Kernel Regression for Planning Heuristics

Caelan Garrett
6.867 Final Project
caelan@mit.edu

December 5, 2014

Abstract

Modern automated planning revolves around forward state-space heuristic search. These planners guide their search with powerful domain-independent heuristics that estimate the distance to a goal state by efficiently solving related, but simplified, planning problems. This approximation trades off efficiently with approximation quality. In the past 15 years, the Planning and Learning community has looked at developing domain-independent learning strategies that compute domain-specific knowledge to augment the search process. These strategies include learning planner configurations, macro-actions, cases, policies, and heuristic functions themselves. We explore different methods of learning a heuristic function in a domain-independent fashion. In particular, we focus on kernelized regression methods to compare symbolic objects such as states and approximate plans inspired from string kernel methods. Finally, we experimentally evaluate the predictive power of these learned heuristics in the blocks world domain by comparing the training error and the resulting search performance.

1 Introduction

State-of-the-art planners almost all use forward state-space heuristic search to solve planning instances. The heuristics and control strategies used in these searches are the key forces that have caused these methods to have success. Heuristics approximate the distance to the nearest goal in the state space, and if this approximate is strong enough, searches can move in the direction of the decreasing heuristic value to quickly find a satisfying plan. These procedures are all defined in a domain-independent manner, meaning that they do not rely on particular information about the class of problems they are currently solving.

But because heuristics are still approximations, there is a fair amount of room to improve their estimates. In particular, an active area of research is learning for planning. In regard to heuristics, this involves learning a new heuristic or more often a correction to an existing heuristic that improves its estimate of the remaining distance. This learning process is also typically domain-independent although it results in a heuristic that is specifically tailored to the learned domain. The learning process is frequently able to compensate for common sequences that cause a loss in approximation quality that occur in a particular domain.

The particular heuristics that form the basis for our learning features and kernels are HSP's h_{add} and h_{max} [1] as well as FF's h_{FF}[11]. These both use the idea of relaxed planning to approximate the true planning task. This is done by removing the delete effects of operators. This approximation is appealing because planning with it can be done in polynomial time by computing a Relaxed PlanGraph (RPG). From the RPG, the FF heuristic derives a relaxed plan which solves the planning task under the delete relaxation. As we will later see, these relaxed plans are central to our learning process.

In contrast to previous work, we focus on kernelized learning procedures for making these regression predictions. In particular, we draw on ideas from string kernels and other symbolic kernels to capture inner products between planning objects. These objects include states, sets of literals, and even relaxed plans. We present several ways to construct kernel products for each of these features.

We experimentally evaluate our heuristics in the standard STRIPS blocks world domain [6]. This domain has four types of operator schema where B_1 and B_2 are blocks: $\mathit{Pick}(B_1)$, $\mathit{Place}(B_1)$, $\mathit{Stack}(B_1, B_2)$, and $\mathit{Unstack}(B_1, B_2)$. Planning tasks involve rearranging blocks from a grounded start configuration to a goal

configuration. These experiments capture the performance of the different learned heuristics at both predicting the distance to goal and improving planning efficiency. Finally, we experiment with different learning methods themselves for approaching these types of problems.

2 Related Work

Much of the work on learning to improve planning performance has been done in the last 15 years. Partially responsible for the expansion of research are ICAPS Planning and Learning workshops as well as learning tracks in ICAPS International Planning Competitions. Learning to improve planning generally falls into five categories: learning effective planner configurations, learning macro-actions, learning cases, learning a policy, and learning heuristic functions. This paper deals with the fifth learning problem, but we will review the other methods as well.

Researchers have developed many different heuristics, search methods, and control strategies each with possibly many parameters. Learning effective planner configurations involves identifying which combination of these is most effective for a given domain. For example, the selective max planner trains a Naive Bayes classifier to predict, for a given state, goal, and domain, which heuristic will be the most accurate for estimating the distance to the goal [4]. The FD-Autotune planner contains a collection of planning tools and uses an iterated local search on the space of configurations to configure its search strategy to most efficiently operate on a particular domain [5].

Macro-actions are new operators that combine several operators that are frequently used in sequence. They assist the search process by skipping several steps in the search process. But they have a utility tradeoff caused from the increased branching factor. Macro-FF, the first system to use these, uses an iterative weight adjustment process to identify the most useful macro-actions [2].

