, the talent scheduling problem, the nurse scheduling problem, and the soft car sequencing problem. We use a max-regret variable heuristic for the traveling salesman problem. And we use the earliest first variable heuristic (also called schedule generation [19]) for the resource constrained project scheduling problem. We also compare using a random value heuristic and a manually designed value heuristic. We use the following manually designed value heuristics. For the open stacks problem, we use the one described in [13], which tries to pick the customer which opens the fewest new stacks. For the talent scheduling problem, we pick the scene which minimizes the cost of new actors plus the cost

Table 1. Cost of partial least squares regression as a percentage of total run time

	1-neighborhood	2-neighborhood
MOSP	0.4%	3.3%
Talent	1.1%	3.2%
RCPSP	9.1%	–
Nurse	6.2%	67.5%
TSP	1.2%	–
CarSeq	0.4%	1.7%

Table 2. Solution quality at the end of 10 minutes

	random	custom	pseudo_cost-0	impact-0	num_red-0	pseudo_cost-1	impact-1	num_red-1	pseudo_cost-2	impact-2	num_red-2
MOSP	23.6	13.6	26.8	19.3	21.4	26.6	18.4	14.0	26.0	20.3	14.3
Talent	999	422	678	865	770	650	687	580	640	693	571
RCPSP	972	119	119	121	426	798	974	828	–	–	–
Nurse	66.5	136.9	127.6	70.3	86.5	93.8	68.2	78.4	98.9	66.5	77.1
TSP	1157	521	527	874	881	535	968	1036	–	–	–
CarSeq	32.4	8.4	24.0	35.4	35.4	31.6	26.6	27.0	29.0	32.7	31.1

of actors who are on-location but not in the scene. For the resource constrained project scheduling problem, we assign the start time to its current lower bound. For the nurse scheduling problem, we assign the nurse to the available shift they most prefer. For the traveling salesman problem, we pick the closest available city. For the car-sequencing problem, we pick the car type which utilizes the most heavily loaded available machine.

The average cost of the partial least squares regression as a percentage of run time is given in Table 1. The costs are generally quite small at just a few percent, however, for nurse scheduling with a 2-neighborhood, it grows to a rather massive 67.5%.

The solution quality at the end of 10 minutes is given in Table 2. The graph for average solution quality over time is given for each problem in Figures 2. The best learned heuristics are: for open stacks score_num_red-1, for talent scheduling score_num_red-2, for RCPSP score_pseudo-cost-0, for nurse scheduling score_pseudo-cost-0, for travelling salesman score_pseudo-cost-0, and for car sequencing score_pseudo-cost-0.

Note that these searches do not tend to find any good solutions during the initial 10 second probing phase where it is using a random value heuristic with no branch and bound. After that however, they may start finding much better solutions. It can be seen that in all the problems, there are some settings which allow the algorithm to learn a value heuristic which is significantly better than random. In some cases, the learned value heuristic is of comparable performance to the manually designed value heuristics. It can be seen that using a k-neighborhood with $k > 0$ is highly beneficial on problems like minimization of open stacks and talent scheduling, where whether a particular value is good or not depends significantly on what other decisions have been made. On other problems however, the extra features from using a larger neighborhood are not useful and only cause over-fitting, degrading the performance of the learned value heuristic. It can also be seen that using score_pseudo_cost to score the training instances is far better than using score_impact or score_num_red in many cases.

