Solving a Problem Domain with Brute Force Goal Regression

Stephen Murtagh and Paul E. Utgoff

Computer Science Department
University of Massachusetts
140 Governor's Drive, Amherst, MA 01003

Abstract

We present an algorithm that solves for an optimal policy in domains defined by a class of automata without enumerating the underlying state-space. This is done by constructing equivalence classes of problem states and regressing them through the automaton. This class of automata describes fully-observable, non-stochastic, sequential domains in which one or more agents interact. We apply this technique to the domain of Connect Four and present the results of that experiment. We show the amount of memory required for our technique was six orders of magnitude smaller than the amount that would be required for a standard retrograde analysis approach.

1 Introduction

The problem of how to represent and solve problem domains has been studied in several different areas. In reinforcement learning, for example, a Markov Decision Process is used to represent a domain and is typically solved by finding an optimal policy, while game theory solves games by finding a set of equilibrium strategies. For each way of representing a domain and its solution there are a variety of techniques available for finding a solution. One property that these techniques share is that in order to find an exact solution, they typically either directly examine each state in the state-space or search the space of all possible policies. Both of these types of approaches, however, require enumerating the entire state-space of the domain.

Algorithms that work by enumeration are limited in the size of the domains they can solve. This is not an issue if a domain is so complex that the only way to describe it is to list every state and action, but if can be describe far more succinctly, then one would expect that there is a better method for solving it. If a domain can be

represented compactly, it follows that there is a large amount of regularity that an algorithm should be able to exploit.

Our work seeks to leverage the descriptions of the domains in order to solve them without enumerating the underlying state-space. Toward this end we define a class of automata that can compactly represent a variety of domains. We then present a new technique that combines goal regression and abstraction to solve relatively large, easily described domains.

Our technique works by regressing sets of states through an automaton describing the domain. Each of these sets contain all states in the domain that are equivalent with respect to the optimal outcome and the optimal next action. When regression terminates, the equivalence classes defined by these sets will have fully partitioned the state-space. Together, the equivalence classes describe the optimal policy for every agent, or equivalently, a set of strategies that form a subgame perfect equilibrium. By describing these sets with boolean formulae over features of the domain, they are more space efficient than listing the individual states and by performing operations on entire sets at once, the amount of computation required to find a solution can be reduced.

Finally, we demonstrate the effectiveness of this technique on the domain description of the game Connect Four. This is a two-player game played on a seven by six board in which the players alternate placing pieces, each of which must be placed at the lowest open cell of a column. The first player to achieve four adjacent pieces on the same row, column, or diagonal wins, and the game terminates. This game has approximately 10^{14} states (van den Herik, Uiterwijk & van Rijswijck, 2001), making a standard brute-force solution infeasible on a single desktop machine. Our approach finds an optimal policy for each agent this domain in a reasonable amount of time while consuming far less memory than an enumeration based approach would.

2 Problem

In order to solve large domains, we need a representation that allows us to describe the domain without listing the individual states. We have chosen to define a class of automata that will be used to represent domains. These automata are deterministic finite automata that are augmented with variables and propositional pre-and post-conditions on the transitions. This representation can be used to describe a large class of problems easily, in a way that will allow us to perform regression in a straight-forward manner.

A domain theory is represented by an automaton defined as:

$$M = (Q, \Sigma, k, \delta, s, F) \tag{1}$$

where Q is a finite set of states, Σ is a finite alphabet, k a finite set of variables, or features, each of which can take on any symbol from Σ, $\delta : Q \times \Sigma^k \rightarrow Q \times \Sigma^k$ is a deterministic transition function, s is the start state, and F is a set of goal states with rewards. Finally, each state in Q has a single agent associated with it. That agent is the only agent that can ever act in that state.

Throughout this article, we refer to two types of states: states in a domain and states of the automaton. In order to alleviate the potential confusion, we always refer to states in the problem domain as problem states. Automaton state, goal state and initial state all refer to automaton states rather than the domain's state.

States in the state-space of the problem are described by k variables, each of which can take on $|\Sigma|$ values. In the automaton, these features determine the set of transitions that can be taken from a given state. The alphabet also contains a special character, ε, which is used only to denote a variable whose value is irrelevant.

A combination of automaton states and features is used to represent the domain. This simplifies the definition of a domain by separating the actual states of the domain, which are represented by the features, from the structure imposed on the domain, i.e. when agents act, what transitions are available, and when agents receive rewards.

From any given automaton state, multiple transitions may be possible depending on the current variable settings. A constraint is imposed that in any automaton state, all transitions whose preconditions can be simultaneously satisfied must move to the same automaton state. This allows us to reason about the transitions an agent could not have taken.