Learning cases is the process of storing a database of complete plans to problems in a domain and adapting them to apply to new problem instances. This requires computing the similarity of a current planning problem to previous problems. This matching is done via heuristics or learning. To learn matchings, the OAKPlan system trains kernelized classifiers that compare the planning tasks using graph similarity measures on a planning encoding called the Planning Encoding Graph [16]. This paper also seems to be the only paper to use kernels in a learning for planning context. However, the problem and kernelized methods are different than the ones we will consider.

Learning a policy involves learning a mapping from each state and goal pair to a set of successor operators. Early learning systems used Inductive Logic Programming and Genetic Programming to learn the policy and greedily apply the policy to find a plan. These methods hope to approximate the optimal policy without needing to explore the entire state-space of a domain [12]. However, learning a strong enough approximation to greedily apply the policy proved difficult. Instead, current research involves learning rough policies to augment the search process by frequently identifying the most relevant search branches, speeding up planning. This typically involves training a classifier [19][3] to select helpful operators.

Finally, learning a heuristic function is often valuable because domain specific learning can improve the estimate of the distance to a goal by identifying where domain-independent heuristics overestimate or underestimate the actual distance. Early work attempted to train regressors on the state and goal pairs directly but was unable to achieve substantial generalization. Later systems derive additional features from these pairs such literals and actions that appear on the relaxed plan graph computed by the FF Heuristic [18][20]. But these systems operate in the feature space.

3 Features and Kernels

The primary objective of this project is to explore how different features and kernels impact estimation of the true distance to the goal and overall search process. Although we focus on the blocks world domain, we restrict ourselves to features and kernels constructed in a domain independent way with the intention that these methods could be applied to planning and learning domains. For all the learned features, we subtract the value of $h_F F$ from the training examples and re-add it in during prediction time. This improves the performance of the heuristics by centering the learning process around a decent initial guess. Without this,

some of the weaker features and kernels make absurd predictions on the wrong order of magnitude when they have insufficient data.

3.1 Heuristic Features

The idea heuristic features is to use features constructed from other domain-independent heuristics. Planning researchers have devised many powerful heuristics each with strengths and weaknesses. The combination of many different heuristics may allow for them to work together to compensate for states where individually one may be weak. The heuristics used here to construct features are

1. h_{goal} - counts the number of goal literals in the state
2. h_{add} - the summed relaxed planning distance of goal literals measured by the additive cost of actions
3. h_{max} - the maximum of the level of relaxed plan graph graph goal literals
4. h_{FF} - the length of a relaxed plan to reach the goal state

3.2 In Common Kernel

The most obvious kernel approach is to compute the number of objects in common in a set. This can be used for initial states and goal states in the features. For states s_1, s_2,

$$\phi_x(s) = \mathrm{I}(x \in s)$$
$$\kappa(s_1, s_2) = \sum_x \phi_x(s_1)\phi_x(s_2) = |s_1 \cap s_2|$$

This computation can be summed together for the start and goal state to make one kernel. Note that the in common kernel can compare other sets as well. Other comparison sets include the set of actions on the relaxed plan, effects derived from actions on the relaxed plan, and goals on the relaxed plan.

3.3 Subsets Kernel

For all the sets previously listed, a kernel can also compute the number of subsets in common.

$$\phi_A(s) = \prod_{a \in A} \mathrm{I}(a \in s)$$
$$\kappa(s_1, s_2) = \sum_{A \subset \mathcal{A}} \phi_A(s_1)\phi_A(s_2) = \prod_{a \in \mathcal{A}} (1 + \mathrm{I}(a \in s_1)\mathrm{I}(a \in s_2))$$

3.4 Approximate Plan Kernels

The following kernels all involve computing the inner product between the relaxed plans computed by the FF Heuristic. The intuition is that obtaining generalization is difficult when learning from just start states and goal states. By generating additional symbolic features through a relaxed plan, learning algorithms have more leverage to identify where the relaxed plan deviates from the true distance to goal. The structure of these kernels is inspired from kernels used on strings, trees, and directed, acyclic graphs (DAGs).