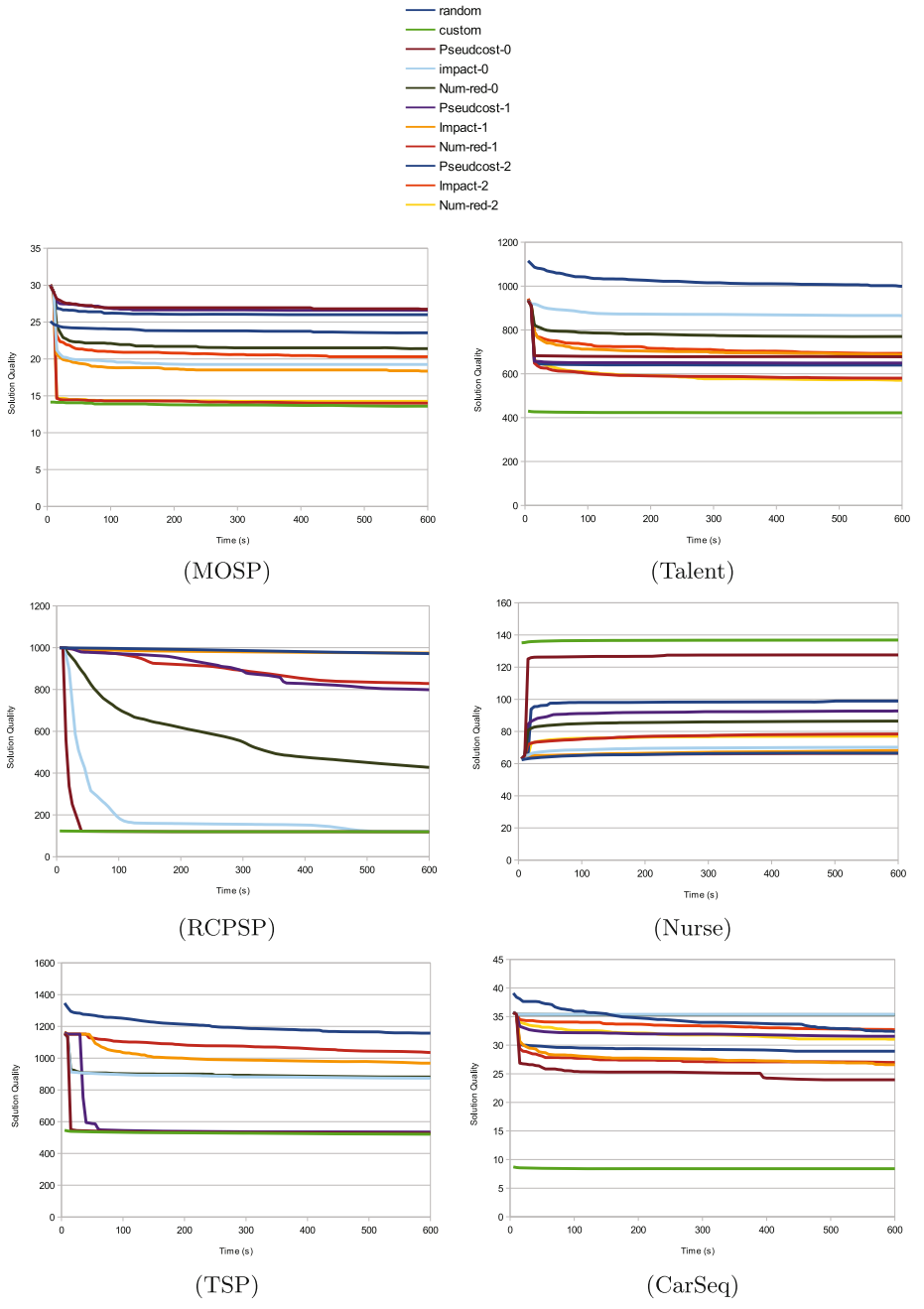

Fig. 2. Solution Quality vs Time graph for various value heuristics for the 6 problem classes. Smaller is better for all except Nurse where larger is better.

Table 3. Number of times each setting was best out of 1000 random samples of 5 instances

	pseudo_cost-0	impact-0	num_red-0	pseudo_cost-1	impact-1	num_red-1	pseudo_cost-2	impact-2	num_red-2
MOSP	0	0	0	0	0	635	0	0	365
Talent	0	0	0	0	1	382	5	1	611
RCPSP	1000	0	0	0	0	0	–	–	–
Nurse	1000	0	0	0	0	0	0	0	0
TSP	761	0	0	239	0	0	–	–	–
CarSeq	840	0	0	0	107	51	0	0	2

Table 4. Solution quality at the end of 10 minutes

	random	custom	impact	action	vsids	ml-5
MOSP	23.6	13.6	19.2	18.9	14.0	14.1
Talent	999	422	992	1068	1236	575
RCPSP	972	119	774	415	118	119
Nurse	66.5	136.9	69.0	76.1	140.5	127.6
TSP	1157	521	108	846	722	529
CarSeq	32.4	8.4	52.8	52.3	29.6	24.4

Although it may be difficult to know beforehand which settings will be best for a problem class, the relative performance of each setting is usually the same across all instances in a problem class, i.e., the good settings tend to do well on all instances and the bad settings tend to do badly on all instances. Hence if we need to solve a large number of instances from the same problem class, we can simply solve a few sample instances using the different settings, and use the setting which had the best average performance on the sample instances for the rest of the instances in the benchmark.

In Table 3, we show the number of times each setting had the best performance for 1000 different random samples of 5 instances from each of the benchmark. As can be seen, simply by trying the different settings on 5 instances, we will almost always end up picking the optimal or near optimal setting for the problem class.

In the third experiment we compare our method against various existing autonomous searches. From the first and second experiments, we can work out the expected solution quality over time curve of our method when we pick the setting by picking the best performing one on a random sample of 5 instances. That is, we take a weighted average of the curves in the first experiment, where they are weighted by the numbers in Table 3. We will call this ml-5. We compare against full impact based search impact, action based search action, the variable state independent decaying sum heuristic vsids, and the random and custom search heuristics from the first experiment.