The constraint does not limit the set of domains that can be expressed. Any automaton that violates the constraint can be converted to one that does not, by adding an additional variable that is used to store the successor state and an automaton state to select the successor state.

The goal of our work is to produce an algorithm that can analytically compute an optimal policy for a given domain without enumerating the state-space. Analytically solving a domain allows us to find an exact solution, and by not enumerating the state-space we can solve otherwise intractably large domains. This is done by generating a hierarchy of equivalence classes which specify the optimal decisions for all of the equivalence classes of problem states. These classes are constructed from the automaton directly.

As described, the automaton can only specify fully observable, non-stochastic do-

mains with one or more agents. The issues that arise from removing these limitations are discussed below.

3 Related Work

We have developed a method for finding an optimal policy in the automaton, which is a compressed, deterministic, fully observable MDP. A large body of reinforcement learning techniques, such as value iteration (Bellman, 1957) and policy iteration (Howard, 1960) could be used instead. Applying such techniques directly, however, would require searching the entire state space. We would like to take a brute force approach that does not have this limitation, and we are motivated, in part, by Etzioni's (1993) method for carrying out a static analysis in a problem domain.

Recent work in relational reinforcement learning provides techniques to leverage compact relational representations similar to the one described above (Kersting, van Otterlo & De Raedt, 2004). These methods, however, still rely on exhaustive lookup tables to perform value function updates, which requires them to iterate over the entire state space and therefore restricts them to small state spaces.

Reinforcement learning techniques on factored MDPs, such as that by Boutilier et al (2000), provide a method for finding optimal policies in large state spaces. This work, however, is largely limited to single-agent domains or co-operative multi-agent domains (Guestrin, Venkataraman & Koller, 2002).

One benefit of the automaton specification is that systems with multiple agents can be modeled easily. Recent work on adapting reinforcement techniques to learning for multi-agent systems focuses on stochastic games, matrix games, or games with hidden information (Tesauro, 2003; Littman, 1994). In these games, every pure strategy, or deterministic policy, may be dominated, requiring expensive calculations or approximation techniques to find optimal mixed, or randomized, strategies. Our approach, however, focuses on the smaller class of non-stochastic, sequential games with no hidden information, which means that an optimal pure strategy can always, in principle, be found.

The techniques set forth in this paper are similar to those used in planning, specifically the idea of goal regression (Waldinger, 1976). A planning domain consists of an initial state, a goal state and a set of operators that alter the state. In goal regression, a set of goal states is regressed through the operators to find the corresponding set of pre-images, which in turn are regressed though the operators until the initial state is reached. Furthermore, all operators must be one to one, meaning that they can be inverted. Our approach performs regression in a similar manner, except that multiple goal states, whose value may not be the same, are used, and

multiple agents may interact. The objective of our work is to find an optimal policy rather than a single solution path.

The approach we have taken resembles Schoppers's (1987) universal plans, which partition the state space according to which action will take the agent close to the goal. Universal plans take the form, if condition *P* arises while trying to reach the goal, then the appropriate action is *A*. The equivalence classes we generate perform the same function from the perspective of the agent applying them. Schoppers's work, however, does not deal with multiple goal states or agents.

4 Approach

Our approach is to generate descriptions of sets of states, which describe membership in the equivalence classes, and to regress these sets backward through the automaton. All members of a single class are equivalent with respect to the optimal outcome and the optimal next transition.

Each goal state defines a single outcome, and it has a tuple of rewards associated with it, one for each agent. An outcome is considered better than another outcome from the perspective of a particular agent if the reward for that agent is higher or the outcomes are the same and the distance from the current set of states to the outcome is shorter.

In effect, one equivalence class is created for each automaton state with a reward, i.e. the goal states. We assume no intermediate rewards. Each of the initial equivalence classes describes the set of problem states that are zero steps from a particular goal state.

These descriptions are then regressed one step through the automaton, by finding the pre-image of each description for each transition that could have generated it (Waldinger, 1976). If there are multiple transitions through which a description could legally be regressed, then a copy of the original is regressed though each transition. The set union of these describe the set of all problem states that are one step removed from the outcome of the original descriptions.

After each new description is generated, it is compared to the previously generated set descriptions and removed if it is a subset of an existing set. This process is called *subsumption checking* (Hollunder & Nutt, 1990). If the subsumed set describes the same outcome as the subsuming set, then this is essentially duplicate checking. However, if the subsumed set describes a worse outcome, from the perspective of the acting agent, then this set ascribes a suboptimal outcome to the described states and thus must be discarded. These retained new set descriptions are then regressed and checked in a breadth-first manner until no further regressions can be performed.