Note that one can compute approximate plans to be used in kernels from other heuristics than h_{FF}. Many heuristics can be modified to return the approximate plan that is considered when deriving the heuristic value. This even applies for non-relaxed planning heuristics such as the FastDownward Heuristic and the Context-Enhanced Add Heuristic [9][10]. Without going into the mechanics of these heuristics, a planner could extract an approximate plan that is not necessarily a relaxed plan which can be used instead for learning.

A key idea for these kernels involves the similarity of operator instances. This involves defining an inner product between operators themselves inside of the kernel. The most obvious operator inner product is to return one if the inputted operators are the same, otherwise outputting zero. But this can run into

generalization problems if there are many possible operator parameterizations. In the blocks world domain, a training example may have never seen the operator $\textsc{Stack}(B_1, B_2)$ for particular blocks B_1, B_2, but we would hope that the learning algorithm could still generalize predictions from a planning problems that involved stacking.

This brings about a second possible operator inner product. Instead, equate operators that are from the same operator schema. Because this set is significantly smaller than the set of parameterized operators, fewer training instances are needed to obtain generalizability. Future work could look at defining and scoring a similarity inner product between operators themselves that is a combination of equating operators that have the same parameters or operators from the same schema. For example, a kernel could weight the similarity by the number of parameters in common between the operators of the same schema.

This similarity could also take into consideration the possible symmetry of parameters. For example, the blocks in blocks world all have equivalent function, so it would be sensical to consider their names relative to other blocks in the plan. Note that the for plan databases, many of these kernel matrices can be computed efficiently through hashing and avoiding processing inner products with training examples that have no common operators. But when more generous operator inner products are introduced, more training examples will have nonzero kernel products reducing the sparseness of interaction.

Another helpful addition to the approximate plans is to add a special start state and goal state pseudo-operators inserted at the beginning and end of the plan respectively. These provide additional context regarding the plan. These pseudo-operators are formed by constructing start operators that have effects equal to the start state without preconditions and goal operators that have preconditions equal to the goal state with no effects.

3.5 Approximate Plan Subplan Kernel

The first kernel we consider attempts to mimic the behavior of substring algorithms in string kernels. For example, the k-spectrum algorithm computes its inner product by counting the common number substrings of length k in the two strings. For a string s, there are $|s| - k - 1$ substrings of length k. An algorithm for counting the number of sequences in common between s, t can be be done by sorting the possible substrings and performing a sorted list comparison. This runs in $O((|s| + |t|) \log(|s| + |t|))$. Other methods run more efficiently in $O(k(|s| + |t|))$ by using suffix trees or hashing [13].

When translating these ideas to planning, we don't have a fixed sequence of operators but a partial ordering of operators respecting preconditions and effects. This ordering can be used to construct a Relaxed PlanGraph DAG where edges are the partial orders. Instead of defining contiguous operators through flattening the plan, we define operators as contiguous if they are on adjacent levels of the RPG.

By modifying the algorithms used for k-spectrum kernels, we can compute the number of k length subplans in the RPG. The modifications use the idea that there may be $O(n^k)$ possible k length subplans in k consecutive levels of the RPG. Instead of computing these explicitly to compare with the other subplan, we use dynamic programming over each operator in each level to compute the number of common paths in $O(n)$. From there, we sum the number of subplans from every combination of length k intervals giving a total runtime of $O(n^3)$. Future work would look at further dynamic programming to improve this algorithm.

A kernel can also count all subplans of any length by summing the result of the k length subplans algorithm for all feasible k. Because longer common subplans are more rare, the kernel can discount short plans using a parameter *lambda* that is discussed in the next section.

3.6 Approximate Plan Subsequence Kernel

Another way to compute a kernel is to flatten each approximate plan into a sequence that respects the partial ordering in the plan. Once flattened, we can apply standard string kernel techniques that compute the number of common subsequences in the full sequence. Note that these subsequences are not constrained to be contiguous. These algorithms count subsequences discounted by a factor of λ^g where $\lambda \in [0, 1]$ and g is the number of gaps in the sequence being counted. In our application, gaps are not a problem, but we would like to add additional weight to longer subsequences over shorter ones. Thus, sequences of length l are multiplied by $1/\lambda^l$ to increase their weight.