The solution quality at the end of 10 minutes is shown in Table 4. The graph for average solution quality over time is given for each problem in Figures 3. It can be seen that our new value heuristic is generally much better at finding good solutions than the other autonomous searches, although for nurse scheduling, VSIDS is so good that it beats our heuristic and even beats the custom search.

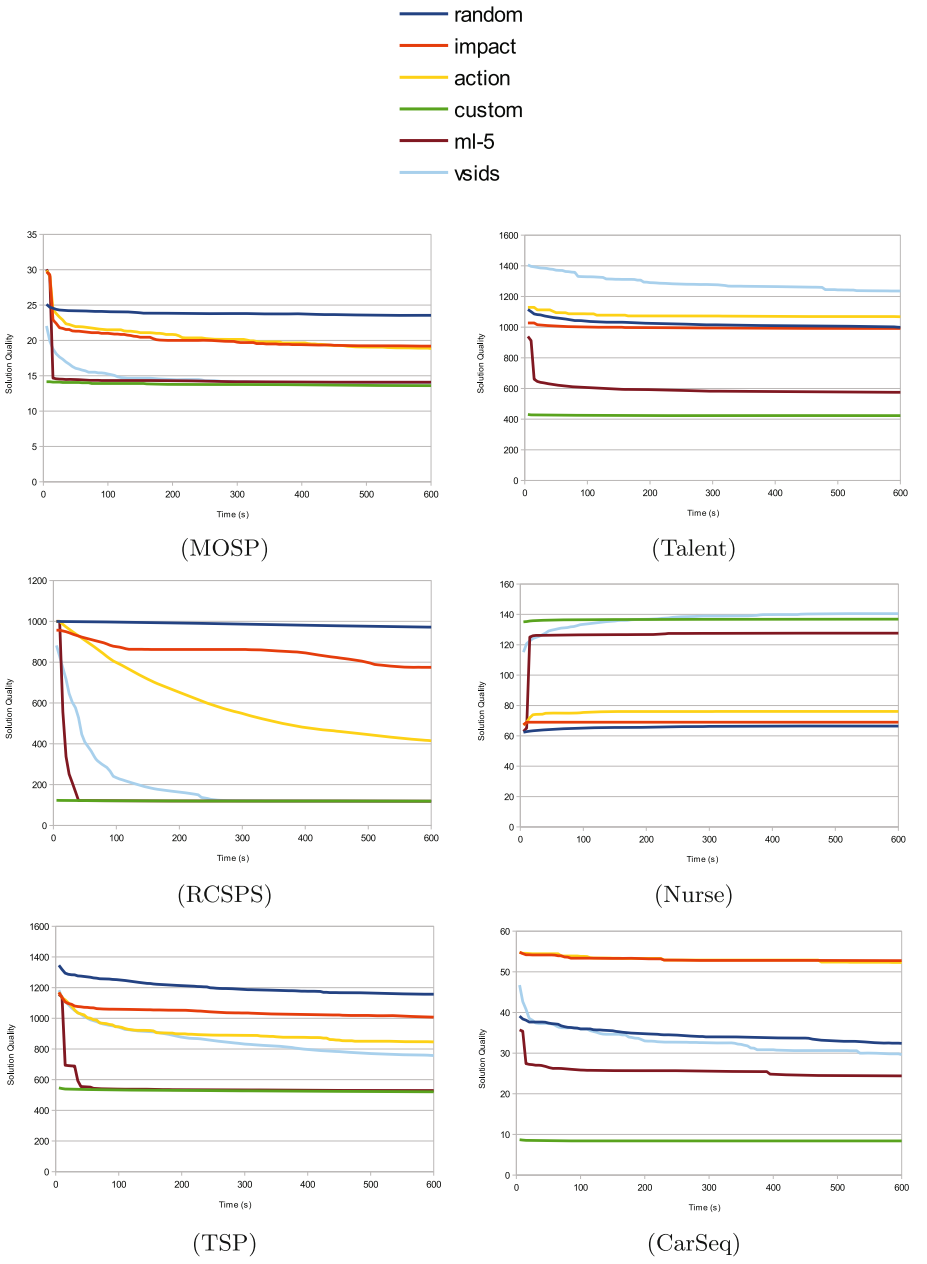

Fig. 3. Solution Quality vs Time graph for various search heuristics on the 6 problem classes. Smaller is better for all except Nurse where larger is better

6 Related Work

A closely related work is Bandit-based Search for Constraint Programming [20]. This method is based on Monte-Carlo tree search and uses reinforcement learning to learn which values should be explored first. They use a reward function based on whether the decision led to a failure depth which was above or below the average, so they do not really target optimization problems. This method differs from ours in that it uses reinforcement learning and does not attempt to predict the reward from features of the domain like our method.