Finally, when the regression is finished, the equivalence classes fully partition the state space and describe the optimal outcome from any point in the state space. An agent can then use these classes in the forward direction by finding the class with the best outcome that contains the current state and taking the associated optimal transition. The number of classes generated depends greatly on the specifics of the automaton, but for the domain tested, Connect-4, the number of equivalence classes is seven orders of magnitude smaller than the size of the state-space. In terms of memory, our approach consumed almost 240 MB as compared to 20 terabytes required when storing the outcomes of five problem states in each byte, a typical approach used in retrograde analysis. This is a reduction of approximately six orders of magnitude in terms of memory required.

4.1 The AUTRE Algorithm

The AUTRE algorithm (AUTomaton REgressor) implements the approach we have described. It maintains a list of descriptions of equivalence classes, which are stored as tuples of the form (s, o, ψ, Φ). The automaton state in which the formula applies is denoted by s, and o is the optimal outcome. The positive constraint ψ is a logical formula that describes what must be true in order for a state to be a member of this set, and Φ, the set of negative constraints, is a set of logical formulae that must not hold for members of this set. Finally, ψ and all $\phi \in \Phi$ are conjunctions of literals, and together they describe the set of all problem states contained in the class. All states for which $\psi \wedge \bigwedge_{\phi \in \Phi} \neg\phi$ is true are members of the class and all other states are not.

The above formulation for equivalence classes was chosen because it is the simplest formulation we have found that allows regression to be performed. In order to know what transitions can potentially be applied, the automaton state in which a class applies must be known. The outcome is needed so that classes can be compared in terms of a rational agent's preference between them. Finally, ψ and Φ are needed to define membership in the class and are separate in order to speed regression.

The algorithm proceeds as follows. First, a tuple for each goal state is created with $\psi = (v_1 = \varepsilon \wedge v_2 = \varepsilon \wedge ... v_k = \varepsilon)$ and $\Phi = \emptyset$, where ε is the special value used to mark an irrelevant variable. This means that settings of the variables in the goal state are unconstrained. The tuples describe the set of states that will reach a given goal state and any agent acting in a goal state can trivially reach that goal state. As the tuples are regressed they will become more constraining.

The tuples are regressed back through the automaton in a breadth-first manner. To regress a tuple (s, o, ψ, Φ), all transitions to state s whose post-conditions are satisfied by ψ are considered. If there is no such transition, no regression is performed, otherwise a copy of the tuple is regressed though each applicable transition.

p	Pre-condition	Post-condition	p'
$v_1 = b$	$v_1 = a$	$v_1 = b$	$v_1 = a$
$v_1 = \varepsilon$	$v_1 = a$	$v_1 = b$	$v_1 = a$
$v_1 = b$	$v_1 = a$	$v_1 = \varepsilon$	$v_1 = a$
$v_1 = b$	$v_1 = \varepsilon$	$v_1 = b$	$v_1 = \varepsilon$
$v_1 = \varepsilon$	$v_1 = \varepsilon$	$v_1 = \varepsilon$	$v_1 = \varepsilon$
$v_1 = a$	$v_1 = a$	$v_1 = b$	$\perp$

Fig. 1. Update rules for regressing clause p through an operator with the listed pre and post conditions

Regressing through a transition $t : (s', (v'_1, v'_2, ..., v'_k)) \rightarrow (s, (v_1, v_2, ..., v_k))$ produces a new tuple (s', o, ψ', Φ'). Conjunctions of literals, such as the positive or negative constraints, are updated by regressing each literal in turn. Figure 1 shows the update rules for regressing a literal through an operator. At any time, there is only one true, non-epsilon literal for each variable. Also, the pre- and post-conditions of a transition are considered to be conjunctions of literals. Therefore, if the post-condition of a transition has a different true literal for a particular variable than the formula being regressed, as in the last line of Figure 1, the literal in the formula is replaced with FALSE. Otherwise, regressing a literal is equivalent to changing the value for a particular variable.

In addition, any transition t' from state s' to $s'' \neq s$, must not be available if the automaton is being traversed in the forward direction, since we assume that at any point, there is only one possible successor automaton state. Therefore, the pre-conditions of all such t' are added to Φ to produce Φ'. Any of these negated clauses where the value of a variable is different from the value in the positive clause, and neither value is ε, can immediately be discarded, as they will never affect the truth of the formula as a whole. For example, in the formula $v_1 = a \wedge \neg(v_1 = b)$, $\neg(v_1 = b)$ can be dropped because v_1 cannot have two different values.

Consider the example of regressing $\psi = (v = b) \wedge (u = c)$ and $\Phi = \emptyset$ from state B to state A, in the automaton shown in Figure 2. By the above update rules, the new positive constraint ψ' will be $(v = a) \wedge (u = a)$. Since we regressed from B, by our assumption the transition to state C must not have been available and therefore, its pre-conditions must not have been satisfied. The pre-conditions $(v = d) \wedge (u = e)$ are then added to Φ, the set of negative constraints on the membership of the equivalence class.

