Modeling and Automatic Tuning in High Performance Computing

Seminar: Future Trends in High Performance Computing

Ludwig Gärtner
Fakultät für Informatik
Technische Universität München
Email: ludwig.gaertner@tum.de

***Abstract*— This literature research lays the groundwork for integrating a machine learning based algorithm selector into the molecular dynamic simulation framework AutoPas. Two reference projects were consulted to analyse the performance of using machine learning models for automatic algorithm selection One is using supervised learning, the other reinforcement learning.**

The results though show no clear favorit among the two, as there are some key performance measurements missing for a final comparison.

***Index Terms*— High Performance Computing, Auto Tuning, Automatic Algorithm Selection, Machine Learning**

I. Introduction

Performance portability is a highly desirable feature of applications in High-Performance Computing. Being able to execute the same application on architecturally different supercomputers without changing the application itself reduces the programmer's burden in managing these different architectures. This may involve generating the optimal compiler flags at compile time or selecting the optimal algorithm to solve the given problem.

This research was motivated by the idea of integrating a machine learning based automatic algorithm selector into the already existing auto tuning library AutoPas [1]. AutoPas acts as a black-box auto tuner for N-Body simulations. It can choose between different containers (namely how the particles are stored), traversal patterns, data layouts for storing the particles within a cell (SoA and AoS), etc. This choice is currently done by comparing all possible combinations at runtime and then selecting the best among those. Ideally, this statistical evaluation will be replaced by a machine learning model, which can make these decisions faster and more accurately.

The results of this paper shall lay the groundwork for a machine learning based algorithm selector by demonstrating how other projects approached their design of algorithm selectors and how they perform.

II. Theoretical Background

A. Automatic Algorithm Selection

John Rice formally introduces the Algorithm Selection Problem in his paper [2] published in 1976. He describes the Algorithm Selection Problem in its most basic form as a set of characteristics:

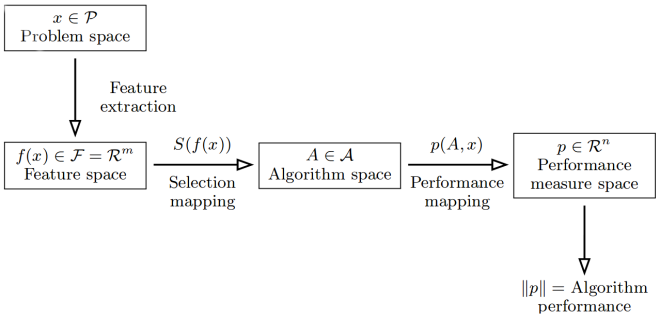

Fig. 1. The Algorithm Selection Problem, extended by the feature space. Taken from [3]

- The first of those is the problem space P. It defines the set of problem instances.
- Secondly, the algorithm space A is the set of algorithms that take a problem instance as an input.
- Finally, the performance measurement R describes different qualities of the solution output of an algorithm, for example execution time, accuracy, or any combinations of those qualities.

Additionally, the model has mappings between these criteria. The selection mapping s takes a problem instance and maps it to an algorithm. The performance mapping p takes the problem instance and the previously selected algorithm and evaluates the performance of the given algorithm. [2]

To make this theoretical model more applicable to real problems Rice extends his definition by another characteristic, the feature space (see Figure 1). A feature vector to a given problem instance represents this instance by holding its measurable properties. This effectively results in a reduction of the dimensionality of the problem space, by considering only important details of the instance but disregarding uninformative noise. Rice describes "the features as an attempt to introduce an approximate coordinate system in P." [2] In this extended model the selection mapping takes the feature vector of a problem instance as an input. The performance mapping however still depends on the problem instance itself coupled with an algorithm.

The objective in the Algorithm Selection Problem is to find a selection mapping s that maximizes the performance to solve every problem instance. The two different philosophies of

this selection are classification and regression based. For the former, the selector takes the features of the given problem instance and explicitly produces the algorithm it predicts to be best at solving the instance. It does not give any other information that would explain its choice. In most cases, this is not a simple binary classification but a multi-class classification, deciding between all possible algorithms in one step. Techniques that implement this approach are for example decision trees or the k-nearest-neighbor algorithm [4]. The regression based selection predicts the performance of every single algorithm when solving a problem instance, and then selects the algorithm based on the comparison of these predictions. This may seem more advisable as one can make the final selection based on more information, but it also comes at the cost of having to train more predictors. Ridge Regression and Random Forests are examples of regression based predictors. [4]

Another design choice is between Offline- and Online Automatic Algorithm Selection. They differ in the time the selection mapping is constructed. For the former approach, the mapping is trained entirely on a pre-constructed data set. After the training procedure, the selection mapping is kept the same during usage. This resembles the supervised learning method in machine learning, which is explained in subsection II-C of this paper. For the online approach, there may be some training of the mapping in advance, but the key difference is that the selection mapping is adjusted and retrained with data that is gathered during execution. See subsubsection II-C.3 about Reinforcement Learning for the technical training side of the online approach. The Contextual Multi-Armed Bandit in subsection II-B covers the problem of exploration versus exploitation one has to face when applying Online Algorithm Selection.