Solution counting [5] is a powerful method for defining autonomous search, giving both variable and value heuristics. It relies on extending propagators to count or estimate the number of remaining solutions they have. It learns estimators for variable value pairs, similar to impact and action based search. Again it does not directly take into account the objective. It would be interesting to explore measures based on counting in our framework, where the current domain D can also be taken into account.

Regret [21] is a commonly used variable selection strategy for CP problems where the objective is the sum of a set of variables. The regret is equivalent to the difference in score_pseudo_cost for the two largest values that remain in the domain of a variable, for these problems.

Pseudo-cost branching [8,9] is an important MIP heuristic for variable selection. It is also used in the ToulBar2 [22] weighted CSP solver. It ranks variables by the expected gain per unit change in the variable, which is the difference between neighbouring values in score_pseudo_cost. Value heuristics are not common in MIP search, since node exploration is more commonly implemented by selecting from a frontier of open nodes, but it would be unsurprising if pseudo-cost had been used as a value heuristic in MIP.

Given that regret for CP and pseudo-costs for MIP are important search heuristics it is surprising that we are not aware of widespread use of pseudo-cost for CP search heuristics.

7 Conclusion

Autonomous search is an important topic for constraint programming, since its removes the burden from the modeller of deciding how best to search for solutions. The majority of work on autonomous search for CP has concentrated on variable selection heuristics since these can have a significant effect on the size of the search tree. But when we consider optimization problems, the value heuristic used can also substantially effect the size of the search tree. Similarly when we are considering optimization problems that are too hard to find/prove optimal solutions, value heuristics can make a significant difference on the quality of solutions found in a limited time. In this paper we define a framework for learning value heuristics by combining a score function, feature selection, and machine learning. We show that we can learn value heuristics that are comparable to programmed heuristics, and the cost of learning can be paid for during the search.

While we have investigated some choices for score functions, feature selection and machine learning each component of the framework could be replaced,

leaving us wide scope for further exploration of the framework. Clearly we can imagine many other: score functions, e.g. the objective of the first solution found in a subtree; feature selections, e.g. a tighter definition of neighbouring variables using constraint activity; and learning methods, such as polynomial regression; which might be worth considering.

Acknowledgments. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work was partially supported by Asian Office of Aerospace Research and Development grant 12-4056.

References

1. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. **14**, 263–313 (1980)
2. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, pp. 530–535. ACM, Las Vegas, June 18–22, 2001
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: de Mántaras, R.L., Saitta, L. (eds.) Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI 2004, Including Prestigious Applicants of Intelligent Systems, PAIS 2004, pp. 146–150. IOS Press, Valencia (2004)
4. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
5. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. Constraints **14**, 392–413 (2009)
6. Michel, L., Van Hentenryck, P.: Activity-based search for black-box constraint programming solvers. In: Beldiceanu, N., Jussien, N., Pinson, E. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 228–243. Springer, Heidelberg (2012)
7. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
8. Benichou, M., Gauthier, J., Girodet, P., Hentges, G., Ribiere, G., Vincent, O.: Experiments in mixed-integer programming. Mathematical Programming **1**, 76–94 (1971)
9. Linderoth, J., Savelsbergh, M.: A computational study of search strategies for mixed integer programming. INFORMS Journal of Computing **11** (1999)
10. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. CoRR abs/1210.7959 (2012)
11. Amemiya, T.: Advanced Econometrics. Harvard University Press (1985)
12. Wold, H.: Estimation of principal components and related models by iterative least squares. In: Multivariate Analysis, pp. 391–420. Academic Press (1966)
13. Chu, G., Stuckey, P.J.: Minimizing the maximum number of open stacks by customer search. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 242–257. Springer, Heidelberg (2009)
14. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.R.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
15. Garcia de la Banda, M., Stuckey, P., Chu, G.: Solving talent scheduling with dynamic programming. INFORMS Journal of Computing **23**, 120–137 (2011)

16. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Explaining the cumulative propagator. Constraints **16**, 250–282 (2011)
17. Miller, H., Pierskalla, W., Rath, G.: Nurse scheduling using mathematical programming. Operations Research, 857–870 (1976)
18. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: ECAI, vol. 88, pp. 290–295 (1988)
19. Hartmann, S., Kolisch, R.: Experimental evaluation of state-of-the-art heuristics for resource constrained project scheduling. European Journal of Operational Research **127**, 394–407 (2000)
20. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 464–480. Springer, Heidelberg (2013)
21. Savage, L.: The theory of statistical decision. Journal of the American Statistical Association **46** (1951)
22. Allouche, D., de Givry, S., Schiex, T.: Toulbar2, an open source exact cost function network solver. Technical report, INRIA (2010)