Next, ψ' and Φ' need to be checked for consistency, which is equivalent to verifying that $\psi' \wedge \bigwedge_{\phi' \in \Phi'} \neg\phi'$ is satisfiable. Because ψ' and all $\phi' \in \Phi'$ are conjunctions of literals, checking that ψ' does not force any individual ϕ' to be true can be done cheaply. All inconsistent tuples are discarded.

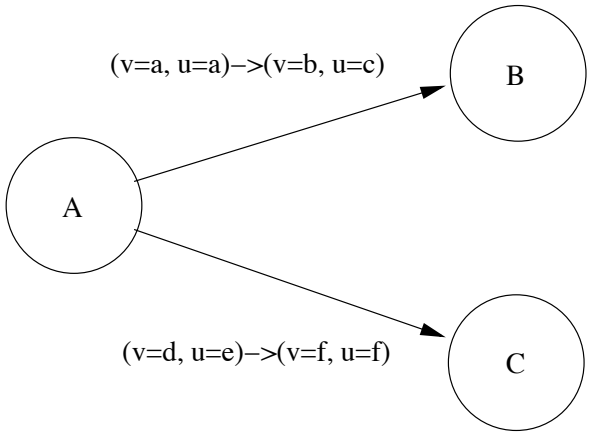


Fig. 2. A sample automaton.

Finally, subsumption checking is performed. The newly generated tuples must be checked against previously generated tuples. If the tuple x describes a subset of the tuple y and does not lead to a better outcome than y, then x is said to be subsumed by the y and can be discarded. A better outcome is defined to be an outcome with a higher value or an outcome which has the same value but can be reached in fewer steps. A tuple that describes a worse outcome will never be used by a rational agent and can be safely discarded.

Let $f = \psi \wedge \bigwedge_{\phi \in \Phi} \neg\phi$ be the formula in a previously generated tuple and let $g = \psi' \wedge \bigwedge_{\phi' \in \Phi'} \neg\phi$ be the formula in a newly generated tuple. If f is at least as general as g and has at least as good an outcome then g is subsumed by f and thus can be removed from consideration. Similarly, if g subsumes f then the tuple containing f is marked as invalid and is no longer considered. This will also make any children of the f, the older tuple, invalid, but they are not updated immediately, because they will be caught when g's children are generated.

Consider the case for which regressing a particular negative constraint through a transition produces FALSE, denoted $\perp$. That clause will no longer impact the logical formula that contains it, as can be seen in Figure 1. Similarly, if regressing the positive constraint results in $\perp$, the formula can never be satisfied and does not need to be regressed any further.

The situation in which a tuple could be legally regressed through multiple transitions can equivalently be handled by creating one new tuple whose constraints are an OR over all constraints generated by regressing through each individually. This OR is instead handled implicitly by creating multiple tuples to simplify the consistency check. When no more regressions can be performed, all states from which a goal state is reachable are members of at least one equivalence class.

Figure 3 shows an example of regression in the domain of Connect Four. The dark gray pieces belong to one player, the light gray 'pieces', which are the same color as the background, represent those cells whose values are unknown or unimpor-

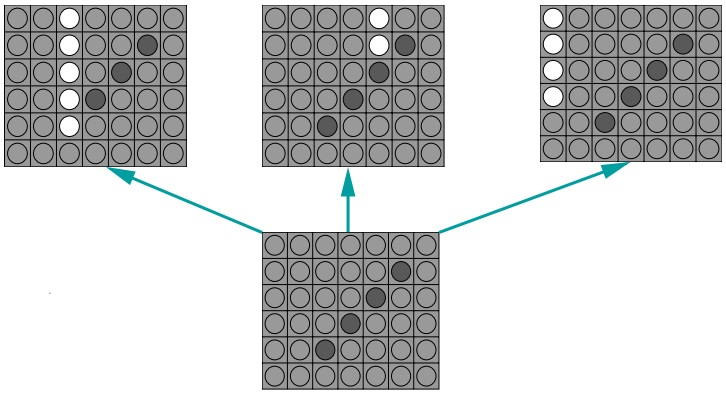


Fig. 3. Example regression from the domain of Connect Four

tant, and white represents empty spaces. The second player's pieces are not shown. Because Connect Four has the constraint that pieces must be placed on the lowest unoccupied cell in a column, when a piece is removed in the course of regression, then all cells above the piece must have been empty.

4.2 Efficiency Improvements

Although the AUTRE algorithm can be more efficient than enumerating the state space, it can still be expensive in terms of both space and time. We have made two additional improvements that can greatly reduce the memory consumption and, to a lesser extent, the run-time.