As the only difference between online and offline approaches lies in the timing of training, this is also where their relative strengths and weaknesses lie. The offline model is faster during execution as no retraining of its decision maker is required. The online model, on the other hand, can start faster as its initial training is made on less data (if there even is any initial training).

B. Contextual Multi-Armed Bandit

The Multi-Armed Bandit Problem is something that is applicable both to Online Automatic Algorithm Selection as well as Reinforcement Learning. Imagine a gambler standing in front of K different slot machines. At any time he can choose which arm to pull / which slot machine to use to maximize his overall profit. As he is playing on the different machines the gambler is gathering information on the profit he is making at each machine. The more rounds he plays on a single machine, the more confident he can be in his expectations. The dilemma he finds himself in is whether to take advantage of what he expects to be the most profitable machine or to invest some, to his knowledge, non-optimal turns trying to find a more rewarding one.

More formally the Multi-Armed Bandit problem is modeled with random variables $X_{i,n}$ which stand for the earnings of playing at slot machine $i = 1 \dots n$ at time step $n \geq 1$. For one machine j the random variables $X_{j,n}$ are independent and identically distributed over the time steps n. The random variables for different machines are also independent but most of the time differently distributed. This results in most machines having different expected rewards $\mathbf{E}[X_i] = \mathbf{E}[X_{i,n}]$ with $n \geq 1$. [5]

The Contextual Multi-Armed Bandit Problem accommodates for the slot machines different behavior depending on a context. So additionally to just pulling the arms on bandits and gather profit information on each machine the gambler has to connect some contextual data to the rewards he gets in each turn [4]. This problem is more applicable to our problem of Automatic Algorithm Selection as the same algorithm does not perform equally on different input. One basic example of this would be with sorting algorithms. The average case time complexity of Quicksort is $O(n \log n)$, but the sorting of some badly aligned data would be in $O(n^2)$. If the gambler could connect indications of such a case in the context data to the performance of his available sorting algorithms, he would be able to choose a better performing one.

One strategy to solve this problem of exploration versus exploitation is to always select the slot machine that is the most promising according to the current knowledge. This is called the *greedy* method. The gambler's plan is to neglect active exploration and to always go for the most rewarding strategy. Keep in mind though that even this selection strategy performs some implicit exploration, as the optimal choice constantly changes due to for example differences in the context or just new data. [4] To start of though, each algorithm has to be executed at least once, to offer the *greedy* selector something to choose from.

A variation of this is the ϵ-greedy strategy. It selects the best choice in some turns, and actively explores in others. Before the decision is made a random variable selects whether to go with the greedy choice with probability ϵ, or to select an entirely random alternative with probability $1 - \epsilon$, with $0 \leq \epsilon \leq 1$. [4]

The last strategy that is discussed here is called the *Upper Confidence Bound* (UCB). As with the strategies above the gambler has to compute the expected reward $E[X_i]$ for selecting machine i. Additionally, he needs the statistical standard error e_i of his gathered data samples for that machine, which represents the uncertainty of his expectation. The final prediction is done by finding the machine that maximizes

$$UCB(X_i) = E[X_i] + \lambda e_i \tag{1}$$

λ is a constant which controls the amount of exploration of the UCB method. A smaller λ results in more focus on the expected reward and less on the possible potential with $\lambda = 0$ being equal to the *greedy* method. The idea behind the UCB strategy is fewer data points for a choice results in a higher e and therefore a higher chance of being the optimal UCB choice. [4]

C. Machine Learning

Machine Learning is the concept of training a model to approximate an unknown function using a representative set of data samples. A data sample consists of an input x_i and an output y_i. The model is trained to find patterns in the data set to generate a function $f(x_i) = \tilde{y}_i$ with $\tilde{y}_i \approx y_i$. In order to improve the accuracy and learning rate of the Machine Learning algorithm a preprocessing step called *feature extraction* is ran on the input data. This generates a feature vector $\phi_i = \phi(x_i)$ of the data sample. Its purpose is to describe the input of the data sample accurately but reduce insignificant variability of the input values. [6]