Established algorithms to count the number of subsequences use dynamic programming. The standard algorithm uses two phases of dynamic programming. To count the number of subsequences in common of size n, the algorithm makes a subproblem for each subsequence of size $i \in 1...n$. For each i, it introduces another dynamic programming problem to count the number of subsequences of length i for the different lengths of input strings s, t. That is, it considers subproblems (x, y) such that $x \in 1...|s|, y \in 1...|t|$. The computation of these subproblems builds off the result of the size $i-1$ subproblem. This results in an algorithm of runtime $O(n|s||t|)$ [14] although more efficient algorithms have been developed that use sparse dynamic programming, tries, or suffix trees [15].

3.7 Approximate Plan Partial Order Kernel

The final kernel we consider directly considers the partial order relations in the approximate plan. The partial order $O_1 \prec O_2$ is created when O_1 has a effect that first satisfies a precondition of O_2. Note that other operators could also have effects that satisfy the same precondition of O_2, but we restrict to counting just one of these for compactness. Once this set is constructed, we can apply the kernel that counts the number of partial orders in common for two sets.

3.8 Combining Kernels in Feature Space

Finally, one might want to use distance information from multiple kernels or features to make a prediction. Combining multiple kernels in the true dual version of a learning problem is nontrivial and an active area of research [8]. But we can combine them trivially by considering them as features in the primal form of the learning problem instead of as true kernels. Therefore, for every training example we augment the feature vector with the kernel inner products as features, allowing us to combine these measures of similarity.

4 Learning Algorithms

I used scikit-learn's implementations of regression algorithms for prediction. I also experimented with mlpy, another python machine learning library, but it had less support for kernelized algorithms and equivalent least squares regression performance to scikit-learn. I was interested in both primal learning with design matrices and dual learning with kernels. The first models considered were Ordinary Least Squares Regression (OLS) and the regularized Ridge Regression (RR). Although scikit-learn doesn't support these kernelized methods, recall that the kernelized example weights can be obtained by performing the same matrix computations as in the primal from. K is the Gram Kernel Matrix such that each entry is the kernel product of the feature vectors corresponding the the entries indices.

$$\alpha = (K(X, X) + \lambda I)^{-1} Y$$
$$y(x) = \alpha^T K(x, X)$$

To contrast with Least Squares Regression and use a model that incorporates kernels differently in its dual formation, I explored Support Vector Regressors.

4.1 Support Vector Regression

Support Vector Regressors (SVR) are adaptations of SVMs from classification problems to regression problems [17]. One of the main types of SVRs is the ϵ-SVR. It attempts to maintain the maximum margin property of SVMs that arrises from the hinge loss function by defining an ϵ distance around the regression value where predictions are not penalized. Here, ϵ is a user specified parameter like C. This gives rise to the ϵ-insensitive loss function which accumulates loss linearly for predictions greater than ϵ from the desired value. The primal form of the soft-margin SVR is the following. Note that it follows the same structure as SVMs except it includes two decision variables ξ_i, ξ_i^* for the soft-margin instead of one.

$$\text{minimize } ||w||^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$$

$$\text{subject to } \begin{cases} w^T x^{(i)} + w_0 & \geq & y^{(i)} - \epsilon - \xi_i \\ w^T x^{(i)} + w_0 & \leq & y^{(i)} + \epsilon + \xi_i^* \\ \xi_i, \xi_i^* & \geq & 0 \end{cases}$$

Its corresponding dual also has two decision variables $\alpha_i + \alpha_i^*$ per training example, but it still admits a kernel through an inner product between feature vectors.

$$\text{maximize } -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*)\kappa(x^{(i)}, x^{(j)}) - \epsilon\sum_{i=1}^{N}(\alpha_i + \alpha_i^*) + \sum_{i=1}^{N} y^{(i)}(\alpha_i - \alpha_i^*)$$

$$\text{subject to } \begin{cases} \sum_{i=1}^{N}(\alpha_i - \alpha_i^*) & = & 0 \\ \alpha_i, \alpha_i^* & \leq & C \\ \alpha_i, \alpha_i^* & \geq & 0 \end{cases}$$

This dual SVR formulation allows us to experiment with kernels in an maximum margin framework where the products are precomputed as an Gram Matrix input to scikit-learn's SVR.

5 Training Data

In order to do learning, we need to compute a dataset of training examples in the domain. These examples are tuples $< s^{(i)}, g^{(i)}, d^{(i)} >$ corresponding the state, goal, and distance to a goal states respectively [12]. The tuple $x^{(i)} =< s^{(i)}, g^{(i)} >$ are the features that can be expanded to create additional features and $y^{(i)} = d^{(i)}$ are the target distance predictions.