The first improvement comes from the observation that no rational agent will select a particular action if a better action is available. In our framework, a rational agent decides on an action by finding the equivalence class with the best outcome that includes the current state. Therefore, any given set description will only need to be evaluated when all descriptions corresponding to better outcomes are false.

This is implemented in the form of a decision list. When a new description has been generated, it is compared to each element on the list, which is ordered, by construction, according to the distance to and the value of the outcome. A property of decision lists is that each formula is evaluated in the context in which all formula above it on the list do not hold. This means that parts of a formula could be redundant, since they may also be true in the context the formula will always be evaluated in, i.e. the preceding formulae on the list. These redundant pieces can be removed, saving space.

Formally, the process works as follows. For each automaton state, there is an associated decision list of ordered formulae. Each formula is considered in the context of the negation of all preceding formula on the list. If the entries on a particular list are $f_1, ..., f_n$ then a formula f_k is equivalent to $f_k \wedge \bigwedge_{i=1}^{k-1} \neg f_i$, because no f_i can be

true in the context of f_k.

When a new tuple is generated, its formula (ψ', Φ') is compared against those on the list corresponding to the automaton state in which the new tuple applies. When it is compared against the ith formula in the list (ψ_i, Φ_i) if $\Phi_i = \emptyset$ and $\psi_i = \phi'$ for some $\phi' \in \Phi'$ then ϕ' can be removed from Φ'. This technique allows AUTRE to remove redundant information from the set descriptions with only a small computational cost, because the new set already must be compared to each old set to check for subsumption.

Decision lists are only one possible way of representing boolean formulas. AUTRE was also implemented using Binary Decision Diagrams (Bryant, 1986) in lieu of of decision list. BDDs are frequently used in circuit verification to represent boolean functions. In AUTRE, however, BDDs performed significantly worse than decision lists both in terms of CPU time and memory consumption. This seems to be a consequence of encoding multi-valued variables as boolean variables. The types of formulas used by AUTRE may also have contributed.

The second improvement to the basic algorithm is the representation of clauses. The number of variables and the range of possible values is specified in the domain description, therefore each conjunctive clause can be described by $k * |\Sigma|$ bits, where each bit indicates whether or not a variable can take on a particular value. Although there are more compact representations, this allows literals to be updated using only a few bitwise operations that can operate on a number of variables in parallel.

Depending on the number of symbols and the specifics of the machine used, multiple variables can be tested or updated in parallel. The domain of Connect Four, for example, can be represented using 42 variables which take on values from a four symbol alphabet, including ε, allowing all variables to be stored in four 32-bit integers. Operations can then be applied to the integers, which is equivalent to applying the operator to 10 variables in parallel in this example. In general, this will improve efficiency when Σ is small, as more variables can be packed into a single integer.

5 Experiments

We implemented the AUTRE algorithm, including the efficiency improvements, and ran it on the game of Connect Four. The purpose of this experiment was to show the feasibility of finding exact solutions to large domains.

Connect Four was chosen for two reasons. First, it is a non-stochastic multi-player game of perfect information. Also, it has a relatively large state-space, approximately 10^{14} (van den Herik, Uiterwijk & van Rijswijck, 2001), but it can be described compactly by a recursive domain theory with 42 variables, 8 automaton

