Learning Adaptation to Solve Constraint Satisfaction Problems

Yuehua Xu[1], David Stern[2] and Horst Samulowitz[2]

[1]Oregon State University, USA xuyu@eecs.oregonstate.edu
[2]Microsoft Research Cambridge, UK {dstern,horsts}@microsoft.com

Abstract. Constraint-based problems are hard combinatorial problems and are usually solved by heuristic search methods. In this paper, we consider applying a machine learning approach to improve the performance of these search-based solvers. We apply reinforcement learning in the context of Constraint Satisfaction Problems (CSP) to learn a value function, which results in a novel solving strategy. The motivation underlying this approach is to solve previously unsolvable instances.

1 Introduction

Constraint-based Problems such as SAT (Satisfiability), QBF (Quantified Boolean Formulas), CSP (Constraint Satisfaction Problems) are hard combinatorial problems. Because of their exponentially large search space, these problems are in general computationally intractable and are usually solved by heuristic search. Different heuristics or solving strategies can result in very different performances across benchmark families (e.g., [1], [2], [3]).

In recent years, various approaches [1], [4], [3] have applied machine learning techniques to solve constraint-based problems. The goal of learning is to automatically adapt the heuristics or solving strategies in a way that the performance of the solvers can be improved. In general, there are two types of adaption methods: static adaption and learning adaption. Static adaption corresponds to automatically selecting the solving strategy from a set of available approaches and clinging to this strategy throughout the solving process (e.g., SAT[1], QBF [2]). While improving the robustness of a solver, static adaption is not able to solve instances that were not solved by any existing strategy. In contrast, learning adaption learns a new different strategy which can potentially solve previously unsolvable instances (e.g., SAT[5], QBF[4], CSP[3]). Motivated by the success of previous work, we consider learning adaption for CSP by utilizing reinforcement learning techniques to discover new solving strategies.

2 CSP Background

A CSP is defined as a pair $\langle V, C\rangle$, where V is a set of variables, each with a finite domain of values, and C is a set of constraints, expressed as relations over the variables in V. In general, CSP instances are solved by making a sequence of

decisions that select a variable at a time and assign a value from its respective domain. A value ordering heuristic is used here in addition to the variable ordering heuristic. After each decision is made, the value of the selected variable is propagated and the domains of the remaining variables are updated accordingly as well as the consistency of constraints is checked, resulting in backtracking of the solver if a conflict is detected. We call an instance s reduced by propagation its sub-instance s'. A complete assignment resulting in no conflicts is a solution to a CSP.

3 Learning Adaption

By directly interacting with the solver, learning adaption is able to discover a new strategy and therefore has the potential to solve instances that cannot be solved with the currently available approaches. The existing learning techniques for constraint-based problems include supervised learning [4] (QBF) and reinforcement learning [5] (SAT). However, to the best of our knowledge for CSP the only existing work focuses on learning a mixture of heuristics which is then statically applied during the entire solving process [3] [6]. Here we describe how general reinforcement learning [7] can be applied in the context of CSP with the goal to adapt the search process dynamically at each decision point based on the structural properties of the current sub-problem to be solved.

3.1 Problem Setup

The search-based CSP solver can be formulated as a reinforcement learning task as below:

- A set of states S. Each state $s \in S$ is an instance or sub-instance of CSP. The state space corresponds to the search space of the CSP.
- A set of actions A. Each action $a \in A$ is a variable ordering heuristic function.
- Transition function $T : S \times A \to S$. Each transition corresponds to a decision made when solving a CSP instance. For example, given a pair (s, a), we can use the variable ordering heuristic a for the (sub-)instance s to select a variable. After the variable is selected and assigned a value (by some fixed value ordering heuristic), the (sub-)instance s is changed to another (sub-)instance s', which is exactly a transition.
- Reward function R. The goal of a CSP solver is to solve instances. In the goal states where a solution is found, a reward is assigned.

This formulation can be extended to also include value ordering functions, but for the sake of simplicity, we only consider variable ordering heuristics for now. After setting up the task, we can employ the Q-learning approach (e.g., [8],[7]) to learn an optimal policy $\pi^* : S \to A$, that is, to learn which variable ordering heuristic should be used given a (sub-)instance.

3.2 Function Approximation and Q-learning

Q-learning is a reinforcement learning technique that learns an optimal action-value function $Q^*(s, a)$, giving the expected value of taking action a in state s. Since the state space here is exponentially large, we approximate $Q(s, a)$ by a function based on the features of a state s.

$$Q(s, a) = w^a \cdot f(s) \tag{1}$$

where w^a is a weight vector for the action a and $f(s)$ is a feature vector representing the state s. Assuming that we have all functions $Q(s, a)$ for all actions, the optimal policy is $\pi^*(s) = argmax_a Q^*(s, a)$. Algorithm 1 gives a sketch of the Q-learning algorithm in the context of CSPs.