There are several different approaches to generating training examples. ICAPS Planning and Learning Competitions typically give a specified set of training examples representing typical planning tasks in the domain [12]. Other frameworks gives planners the domain representation and allow planners a fixed amount of time to explore the domain themselves and derive example plans. Another approach that may be more useful in application would be to remember the set of previously specified planning tasks and use them as training examples. This is appealing if previous tasks are representative of future planning tasks because the training set is smaller set of examples that are most relevant to learning future search control instead of potentially a large collection of diverse planning tasks in the domain that may not all be relevant. The method of obtaining training examples and number training examples needed are both strongly dependent on the complexity of the domain. Domains with large state spaces or a large number of actions will need many training examples unless most queried planning tasks interact with a small portion of the state space.

Additionally, training examples can be labeled with different distances which may affect performance. A non-optimal planner used to generate a training example may overestimate the minimum distance to a goal state if many satisfying states at different distances exist. Incorporating training examples that may not reflect the minimum distance to the goal will likely give learned heuristics that experimentally prove "less admissible" than training examples with the optimal distance making them less appealing for approximately optimal planning or even feasible search itself. However, optimal search is often prohibitively time expensive, so distances are often computed by a non-optimal search despite this potential decrease in training example quality. In a similar regard, some features extracted from the state goal pairs could have multiple possible values. For example, the FF heuristic approximates the optimal relaxed plan cost. This approximation admits many possible costs depending on the order in which actions are linearized. One could instead attempt to compute the optimal relaxed plan cost, but that problem is NP-hard making it not worthwhile.

I focused on learning for blocks world domains where size is a function of the number of blocks in the world. Instead of using training examples from blocks world problem instances used in ICAPS competitions, I generated my own training examples by exploring the state space of the domain. Because blocks world is an invertible domain, meaning that the state space is undirected, I generated a set of states reachable from

Heuristic	Average Testing Error
h_{add}	26.8385
h_{max}	45.7516
h_{FF}	14.8385
Heuristic Features	6.96932
Start and Goal Kernel Features	25.5022
Heuristic and Start and Goal Kernel Features	23.0394
Start and Goal In Common Kernel	25.5867
Start and Goal Subsets Kernel	10.8754
Relaxed Plan In Common Kernel	6.40037
Relaxed Plan Schema In Common Kernel	6.10168
Relaxed Plan Subsets Kernel	7.57267
Relaxed Plan Schema Subsets	7.27341
Relaxed Plan Add Effects In Common Kernel	10.508
Relaxed Plan Add Effects Subsets Kernel	8.84676
Relaxed Plan Subplans Kernel	9.10168
Relaxed Plan Subplans Kernel $\lambda = .5$	9.62891
Relaxed Plan Schema Subplans	9.10168
Relaxed Plan Subsequences Kernel	10.8104
Relaxed Plan Subsequences Kernel $\lambda = .5$	11.2192
Relaxed Plan Schema Subsequences Kernel	5857.75
Relaxed Plan Partial Order Kernel	9.81815
Relaxed Plan Schema Partial Order Kernel	**5.44768**

Figure 1: Features or Kernels versus Average Testing Error

a seed state and sampled initial states and goal states from this set. Note that goal states are allowed to be partially specified states, so I would often drop a number of required literals chosen uniformly at random. I choose my seed state to be the state where all blocks are all on the table. For a small number of blocks less than six, I explored the state space via a Breadth-First Search. For larger domains, I instead used random walks. Once initial and goal states were sampled, I computed the distance to goal by performing a greedy best first search using the FF heuristic and helpful actions resulting in a not necessarily optimal distance to goal.

6 Experiments

The first experiment in Figure 1 measured the average testing error when predicting the distance to goal using the features and kernels previous described. Each learning model was trained using Ordinary Least Squares Regression. Training data and testing data was constructed using 50 random walks on blocks world with eight blocks generating a set of approximately 300 states. The data was split 50/50 for training and testing data.