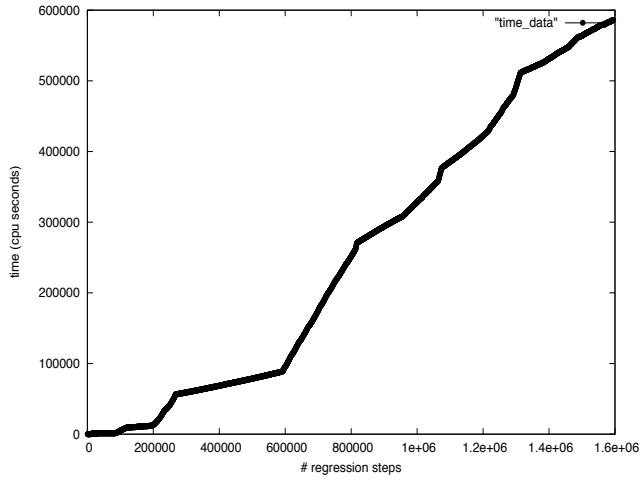


Fig. 4. Number of CPU seconds vs. number of regressions performed

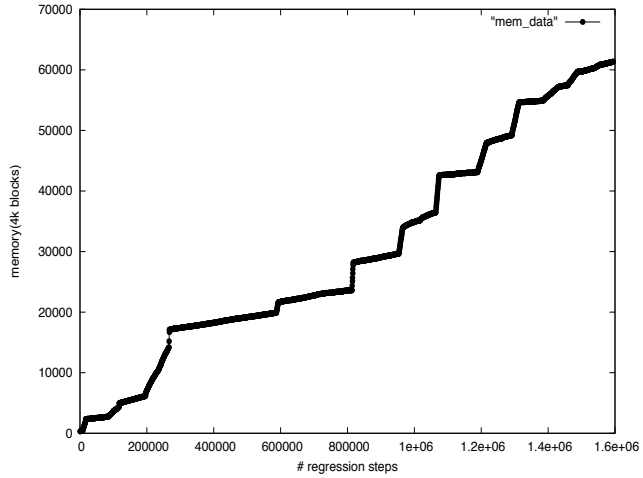


Fig. 5. Memory usage vs. number of regressions performed

states, and 273 transitions. AUTRE completed the game of Connect Four in less than 7 days of computation and 1.6×10^6 regression operations. Thus each regression step was, on average, equivalent to solving 62 million problem states and AUTRE computed the equivalent of almost 200 million states per second. Just 240 megabytes were required to store the equivalence classes generated by the regression process and thus can fit in the RAM of the average machine.

Figures 4 and 5 show the CPU time and memory consumption respectively. The running time, as shown in Figure 4 grows approximately quadratically with the number of regression steps. This is expected since each time a regression is performed, the resulting formula must be checked against all previous formula.

Figure 5 shows that the amount of memory used appears to grow logarithmically with the number of regression steps, although this is uncertain. Our conjecture is that this is due to both the subsumption checking and the decision lists. As computation progresses, more equivalence classes are generated but those classes tend to

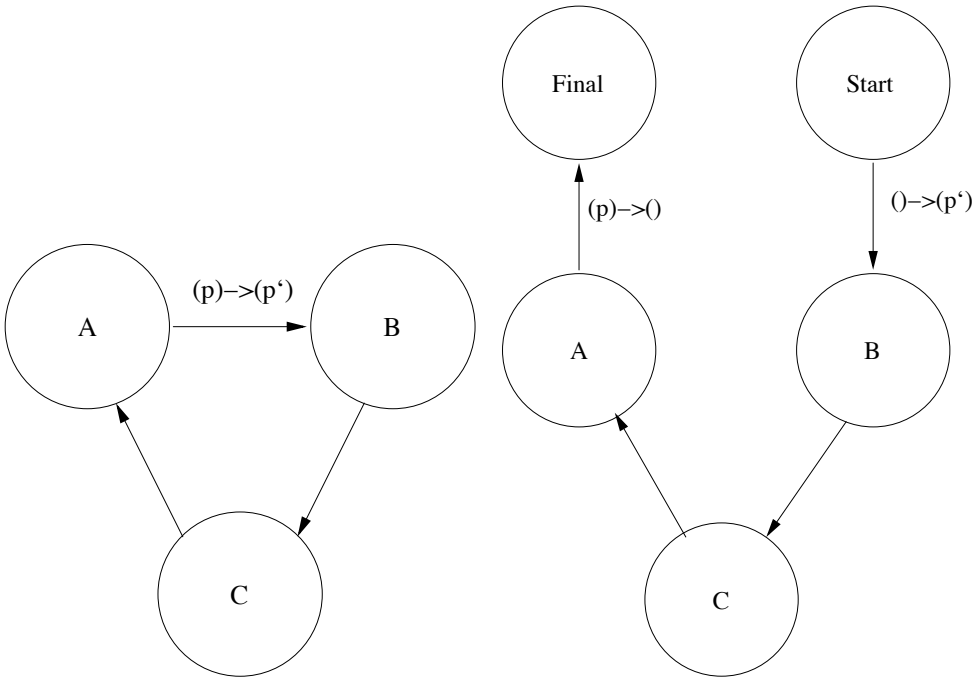
contain fewer instances than older ones and are more often subsumed. Finally, the use of decision lists means that newer formula are evaluated in a more constrained context than older formula, which allows shorter formula to be stored. This was observed in the the solution of Connect Four.

6 Discussion

There is currently no brute-force solution of Connect Four in the literature, although it has been solved previously, however, using knowledge-based techniques (Allis, 1988). AUTRE's input is simply an automaton encoding the rules of the game. Our investigation of Connect Four shows that our technique significantly reduces the size of the problem by more than a factor of sixty million from one hundred trillion states to three million equivalence classes. More importantly, our approach is domain general and does not require enumerating the state space, which allows for a faster and more compact solution process.

The bulk of the running time is spent doing subsumption checks. Whenever a new tuple is generated, it must be compared to previously generated tuples. Currently, old tuples are stored on a decision list, one list per automaton state. Each decision list contains all tuples that apply in the corresponding automaton state. Subsumption could therefore be sped up by further dividing the lists, so that fewer tuples would need to be compared. A data structure that allows for more efficient lookup could also be used, but the formula length reduction gained from using decision lists would be lost.