Training a Machine Learning model is divided into two phases: the *training phase* and the *validation phase*. In the training phase, a portion of the data samples is used to train the model. The technical side of this training depends on the machine learning algorithm that is used. For two examples see subsubsection II-C.1 about Ridge Regression and subsubsection II-C.2 about Regression Forest models. In the validation phase, the rest of the data samples are used to validate the accuracy of the trained model. Dividing the data set into training and validation sets serves the purpose of generalization. Generalization means the machine learning model detects patterns that occur in real data. The opposite of that is overfitting which lets the model focus on data that is specific only to the training set but hurts the prediction of data points outside the training data set. This way of training a model is called *supervised learning*. Its name comes from an external knowledgeable supervisor providing the labeled training data set. [6]

Here are some notational remarks for the rest of this paper:

- $\phi_i \in \mathbb{R}^p$ is a feature vector containing p feature values.
- $X = (\phi_1, \ldots, \phi_n)^T$ is a $n \times p$ matrix of n feature vectors.
- $y = (y_1, \ldots, y_n)^T$ is the target vector for the dataset.

1) Ridge Regression: A Ridge Regression model is trained by fitting a linear function

$$(X^T)^{-1}(X^T X + \alpha I_p)w = y \tag{2}$$

with I_p being the $p \times p$ identity matrix and α a regularization constant on the training data set (X, y). The goal of the training is to find a fitting weight vector w by converting Equation 2:

$$w = (X^T X + \alpha I_p)^{-1} X^T y \tag{3}$$

This weight vector can then be used to predict the target value of a new input:

$$f_w(x_{n+1}) = w^T \phi_{n+1} \tag{4}$$

For $\alpha = 0$ this model is equal to a simple linear regression model. For any $\alpha > 0$ large values in w are penalized which regularizes the model by avoiding overfitting the model to training data. Additionally this regularization increases the numerical stability of ridge regression in case X is rank deficient (not full ranked) [7]. See Figure 2 for a graphical representation of the impact of the regularization constant α.

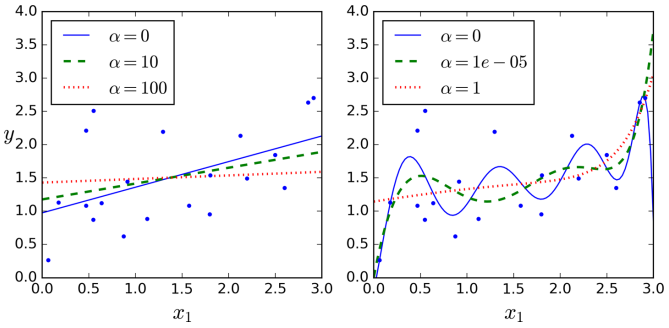

Fig. 2. Ridge Regression with regularization constant α. On the left is the regression model trained on linear data. On the right the data is expanded with a 10 degree polynomial expansion. Taken from [8]

The time complexity of training a Ridge Regression model is bound to the complexity of inverting the $p \times p$ Matrix

$$A = X^T X + \alpha I_p \tag{5}$$

in Equation 3 which is in $O(p^3)$.

The generalization performance of a ridge regression based prediction model is dependent on the quality of its feature set. With uninformative or highly correlated features comes a deteriorated generalization performance. A preprocessing step called *feature selection* is performed on the feature set to work around the problem of constructing a feature set that does not suffer from this effect. The *two phase forward selection* is one prominent method for feature selection [7]. In this method, one starts by adding l features to an empty set, always minimizing the cross-validation error (Forward Selection). After that, the feature set ϕ_i is augmented by new features $\phi_{ij} * \phi_{ik}$ for $j = 1..l$ and $k = j..l$ (Quadratic expansion). Another Forward Selection of q features gives the final feature set. The reason for the first selection is scalability. Uninformative features are filtered out to not blow up the intermediate feature set with more uninformative features during the expansion [7].

A quadratic expansion of the feature set improves the predictive quality of the model even further. Hutter states that "[algorithm] runtime can often be better approximated by a polynomial function than a linear one" [7]. For the same reason expansions with even higher degrees were performed for Figure 2.

Due to its computational and conceptual simplicity, and at the same time predictive competitiveness, Ridge Regression "has been used most frequently in the past for building EPMs" [7] (Empirical Performance Model) which are used to predict the performance of algorithms.

2) Regression Forest: The Regression Forest is another way of mapping a feature vector to a target value. A Regression Forest consists of several Regression Trees. The leaves of a Regression Tree partition the input/feature space into disjoint Regions $R_1, \ldots, R_M$. The predicted target value for a new

input is then

$$\mu(x_{n+1}) = \sum_{m=1}^{M} c_m \cdot \mathbb{1}_{x \in R_m} \tag{6}$$

where $\mathbb{1}_{x \in R_m}$ is the indicator function and c_m is the average of all y_i with $x_i \in R_m$. The complexity of prediction with a Regression Tree is linear in the depth of the tree. [7]