Note that testing errors above h_{FF}'s testing error indicate that the learning process resulted in a worse heuristic than if it made zero correction to h_{FF}. The table indicates that using purely the start and goal state in a kernel gives poor predictive performance. Also, increasing the weight of longer subplans or subsequences by setting $\lambda = .5$ has little or a negative effect. Additionally, using the operator schema comparison instead of the operator instance improves performance for most of the relevant heuristics. This might be because there are too many classes of parameterized operator instances to generalize well.

The best performing method of the features and kernels was the Relaxed Plan Schema Partial Order Kernel. It had about a third of the error of h_{FF}. Other notable heuristics were Heuristic Features and the Relaxed Plan In Common Kernel.

Although producing accurate estimates of the distance to the goal is usually a good bellwether for the performance of a planner using the heuristic, the next experiment in Figure 2 directly measures this performance. We selected a subset of learned heuristics that performed well in the previous experiment. The

Heuristic	States Expanded	Heuristic Evaluations	Length
h_{FF}	12.2727	50.9636	7.61818
Heuristic Features	10.8909	46.4000	6.67273
Relaxed Plan In Common Kernel	12.0909	55.5091	6.85455
Relaxed Plan Schema In Common Kernel	11.2727	53.5455	6.87273
Relaxed Plan Partial Order Kernel	**8.94545**	**38.4909**	**7.49091**
Relaxed Plan Schema Partial Order Kernel	15.9454	66.4727	8.36364

Figure 2: Planning Performance of Learned Heuristics

Learning Algorithm	Testing Error
OLR	5.32076
RR $\lambda = 0.001$	5.1414
RR $\lambda = 1$	5.14095
RR $\lambda = 1000$	4.99
SVR $C = 0.001, \epsilon = 0.01$	6.0933
SVR $C = 0.001, \epsilon = 0.1$	6.0893
SVR $C = 0.001, \epsilon = 1$	6.00665
SVR $C = 1, \epsilon = 0.01$	5.03157
SVR $C = 1, \epsilon = 0.1$	4.97945
SVR $C = 1, \epsilon = 1$	**4.97846**
SVR $C = 1000, \epsilon = 0.01$	5.0795
SVR $C = 1000, \epsilon = 0.1$	5.0228
SVR $C = 1000, \epsilon = 1$	4.99375

Figure 3: Learning Algorithm versus Average Testing Error

data collection and division is the same as in the previous experiment. We used a greedy best first search with each heuristic, recording the number of states expanded, heuristic evaluations, and resulting length of the plan. We do not report runtime because these problems are small enough such that computing additional features made the overall search slower than h_{FF} due to unoptimized implementations.

Note that the Relaxed Plan Partial Order Kernel that uses similarity by the operator instance instead of with the scheme performed best. This is contrary to our prediction given the previous table. This might be because although it has more prediction error, it is able to better order which states have a lower distance to the goal. Note that it searches, on average, only about 1.5 more states than the plan length which demonstrates its power.

Given that Relaxed Plan Partial Order Kernels seem to work best with ORL, we investigate the testing error when using a different learning method but this same kernel. We use the operator schema version because it obtained a lower testing error despite worse performance than its counterpart. In particular, we look at comparing differently parameterized RR and SVRs. Figure displays the results. An SVR with moderate regularization and a large insensitivity boundary performed the best. The insensitively boundary likely provides some regularization itself by pushing the support vector margins out. But the takeaway is that regularization is still useful in learning for planning contexts.

7 Conclusion

Modern domain-independent heuristics and control methods have proved successful at making heuristic search an efficient strategy for planning. But planning efficiency can be further increased by applying domain-independent learning strategies to speed up planing for a specific domain. These methods include learning configuration tuning, macro-actions, cases, policies, and heuristic functions.

We looked at learning heuristic functions by learning a correction to h_{FF}. In addition to standard learning features, we emphasized kernelized learning algorithms for symbolic planning objects which don't appear to have been previously studied in this context. Kernels that simply compared start and goal states

performed poorly. A better strategy was to use kernels that compared approximate plans for solving the problem, for example relaxed plans. This led to the most powerful kernel we achieved that compares the similarity of partial orders in a relaxed plan. This was seen through analysis of the raw prediction testing error and resulting planner performance. While the version that compared only operator schema achieved the best testing error, the version that compared full operator instances obtained the best search performance. Finally, a regularized version of an SVR performed the best on our domains indicating that regularization is important in this learning context.