Initialize all weights to 0;
repeat
  Use a fixed CSP instance as input;
  $t = 0$ (Decision Level set to 0);
  repeat
    Choose a heuristic $a_t = h_i$ using the current policy;
    Store current feature vector $f(s)$ labeled with h_i at level t;
    Select a variable based on h_i, assign its value and propagate, then check the consistency and backtrack if needed;
    $t = t + 1$ (Increase Decision Level);
  until *solver terminates (either solving the problem or timing out)* ;
  if *a solution is found* **then**
    for *j = t-1 to 0* **do**
      s is the state at level j, s' is the state at level $j + 1$;
      $w^{a_j} = w^{a_j} + \alpha \cdot (R_j + \gamma \cdot max_{a'} Q(s', a') - Q(s, a_j)) \cdot f(s)$
    end
  end
until *no more episodes* ;

Algorithm 1: Reinforcement Learning Algorithm [7] in the context of CSP.

The algorithm tries to solve an instance with the current policy (initially randomly selecting a heuristic at each level t). At each decision level t the algorithm stores the feature vector $f(s)$ representing the current state s and it also keeps track of which variable ordering heuristic h_i it utilized in the state s. Hence, when reaching a solution we have a list of pairs each consisting of a feature vector labeled by the heuristic employed on this (sub-)problem. This training data is then used to update the weights and the policy accordingly (taking into account the learning rate α and the discount factor γ). The entire process is repeated until a number of episodes is reached always starting with the same CSP as input.

4 Preliminary Results and Discussion

While in general several parameters need careful tuning when employing reinforcement learning in a particular application (e.g., how to balance the tradeoff

between exploitation and exploration; learning rate), even more care has to be taken when applying it in the context of CSP. Here we briefly highlight a few key issues.

As stated earlier a reward is assigned when the solver terminates. However, the basic approach of a binary reward (1 in goal states, 0 elsewhere) is not applicable as we verified in experiments. In most applications the discounted propagation of the binary reward allows for distinguishing between qualitatively different states (e.g., in both states the problem is solved but different runtimes occur) but this is not necessarily the case with CSP. For example, in CSP an instance can be solved by making only a few decision, but it can require a much larger runtime than the solution that performs many more decisions due to different costs of propagation. Therefore, we used the following weighted reward: $R = 1 - (runtime/timeout)$. However, dealing with the reward in this fashion does not seem to yield a completely satisfactory solution to this problem either since learning did not converge to the minimum runtime in some cases. It might be necessary to further emphasize the differences in runtime in the reward function. Note, however, that Q-Learning with function approximation as it is described here is in general not guaranteed to converge [9].

Another question that arises in this context as well is if it would be sensible in the context of CSP (and tree search in general) to assign higher rewards to decisions made closer to the root of the tree rather than the other way around as it is the case in reinforcement learning (high rewards at the leaf nodes). This is based on the intuition that decisions made at the top of the search tree have in general an essential impact on the performance of a search-based solver.

Another main problem is the fact that one of the most effective value ordering heuristics is based on randomization. Clearly, this introduces a high variance to the learning process since a heuristic applied in the same state twice can result in a different performance due to the randomized value selection. It remains to be shown that the learning process is sufficiently robust to deal with randomization.

We implemented Algorithm 1 within a CSP solver and conducted first preliminary experiments. We used two variable ordering heuristics with non-random value selections and applied the learning algorithm on one single instance. While both heuristics on their own are only able to solve a given instance in more than 3 seconds, the learning process is able to learn in which context which heuristic is most effective and the resulting interleaving of both heuristics achieves a runtime of about 0.01 seconds. We also experimented with three variable ordering heuristics, and the learning algorithm was for instance able to conclude that one of the heuristics is superior to the others on the given problem which was indeed the case.

While these preliminary results are quite promising several important open questions remain. For instance, does the learned approach generalize to other problems (at least within a benchmark family)? Does the learning process also converge for a larger range of heuristics? How robust is the learning process with respect to randomization?

References

1. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for sat. Journal of Artificial Intelligence Research **32** (2008) 565–606
2. Pulina, L., Tacchella, A.: A multi-engine solver for quantified boolean formulas. In: CP. (2007) 574–589
3. Epstein, S., Petrovic, S.: Learning to solve constraint problems. In: ICAPS-07 Workshop on Planning and Learning. (2007)
4. Samulowitz, H., Memisevic, R.: Learning to solve qbf. In: AAAI. (2007) 255–260
5. Lagoudakis, M., Littman, M.: Learning to select branching rules in the dpll procedure for satisfiability. Electronic Notes in Discrete Mathematics (ENDM), Vol. 9, LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (2001)
6. Petrovic, S., Epstein, S., Wallace, R.: Learning a mixture of search heuristics. In: CP-07 Workshop on Autonomous Search. (2007)
7. Irodova, M., Sloan, R.H.: Reinforcement learning and function approximation. In: FLAIRS. (2005) 455–460
8. Watkins, C.: Learning from Delayed Rewards. PhD thesis, Cambridge University (1989)
9. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. The MIT Press (1998)