The construction of a Regression Tree starts with the set of all training points $D = (\phi_1, y_1), \ldots, (\phi_n, y_n)$ at the root node of the tree. From there on the data points of a node are recursively split into two child regions $R_1(j, s)$ and $R_2(j, s)$. A data point x_n falls into region $R_1(j, s)$ if $\phi_{nj} < s$ and into $R_2(j, s)$ otherwise. The split variable j and split value s are to be selected to minimize the squared difference to the mean in each region:

$$l(j, s) = \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \tag{7}$$

To avoid overfitting a node is split only if its region contains a number of nodes greater than n_{min}, rather than splitting until every node contains only data points with equal features. The complexity of constructing such a Regression Tree is at best $O(p \cdot n \log^2 n)$ in case of a balanced Tree. At worst its complexity is $O(p \cdot n^2 \log n)$ where only ever singular nodes are split of the dataset. According to Hutter et al. "[Regression] trees are not perfectly balanced, but are much closer to the best case than to the worst case." [7]

The use of a Regression Forest is yet another attempt to address the persistent problem of overfitting. Instead of training and predicting with only one tree, a set of Regression Trees is constructed. Each of the trees is trained a little different. One has the choice of using different subsets of the training data to train a tree or only using random subsets of the feature space as split variables. Hutter et al. recommend the latter based on some, only mentioned, experiments. [7]

Prediction with a Regression Forest is done by computing the mean over all predicted outputs of the trees within the forest. The predictive quality grows with the number of trees, as do the computational costs of construction and prediction. [7]

3) Reinforcement Learning: Reinforcement Learning takes a different approach of training models compared to the previously described supervised learning. Instead of being given a training dataset that is (hopefully) representative of the entire problem space, the reinforcement learning model is put in a position where it explores the problem space on its own. It does this exploration while simultaneously trying to achive a certain goal [9]. This ties it back to the Contextual Multi-Armed Bandit problem from subsection II-B.

Training a reinforcement learning model is a constant cycle of "sensation, action, and goal" [9]. One starts by observing and analyzing the current situation/context, which can formally be described as taking the input sample x_{i+1} and producing its feature vector ϕ_{i+1}. The second step is to pick an action based on its predicted output $\tilde{y}_{i+1}$. The final step is to compare the predicted output $\tilde{y}_{i+1}$ to the actual observed output of the performed action y_{i+1} to work the consequence into the prediction model such that future actions bring it closer to its goal. In reality, it is computationally too expensive to retrain the model after every single action. Therefore one can wait until a minimum number of new data samples is available and only then retrain the prediction model. [4]

III. Analysis of Reference Projects

A. SATzilla07

SATzilla [10] is a collective solver for the Propositional Satisfiability Problem (see subsubsection III-A.1). Collective solver means that instead of trying to optimize a single solver an algorithm selection model decides for every given instance which solver to use.

The version of SATzilla that is described and analyzed in this section is SATzilla07, which was SATzilla's entry into the SAT 2007 competition. All of the performance analysis that is done in subsubsection III-A.3 of this paper is based on the results in this competition.

1) The Propositional Satisfiability Problem - SAT: In the rest of this paper the problem of Propositional Satisfiability will be abbreviated by SAT.

To understand the applicability of the algorithm selection problem to SAT, there will be an explanation of SAT itself as well as a brief overview of two classes of solvers in this section.

SAT is the problem of determining whether one can assign truth values to the variables of a *formula* of propositional calculus such that the formula evaluates to *true*. It has been shown that this problem is NP-complete. [11]

With its origins in Logic, which "as a science was invented by Aristotle" [12] in Ancient Athens, it is the oldest of all problems known to be NP-complete [11]. It has not only ties to Logic but also to other scientific fields like graph theory, computer science, and more. Because of these ties most practical NP-hard problems, or at least component problems of these, can efficiently be translated to SAT problems, resulting in an ever-increasing interest in good SAT solvers. [12]

More formally SAT can be defined as follows.

- $U = \{u_1, \ldots, u_n\}$ is the set of boolean variables.
- $T : U \mapsto \{\textit{true}, \textit{false}\}$ is the (partial) truth assignment for these variables.
- literals u and $\neg u$ (see Equation 8 for their evaluation under T)
- a clause C is the disjunction ($\vee$) of literals
- a formula ϕ is the conjunction ($\wedge$) of clauses

$$\begin{aligned} u \text{ is } \textit{true} \text{ under } T &\iff T(u) = \textit{true} \\ \neg u \text{ is } \textit{true} \text{ under } T &\iff T(u) = \textit{false} \end{aligned} \tag{8}$$

A formula ϕ is satisfied by T iff. *all* clauses $C \in \phi$ are satisfied by T. A truth assignment T satisfies C iff. *at least one* literal $u \in C$ is *true* under T. [11]

$$\{\{a \vee \neg b\} \wedge \{b\}\} \tag{9}$$

$$\{\{a\} \wedge \{\neg a\}\} \tag{10}$$

Fig. 3. examples for a satisfiable formula (9, by setting a to *true* and b to *true*) and an unsatisfiable formula (10)