The efficiency of AUTRE is also dependent on the specifics of the automaton. For the domain of Connect Four, the obvious formulation of one variable for each square on the board and a transition for each possible move worked well. However for some domains the use of propositional logic in the automaton can make description difficult. For example, in the domain of checkers, a player wins if the opponent has no available moves. Therefore, the formula describing the states one step from a win for one player is the conjunction over the negation of all possible opponent moves, approximately 500. When this is regressed one step, hundreds of new formula which are almost as long are created. Thus the amount of memory needed grows much faster for Checkers than it does for Connect Four where it is much easier to describe a win. One technique we are exploring to remove this expense is decomposition of the domain.



(a) Before decomposition (b) After decomposition

Fig. 6. Decomposition on transition $(p) \rightarrow (p')$

6.1 Domain Decomposition

Not all domains can be solved quickly with AUTRE. Some domains are simply too large or complex for the techniques described above to work quickly. A solution to this problem is to decompose the domain into small, tractable pieces. The pieces can then be stitched together to solve the complete domain. This idea is similar to the concept of subgoals and temporally abstracted actions, such as options (Sutton, Precup & Singh, 1999), in reinforcement learning.

One possible method for decomposing domains in AUTRE is to break the automaton along a set of transitions. In other words, given an automaton and a group of transitions, define a new automaton with a new start state, which has transitions with no pre-conditions and the post-conditions of the chosen transitions, and a new final state reachable by transitions with the pre-conditions of the chosen transitions and no post-conditions. Figure 6(a) shows a sample automaton before composing on the transition $(p) \rightarrow (p')$, while Figure 6(b) shows the resulting automaton. No other transition is affected by the decomposition.

Using this technique, a domain encoded by an automaton is broken into several sub-domains. When a particular set of transitions is taken, the agents move to a different sub-domain. Furthermore, each sub-domain is described by the same automaton,

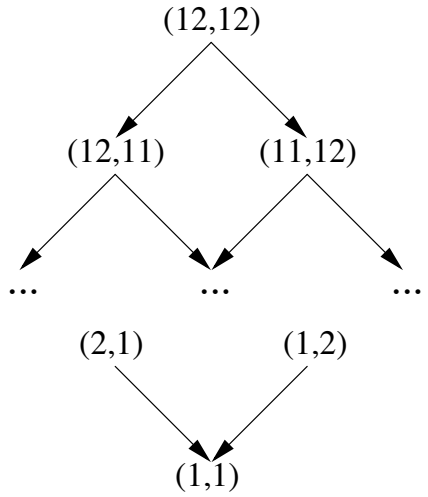


Fig. 7. Example domain decomposition of checkers. Tuples show the number of pieces for the players.

which means that an optimal policy in one sub-domain will be the optimal policy of almost any sub domain. The exceptions are the first and last sub-domains, where the agents use the initial and final states of the original automaton.

Although the proposed technique produces a single automaton for all subproblems, a domain could equivalently be decomposed into several different automata, and the AUTRE algorithm could be run on each one to obtain the optimal policy. Either way, the solutions to the subproblems must be reassembled into a solution for the original domain.

Combining the optimal policies for the sub-domains can be done by constructing a new abstract automaton that describes the connections between the sub-domains. The states in this new automaton would correspond to different sub-domains and the variables would be the same as the original domain's. Finally, for each equivalence class membership formula that is applicable in the start state of a particular sub-domain a new transition is constructed whose pre-condition is that formula and whose post-condition is the final variable settings in that sub-domain using the optimal policy.

Consider the game of checkers; one simple set of transitions on which the automaton could be decomposed is those corresponding to piece captures. Based on this selection, the new automaton would describe how to play optimally from one capture to the next. Then by constraining the number of pieces in each sub-domain, the problem would be decomposed into the network of connected sub-domains shown in Figure 7. This decomposition is similar to the one used by the system Chinook for end-game databases which are based on the total number of pieces on the board rather than the number of pieces for each player (Schaeffer, Culbertson, Treloar, Knight, Lu & Szafron, 1992).

6.2 Parallelization

Parallel algorithms have been used in retrograde analysis to solve games quickly (Lake, Schaeffer & Lu, 1993; Romein & Bal, 2003). A question that arises is whether the AUTRE algorithm could be parallelized. We sketch two possible methods for doing so.

The first is to distribute the open nodes among all available processors. Although this could substantially speed up the algorithm, the subsumption checking and clause removal could cause a significant bottleneck. All decision lists would need to be available to, and consistent across, every processor. In a shared memory environment, this technique would work well, but in a multicomputer system, this would incur a high communication overhead and force processors to synchronize often. If the decision lists were not consistent, then processors could both duplicate work and allow invalid set descriptions to be regressed.