8 Future Work

The primary extension of this work is to apply these learned heuristics to planning competition domains and compare their performance against established systems. Another interesting class of domains for application are continuous robotic manipulation planning problems [7]. In these problems, operators have a substantial base time cost that arises from geometric primitives. This makes a search that explores even only 100 states costly. Thus, even the slightest heuristic performance improvement will be noticeable. Because blocks world shares some of the typical pitfalls for domain-independent heuristics as manipulation domains, the same learning techniques are likely to work well.

As mentioned in the related work, there are many different learning problems related to improving planning. Application of the features and kernels discussed here may improve performance of some of these techniques. In particular, kernelized classification using approximate plans may improve resulting policies used to determine helpful operators for search.

While this paper looked at feasible planning, cost sensitive planning, where operators have non-unit costs, is important in some applications. The kernel inner products involving operators should extend to these cost additions. In these cases, the similarity can be multiplied by the cost of the operator, which is usually constant over an operator schema, to weight similarity of plans based on the cost. Consider a cost sensitive planning domain where heuristics approximate the cost to goal install of distance. Some approximate plans may have the same length with wildly different cost. The corresponding scaling of inner product will allow kernelized learning algorithms to proportionally scale the predicted heuristic cost.

A final extension could look at modeling distance heuristics with count data regression. The distance to goal is an integer value, so it might be worthwhile to explore regression techniques like Poisson Regression or Negative Binomial Regression that output an integer prediction by modeling the function as a discrete random variable. These methods could improve heuristic prediction, but they also might hurt search performance. Real value predictions inherently are unlikely to have have the same value causing the search to have an ordering on the estimated distances of successor states with no ties. This may be useful for when a prediction would round to an incorrect value, but the search control still selects the best successor action because the real valued prediction is lower for the best successor state. A future study could help understand counting regression performance in a planning context.

References

[1] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33, 2001.

[2] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res.(JAIR)*, 24:581–621, 2005.

[3] Tomás De La Rosa, Sergio Jiménez Celorrio, and Daniel Borrajo. Learning relational decision trees for guiding heuristic planning. In *ICAPS*, pages 60–67, 2008.

[4] Carmel Domshlak, Erez Karpas, and Shaul Markovitch. To max or not to max: Online learning for speeding up optimal planning. In *AAAI*, 2010.

[5] Chris Fawcett, Malte Helmert, Holger Hoos, Erez Karpas, Gabriele Röger, and Jendrik Seipp. Fd-autotune: Domain-specific configuration using fast downward. In *PAL 2011 3rd Workshop on Planning and Learning*, page 13, 2011.

[6] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[7] Caelan Reed Garrett, Tomás Lozano-Pérez, and Leslie Pack Kaelbling. Ffrob: An efficient heuristic for task and motion planning. In *International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2014. URL `http://lis.csail.mit.edu/pubs/garrett-wafr14.pdf`.

[8] Mehmet Gönen and Ethem Alpaydın. Multiple kernel learning algorithms. *The Journal of Machine Learning Research*, 12:2211–2268, 2011.

[9] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246, 2006.

[10] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In *ICAPS*, pages 140–147, 2008.

[11] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal Artificial Intelligence Research (JAIR)*, 14:253–302, 2001.

[12] Sergio Jiménez, Tomás De la Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(04):433–467, 2012.

[13] Christina S Leslie, Eleazar Eskin, and William Stafford Noble. The spectrum kernel: A string kernel for svm protein classification. In *Pacific symposium on biocomputing*, volume 7, pages 566–575, 2002.

[14] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *The Journal of Machine Learning Research*, 2:419–444, 2002.

[15] Juho Rousu and John Shawe-Taylor. Efficient computation of gapped substring kernels on large alphabets. In *Journal of Machine Learning Research*, pages 1323–1344, 2005.

[16] Ivan Serina. Kernel functions for case-based planning. *Artificial Intelligence*, 174(16):1369–1406, 2010.

[17] Alex J Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and computing*, 14(3):199–222, 2004.

[18] Sung Wook Yoon, Alan Fern, and Robert Givan. Learning heuristic functions from relaxed plans. In *ICAPS*, 2006.

[19] Sung Wook Yoon, Alan Fern, and Robert Givan. Using learned policies in heuristic-search planning. In *IJCAI*, volume 7, pages 2047–2052, 2007.

[20] Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718, 2008.