Resolution based solvers are the first class of SAT solvers to be discussed here. The idea behind this approach is to eliminate variables from the formula via resolution and by this to find contradictions that prove the *unsatisfiability* of formula ϕ. Following are the rules of resolution:

- $\{u_i \vee V\} \wedge \{\neg u_i \vee W\}$ resolves to $\{V \vee W\}$ (with at least one of V and W not empty)
- $\{u_i\} \wedge \{\neg u_i\}$ resolves to $\{\perp\}$

To eliminate a variable u_i from the formula ϕ, ϕ is first augmented by any pair of clauses C_i, C_j with $u_i \in C_i$ and $\neg u_i \in C_j$. Then all clauses $C_i \in \phi$ with $u_i \in C_i$ or $\neg u_i \in C_i$ are removed from ϕ. [12]

If during any step of the resolution the empty clause $\{\perp\}$ is generated, ϕ is unsatisfiable. Additionally, this resolution is refutation complete, meaning it is guaranteed to produce the empty clause if ϕ is unsatisfiable [12]. Therefore this solver is guaranteed to classify the formula correctly as sat. or unsat. if ran until all variables are eliminated.

Local Search based solvers try to find solutions of the formula to show satisfiability. [11]

For this local search, a cost function c is defined over the space of truth assignments. The cost of a given assignment T_i is assessed by counting the clauses $C \in \phi$ that are not satisfied by T_i. Any global minimum T_i of this cost function c satisfies ϕ iff. $c(T_i) = 0$. [11]

To start of a random initial truth assignment T_0 is taken. In every step after that, T_i is improved by evaluating the cost of any assignment with Hamming distance 1 from T_i. The best neighbour of T_i becomes the next T_{i+1}, even if $c(T_{i+1}) > c(T_i)$. This is different from conventional local search algorithms but is essential for the search to not get stuck in local minima. Cook writes in his paper that "if restricted to improving steps, [local search] performs very poorly, and in practice steps which make no change to the score dominate the search" [11]. Despite this change to the search algorithm it still can get stuck in local minima. Therefore a maximum amount of search steps until a retry, and also a maximum amount of retries, are defined. After all retries are executed without finding a satisfying truth assignment the algorithm classifies the given formula as *unsatisfiable*. [11]

Note the key differences between the two solving approaches. The *Resolution* solver tries to prove unsatisfiability via contradiction. The *Local Search* solver looks for satisfying solutions. The former will give definite and proven answers for any formula and the latter only in case of a satisfiable one. It is because solvers like *Local Search* can only prove one of their answers one has to include the performance metric of *classifying accuracy* in his algorithm selection model.

2) The Algorithm Portfolio: SATzilla07 was built around an Algorithm Portfolio which is basically a selection mapping (see subsection II-A) for the SAT problem. A regression-based model was used, meaning for each solver a Ridge Regression (see subsubsection II-C.1) model was trained to predict its runtime, and the selection then made based upon those predictions. [10]

For each category of the SAT 2007 Competition (see subsubsection III-A.3) a separate model of SATzilla was trained to maximize its performance in each category [10]. The training data for each category consisted of the respective datasets from previous SAT competitions, for example, the model competing on the random category was trained on the combined random datasets from previous competitions. [10] Their overall instance collection held 4811 problem instances. 40% of these were used as training samples, 30% went into the validation set, and the remaining 30% were used to analyze SATzilla's performance (see subsubsection III-A.3). [10]

Touching back on the definitions of the Automatic Algorithm Selection problem (see subsection II-A) we have formulas of propositional calculus filling the problem space P and some candidate SAT solvers in the algorithm space A [10]. As for the feature space, a total of 48 features were hand picked as a preliminary set, among those were, for example, the number of clauses, the number of variables, statistics about the ratio of positive and negative literals in each clause, and some probing features for different solving approaches. See [13] for the entire list of preliminary features. Afterwards a *two phase forward selection* (see Equation II-C.1) was performed to construct the final set of features F [10]. Unfortunately, there was no specific definition of a performance measure mentioned in Xu's paper [10], but as he includes the speed of execution and classifying accuracy in his performance analysis a good guess might be that it is a mixture of at least those two.

To generate a labeled training dataset for the Ridge Regression models, a set of formulas were handpicked to be as representative of the practical use case as possible. Every formula was then solved by every candidate solver and their performances noted as future target labels. Lastly, the features of all training formulas were generated and coupled with the target labels to form the final training set for each algorithm [10]. In the worst case, this process of evaluating the performance of the seven used solvers on all 1925 training samples with a cut-off time of 1200 CPU seconds could have taken a total of 16170000 CPU seconds (187.2 days). Note though that only about 32% of instances were not solved within the time limit by any solver [10]. Consequently the runtime will have been below that, how far was not mentioned.