The second approach would be to assign a processor to each automaton state. Since each decision list is relevant for a single automaton state, they would not need to be shared across processors. This could cause every set description to be passed to a different processor after it is generated, but descriptions tend to be small while a large amount of time is spent comparing new tuples against the decision lists. This approach does have a drawback, however, which is that the number of processors would be limited by the number of automaton states, which could be very small.

6.3 Rule Compression and Feature Construction

The set of equivalence classes generated by our approach can be viewed as a list of rules, described as logical formula, for deciding which transition to take. After these rules are created, there are two additional techniques that could be applied: rule compression and feature construction (Muggleton, 1987).

Although the number of classes generated is much smaller than the size of the state-space, many classes could still be generated. The size of the formulae to describe the individual classes could also be large. This is due, in part, to representing each formula as a conjunction of literals and negations of clauses of literals. However, after the set of classes has been finalized, this representation is no longer necessary and the formulae could be rewritten in a more compact form. This would have a negligible effect on the amount of work required for an agent to match classes to its current state.

There are domains that are still too large for our approach to handle. Even for domains that are tractable, it may not be possible or necessary for an agent to store the entire set of rules. In these situations, feature construction could be used to

reduce the amount an agent needs to store, and create rules that can be utilized in the unexplored regions of the state space. One technique that could be applied is to create features out of the number of satisfied positive literals (Fawcett, 1996).

7 Summary

We have presented the AUTRE algorithm, which exploits a compact representation to solve domains without enumerating the underlying state-space. Our technique uses goal regression to generate classes of equivalent problem states that can be regressed as a single unit.

Our algorithm, AUTRE, was tested by applying it to the domain of Connect Four by producing equivalence classes that fully describe the optimal actions and outcomes over the entire state-space. The amount of memory required for our technique was six orders of magnitude smaller than the amount that would have been required for a standard retrograde analysis approach that stores a database of positions and outcomes. Because of this, our method can be used to solve domains that would otherwise be intractable.

8 Acknowledgments

This work was supported by grant IRI-0097218 from the National Science Foundation.

9 References

Allis, L. V. (1988). *A knowledge-based approach to connect four: The game is over, white to move wins*. Master's thesis, Department of Mathematics and Computer Science, Vrije Universiteit.

Bellman, R. E. (1957). *Dynamic programming*. Princeton, N.J.: Princeton University Press.

Boutilier, C., Dearden, R., & Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence, 121*, 49-107.

Bryant, R. E. (1986). Graph-based boolean function manipulation. *IEEE transactions on computers*, 677-691.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence, 62*, 255-301.

Fawcett, T. E. (1996). Knowledge-based feature discovery for evaluation functions. *Computational Intelligence, 12*, 42-64.

Guestrin, C., Venkataraman, S., & Koller, D. (2002). Context specific multiagent coordination and planning with factored mdps. *Proceedings of the Eighteenth National Conference on Artificial Intelligence* (pp. 253-259). Edmonton, Canada.

Hollunder, M., & Nutt, W. (1990). Subsumption algorithms for concept languages. *Proceedings of the Ninth European Conference on Artificial Intelligence* (pp. 348-353). London, England: Pittman.

Howard, R. A. (1960). *Dynamic programming and markov processes*. MIT Press.

Kersting, K., van Otterlo, M., & De Raedt, L. (2004). Bellman goes relational. *Proceedings of the Twenty-First International Conference on Machine Learning*. Banff, Canada: Omnipress.

Lake, R., Schaeffer, J., & Lu, P. (1993). *Solving large retrograde analysis problems using a network of workstations*, Edmonton, Alberta: University of Alberta, Department of Computing Science.

Littman, M. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the eleventh international conference on maching learning* (pp. 157-163). San Francisco, CA.

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 287-292). Milan, Italy: Morgan Kaufmann.

Romein, J. W, & Bal, H. E (2003). Solving the game of awari using parallel retrograde analysis. *IEEE Computer, 36*, 26-33.

Schaeffer, J., Culbertson, J., Treloar, N., Knight, B., Lu, P., & Szafron, D. (1992). A world championship caliber checkers program. *Artificial Intelligence, 53*, 273-289.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 1039-1046). Milan, Italy: Morgan Kaufmann.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence, 112*, 181-211.

Tesauro, G. (2003). Extending q-learning to general adaptive multi-agent systems. *Neural Information Processing Systems 2003*. Vancouver and Whistler, British Columbia, Canada.

van den Herik, H. J., Uiterwijk, J. W. H. M., & van Rijswijck, J. (2001). Games solved: Now and in the future. *Artificial Intelligence Journal, 134*, 277-312.

Waldinger, R. (1976). Achieving several goals simultaneously (pp. 94-136). In Elcock & Michie (Eds.), *Machine Intelligence*. New York: Wiley & Sons.