It is not guaranteed that every feature can be computed for every instance. Therefore formulas for which the feature computation failed were excluded from the training dataset, and the solver with the best average runtime on those formulas was put in place as a backup solver. [10]

As the last step before training two presolver were selected. A presolver is an algorithm that solves "a large proportion of instances quickly" [10]. To let the Ridge Regression models focus on hard instances (ie. they cannot be solved by the presolvers quickly) those instances were filtered out of the training set. [10]

With the final training dataset, the Ridge Regression models for every algorithm were trained as explained in subsubsection II-C.1.

The previously constructed Algorithm Portfolio is then used on a per-instance basis as follows: [10]

- Presolvers are run unconditionally for a short amount of time to catch easy problem instances.
- The feature vector is computed for the given instance.
- In case the feature vector could not be computed the instance is solved by the backup solver.
- Otherwise, the runtime of each algorithm is predicted by the model, and the expected best algorithm solves the instance.

The pre- and backupsolvers are optimizations that are tailored to SAT. They are described here to illustrate that optimizations can be made outside the Machine Learning models.

3) Performance Analysis: For the performance analysis of the previously described *SATzilla07* its results in the SAT 2007 Contest are reviewed. The Contest is held for three different categories:

- *Random* contains randomly generated SAT formulas.
- *Handmade* formulas are constructed to give a challenge to known solvers.
- *Industrial* formulas are typically very large and represent real-world applications.

Figures 4 to 7 are all built up the same. In these figures, SATzilla07's performance is compared to three component solvers (meaning not collective) and the oracle which simulates an upper performance (ie. lower runtime) bound of SATzilla07 by always choosing the optimal solver. On the left are the average runtimes over the entire category. The white bar on top of the SATzilla column is the average duration of the feature computation. On the right is the cumulative runtime distribution function for every solver. Runtimes of pre-solving, as well as the average feature computation, are indicated by the strips on the bottom.

In the random category (see Figure 4) SATzilla07 solved instances on average about three times faster than the best component solver. Its average runtime was only about 30% over the achievable optimum. In addition to that, it managed to solve nearly all problem instances within the time limit which is about 15% more than the fastest component solver. Of its solved problems, 94% were classified correctly. [10]

In the handmade category (see Figure 5) SATzilla07 was on average nearly twice as fast compared to the fastest component solver but also took twice the time of the oracle. It managed to solve about 95% of the given instances, again around

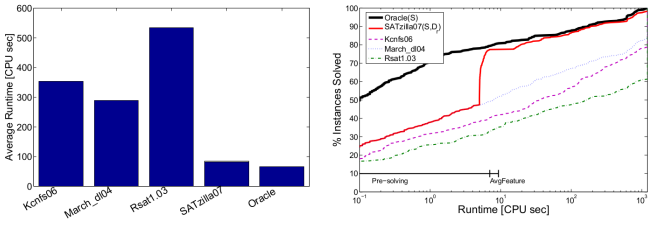

Fig. 4. Performance of SATzilla07 in the *random* category of the SAT 2007 competition, taken from [10]

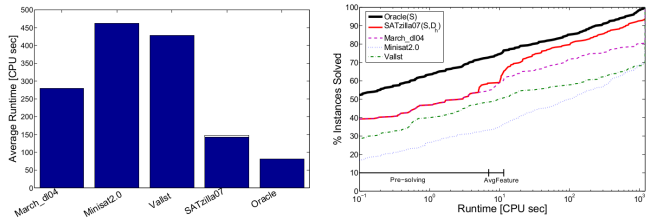

Fig. 5. Performance of SATzilla07 in the *handmade* category of the SAT 2007 competition, taken from [10]

15% more than the best component solver. The classification accuracy dropped to 70% which is its worst value between all categories. [10]

The industrial category (see Figure 6) is where SATzilla07 runtime-wise performed the worst relative to the fastest component solver, namely only around 23% faster. It took on average a little over twice the time to solve the problems compared to the oracle. This is also the only category where the feature computation with around 6% made up a considerable amount of the average solution time. Despite the longer execution times it still was able to solve around 95% of the given formulas, with 92% of those being classified correctly. [10]

SATzilla07's overall relative performance (see Figure 7) looks closest to the handmade category. It was twice as fast as the best component solver but also twice as slow as the oracle. It still managed to solve around 95% of problems within the time limit with the closest component solver being at around 75%. The overall classification accuracy lies by 78%. [10]

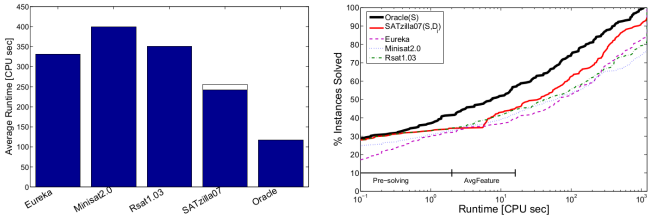

Fig. 6. Performance of SATzilla07 in the *industrial* category of the SAT 2007 competition, taken from [10]

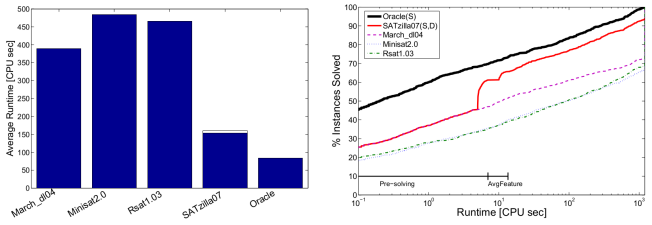

Fig. 7. Performance of SATzilla07 in the combination of all categories of the SAT 2007 competition, taken from [10]

B. Reinforcement Learning for Automatic Online Algorithm Selection

The second reference project of this research is the scientific paper "Reinforcement Learning for Automatic Online Algorithm Selection - an Empirical Study" written by Hans Degroote et al. He states that his motivation for this research came from observing that the machine learning based state of the art algorithm selection models left all the newly generated data during usage unused. Therefore he developed a Reinforcement Learning based (see subsubsection II-C.3) model that continues its training on this new data. [4]

A Regression Forest model (see subsubsection II-C.2 was used as a predictor for the performance of each algorithm and the selection was then made based on this prediction. The problem fields where this approach was tested came from ASLIB, which is a database containing several algorithm selection problem datasets [4]. These datasets included problem instances, instance features, and solving algorithms for each problem field. Included problem fields were, for example, different satisfiability problems, constraint solving, and container pre-marshaling. For the entire list, as well as details on each dataset see [13].

1) Solution Strategy: The Reinforcement Learning approach leaves it to the user whether or not to perform some initial training of the model. In this case, the decision was made to use 10% of all data as an initial offline training dataset [4]. Following the offline training is the online phase which used 80% of the dataset. Predictions for the performance of every algorithm are made, the optimal algorithm according to the current strategy (see subsection II-B) is selected and executed, and the performance of the selected algorithm, coupled with the instance features, is used to train its prediction model [4]. Finally, the remaining 10% of the data were used to validate the model after all of the training was done. [4]

Note that even though 90% of the overall data was used for training, no algorithm is trained with its performance information on all of those training instances, as during the online phase each instance was only solved by one algorithm.

Using so many different problem fields as a benchmark for the same algorithm meant that a raw machine learning model without outside optimizations was used as a selection mapping. [4]

2) Performance Analysis: To start this section, the figures used to represent the performance analysis are explained. Each algorithm is evaluated with a PAR10 score [4]. The PAR10 score is either the time it took the algorithm to solve a problem instance within the time limit, or the time limit multiplied by 10 if it failed to do so. This multiplication by 10 represents a penalty for not being able to solve the instance.

A state of the art reference performance is indicated by the red line in each figure. This performance was achieved by the regression forest based solver from LLAMA [14].

As an additional reference, the performance of a greedy-full-information model was included. It is trained with the performance information of every algorithm executing all previous instances. [4]

For the evaluation every score is normalized. Normalizing is done by awarding the virtual best solver (like the oracle for SATzilla07, subsubsection III-A.3) a score of 0. The score of 1 represents the single best solver's average performance for all instances.

The parameters of the experiments were kept the same for all experiments and are as follows:

- n_{min} for a regression tree: 5
- amount of trees for regression forest: 500
- LCB λ: 1
- ϵ-greedy ϵ: 0.05
- the minimal amount of data points to retrain model for an algorithm: 16
- amount of repetitions per experiment: 10

The first comparison was made between learning during the online phase (*greedy*) and not doing so (*greedyNL*). Both methods are trained during the offline phase. The average performance of the *greedy* method during the online phase is already better than the *greedyNL* one, with the *greedyFI* being ahead of both (see Figure 8).

In the validation phase, the lead of the *greedy* method over the *greedyNL* is further extended. It is noteworthy that the the *greedy* method comes close to the benchmark performance of LLAMA (see Figure 9). At the point of validation the *greedyFI* method is trained with the same information as the LLAMA selection mapping, hence the similar performance. [4]

Figures 8 and 9 visualize the comparison between the different strategies on how to solve the Contextual Multi-Armed Bandit problem (see subsection II-B). The first observation is that the *LCB* and the *greedy* methods performed nearly identically during the online training, as well as the validation phase. Secondly, algorithms selected by the *ϵ-greedy* strategy had a worse average runtime during the online phase but performed slightly better during the validation when compared to the *greedy* method. An improvement was observed for all learning strategies.

What this paper fails to mention is the time each run spent retraining its models. It would have made the comparison of the *greedy* and the *greedyNL* strategies much more useful. One desireable analysis could have been that the *greedy* method lost some time to the retraining of its predictive models,

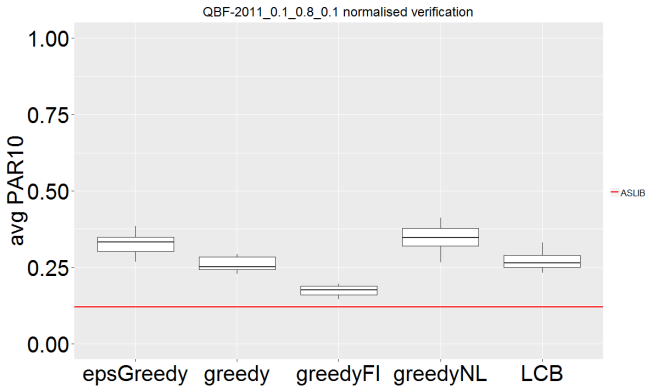

Fig. 8. Comparing the different exploration strategies, *greedyFI*, and *greedyNL* during the online phase. Taken from [4]

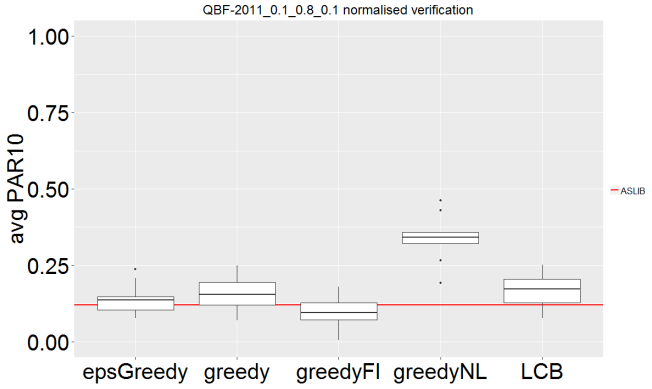

Fig. 9. Comparing the finished training of the exploration strategies, *greedyFI*, and *greedyNL* in the validation phase. Taken from [4]

but overtook the *greedyNL* method after n solved problem instances.

IV. Conclusion

The results of the first reference project show that supervised machine learning based methods are state of the art selectors for the offline automatic algorithm selection problem. In problem fields like propositional satisfiability its average performance is better than that of any single solver.

Furthermore the analysis of the second reference project shows some promise for a reinforcement learning based approach to training the selector to reduce the initial training overhead of supervised learning. However the given performance data for this project was lacking in the key areas to confirm this promise.

Future research could be invested to compare the methods in greater detail, mainly regarding the minimization of the overall execution time.

References

[1] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann, "Autopas: Auto-tuning for particle simulations," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rio de Janeiro: IEEE, May 2019.

[2] J. R. Rice, "The algorithm selection problem." *Advances in Computers*, vol. 15, pp. 65–118, 1976. [Online]. Available: http://dblp.uni-trier.de/db/journals/ac/ac15.html

[3] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," *AI Magazine*, vol. 35, pp. 48–60, 2014.

[4] H. Degroote, B. Bischl, L. Kotthoff, and P. De Causmaecker, "Reinforcement learning for automatic online algorithm selection - an empirical study," vol. 1649, 2016, pp. 93–101. [Online]. Available: https://lirias.kuleuven.be/retrieve/403396

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002. [Online]. Available: https://doi.org/10.1023/A:1013689704352

[6] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[7] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: The state of the art," *CoRR*, vol. abs/1211.0906, 2012. [Online]. Available: http://arxiv.org/abs/1211.0906

[8] A. Géron, "Hands-on machine learning with scikit-learn and tensorflow by aurélien géron," https://www.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch04.html, accessed: 18.6.2019.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[10] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for SAT," *CoRR*, vol. abs/1111.2249, 2011. [Online]. Available: http://arxiv.org/abs/1111.2249

[11] S. A. Cook and D. Mitchell, "Finding hard instances of the satisfiability problem: A survey," vol. 35, 01 2000.

[12] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.

[13] B. Bischl, P. Kerschke, L. Kotthoff, M. T. Lindauer, Y. Malitsky, A. Fréchette, H. H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, and J. Vanschoren, "Aslib: A benchmark library for algorithm selection," *CoRR*, vol. abs/1506.02465, 2015.

[14] L. Kotthoff, "LLAMA: leveraging learning to automatically manage algorithms," *CoRR*, vol. abs/1306.1031, 2013.